Stack and Queue

Data Structures – CS2001 – Fall 2021

Dr. Syed Ali Raza

School of Computer Science

National University of Computing & Emerging Sciences

Karachi Campus

Outlines

- Stack techniques.
  - Stack using Array
  - Stack using Linked List

Stack

- A *stack* is a linear data structure that can be accessed only at one of its ends for storing and retrieving data.
- It implements First-in-Last-Out (FILO) or Last-in-First-Out (LIFO) mechanism.
- Main operations include:
- Push: Add an element on top of the stack.
- Peek: return an element from the top of the stack if it is not empty.
- Pop: return and remove an element from the top of the stack if it is not empty.
- isEmpty: return True if no element is in the stack, otherwise False.

Source: https://www.geeksforgeeks.org/

Stack using Array

- Need to define max size of the array.
- It forms a fixed size stack. Therefore, it can become full.
- In the push operation, add an element on top of the stack is there is a space, otherwise, indicate the stack is full.
- All the operations take O(1) time.

Source: https://www.geeksforgeeks.org/

Stack using Linked List

- No need to define max size of the array.
- It forms a dynamic size stack. Therefore, it can’t become full.
- All the operations are done normally.

Source: https://www.geeksforgeeks.org/

Check Balanced Parentheses Using Stack

- Given a string, write a program to examine whether the pairs and the orders of parentheses like “{“, “}”, “(“, “)”, “[“, “]” are correct.

- ***Input**: exp = “[()]{}{[()()]()}”*
  ***Output**: Balanced*

- ***nput**: exp = “[(])”*
  ***Output**: Not Balanced*

Stack Applications

- Expression Conversion: prefix, postfix or infix notation
- Syntax Parsing: like checking program blocks.
- Backtracking
- Parenthesis Checking
- String Reversal
- Function Call

Queue

- A *queue* is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front.
- Here both ends are used.
- It implements First InFirst Out order.
- Main operations are Enqueue, Dequeue, peek, isEmpty and isFull.
- All operations are performed in O(1).

Queue

- Applications include CPU task scheduling, asynchronous data sending and receiving, etc.
- It can be implemented using array or linked list.
- In array implementation, circular queue is more useful.
- Circular queue, also known as ring buffer, moves end of queue to start if the array's upper bound is reached.
- Similarly, start of queue can circle back to initial index of the array (if there is a need).

Queue Practice

- Given a queue, the task is to reverse the queue using another another empty queue.

Priority Queues

- There are certain situations when this First In First Out rule needs to be relaxed somewhat.
- For example, in a hospital's emergency department, patients are queued and served in FIFO order. However, if there is a patient with need for urgent treatment then this patient gets treatment on priority basis.
- In a priority queue, jobs with higher priority are pushed to the front of the queue.

Priority Queues

- Each item in the priority queue is associated with a priority.
- The item with the highest priority is the first item to be removed from the queue.
- If more than one item has the same priority, then their order in the queue is considered.
- It can be implemented as array or linked list.

Priority Queues

- There are two ways of implementing priority queue.

1. Use sorted queue: keep element with the highest priority at the front. Dequeue will be O(1) but enqueuer will be O(n) because we need to traverse the queue to find right position for the new element based on its priority.
2. Use unsorted queue: As usual enqueue elements at the rear, O(1). However, to dequeue we need to search in the queue element with the highest priority, O(n).

Priority Queues

- What will happen if implement as doubly linked list?
  - If using sorted:
    - What will be the enqueue and dequeue times?

- If you implement a priority queue using a list or an array sorted or not then one of the operations, enqueue or dequeue max, takes a linear amount of work.

Recursion Using Stacks

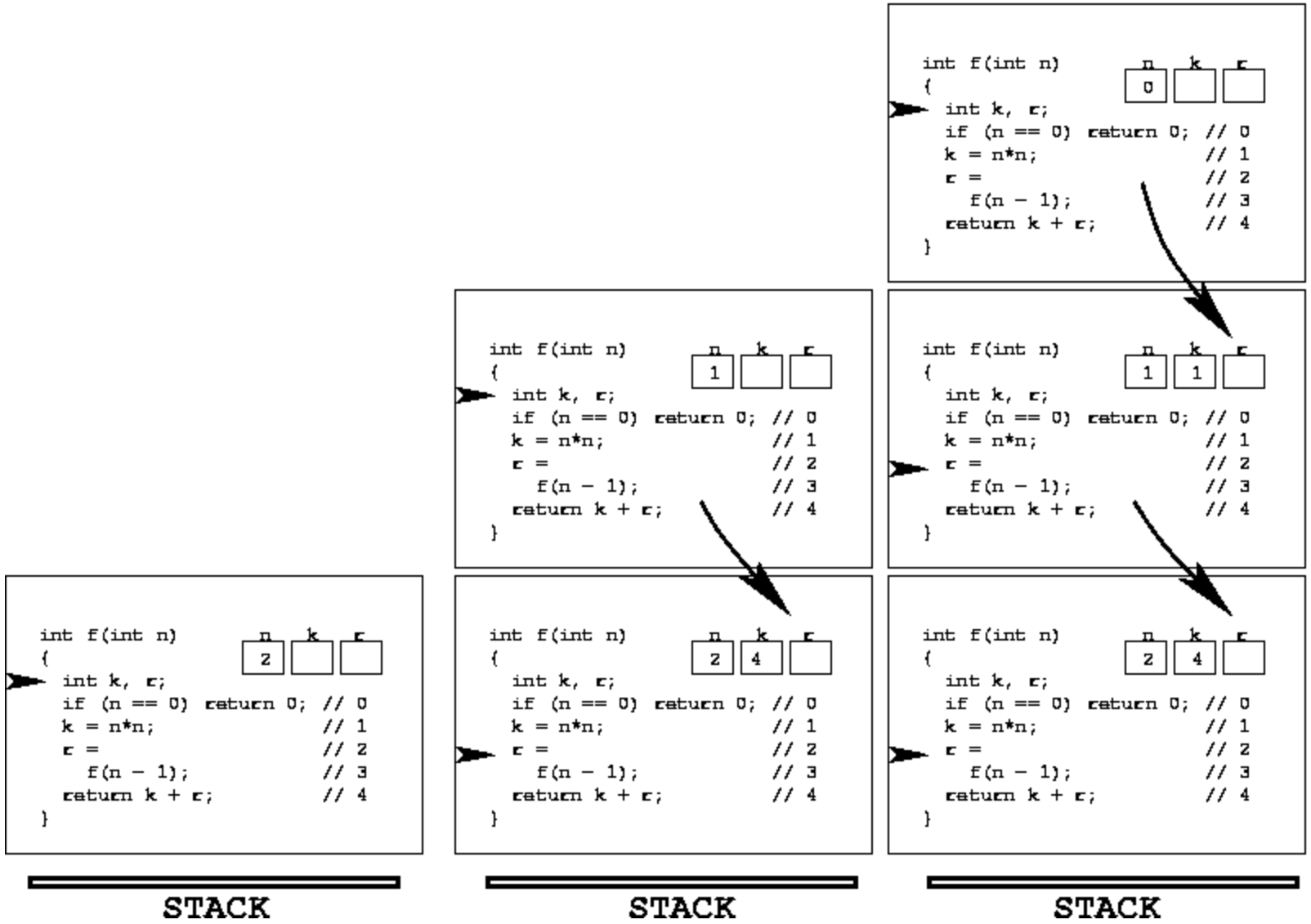
Resource: https://inst.eecs.berkeley.edu/~cs164/fa12/ta-materials/recursionOnStack.ppt

What's Going On?

```
int main() {
  // Get Input
  int x, y;
  cout << "Enter value x: ";
  cin >> x;

  // Call f
  y = f(x);

  // Print results
  cout << y << endl;
  return 0;
}
```

```
#include <iostream>
using namespace std;

int f(int n) {
  int k, r;
  if (n == 0)
    return 0;          // 0
  k = n*n;             // 1
  r =                  // 2
          f(n - 1);    // 3
  return k + r;        // 4
}
```

What's Going On…

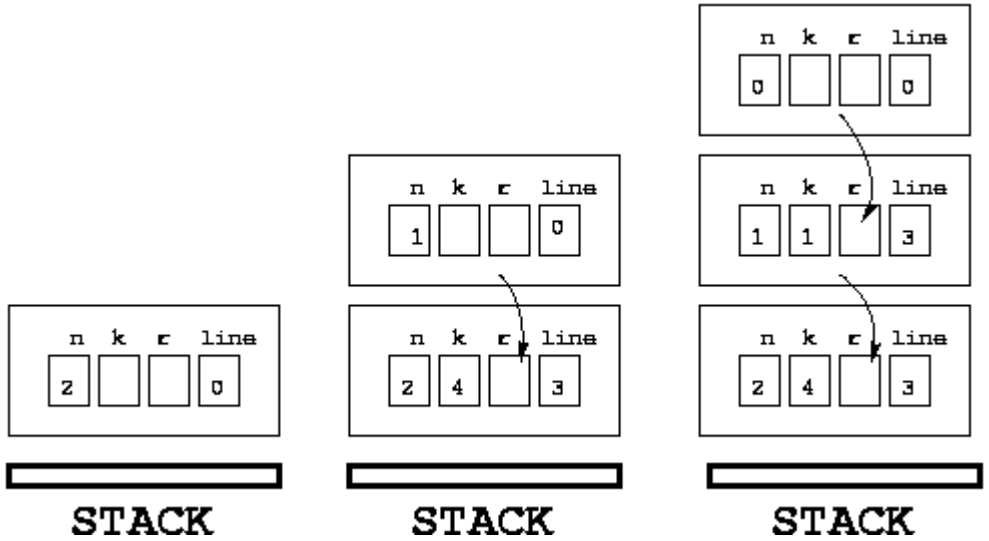


What’s Going On

- Recursive calls to f get pushed onto stack
  - Multiple copies of function f, each with different arguments & local variable values, and at different lines in function
- All computer really needs to remember for each active function call is values of arguments & local variables and the location of the next statement to be executed when control goes back
  - “The stack" looks a little more like :

Real Stack

- Looks sort of like bunch of objects
  - Several pieces of data wrapped together
- We can make our own stack & simulate recursive functions ourselves



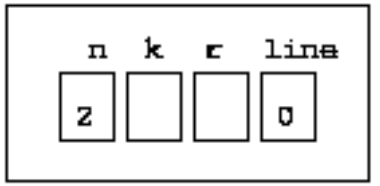
Recursion to Iteration

- Can translate any recursive function to iterative using a stack
  - We use our stack to model “the stack”

- First step…
  - Make a class for the activation records

Activation Record to Class

- We need
  - Argument(s)
  - Local variable(s)
  - Next line to be executed




```
class f_record {
  public:
    int n;              // f's argument
    int line;           // current line in f
    int k, r;           // local variables
    f_record(int arg):n(arg),line(0) { }
};
```

Recursion to Iteration

- Whenever we want to make a call to function ‘f’, we make an instance of the activation class (f_record).
- Keep stack of activation records for calls to f (our example function)
  - To make call like `f(5)`, we create `f_record` with `n` set to 5 & push it on stack

```
Stack<f_record> S;
f_record call(5);
S.push(call);
```

Loop

- Then loop as long as there are activation records on stack
- At each iteration of loop we simulate execution of next line of activation record on top of stack.
- In other words, we pop and push each activation record for each line of the function.

Loop

When activation record on top of stack comes to its return line we'll have variable for it to store returned result in

we're simulating ...

```
int f(int n) {
  int k, r;
  if (n == 0)
    return 0;          // 0
  k = n*n;             // 1
  r =                  // 2
          f(n - 1);    // 3
  return k + r;        // 4
}
```

```
// Initialize recursive function simulation:
// Returned result will be stored in retval
int retval;
Stack<f_record> S;        // stack of active calls
S.push(f_record(x));      // call f(x)

// While there's an active function call left
while(!S.isEmpty()) {
  f_record curr = S.pop();
  int LINE = curr.line++;
  if (LINE == 0) {
    if (curr.n == 0)
      retval = 0;
    else
      S.push(curr);
  }
  else if (LINE == 1) {
    curr.k = curr.n*curr.n;
    S.push(curr);
  }
  else if (LINE == 2) {
    f_record recurse(curr.n - 1);
    S.push(curr); S.push(recurse);
  }
  else if (LINE == 3) {
    curr.r = retval;
    S.push(curr);
  }
  else if (LINE == 4)
    retval = curr.k + curr.r;
}
y = retval;
```

Translation

- Bit more code compared to original function but code is pretty easy to come up with
  - Just have separate case for each line of function
  - Each individual case is straightforward
  - Thing to remember is any recursive function can be translated to iterative segment of code using a stack
    - Procedure for doing it is pretty much what we did with our example