ELEC 278

Tutorial Week 5

2022 Fall
Instructors:
Dr. Jianbing Ni & Dr. Mohammadali Hedayati

Tutorial TAs:
AmirHossein Sojoodi & Shayan Noei

Adapted from slides by
Bryony Schonewille & Shayan Noei

Stacks

A data structure where new data is added at one end, often called the top, and data is removed from the top.

Stack holds a LIFO structure -> Last in First Out
Adding a value to stack is call**ed Pushing**
Removing a value from stack is called **Popping**

Putting a book on a table, then another book on top of the first, then another book on top of the second is referred to as stacking books, and the first book to be retrieved would come off the top – the last one to be placed on the stack

Stacks as Arrays

- An array to hold your stack

```
int stack[100];
```

- A variable to point the index of the top of stack (either place for next value or top value -> initialize as -1)

```
int tos = 0; \\ stands for top of stack
```

- Push (add value to stack):

```
stack[tos] = value;
tos = tos + 1;
```

- Pop (remove value):

```
tos = tos – 1;
popped_value = stack[tos];
```

Stacks as Linked Lists

What do you need to implement it as a linked list?

- At least a singly linked list with a phead pointer -> this is already the minimum implementation of a linked list

What are the pros and cons of using a linked list?

- The advantage of using a linked list is that the stack could grow to an arbitrary size
- The small (really, tiny) disadvantage is that every push onto the stack or pull from the stack may take a little more time than just writing to or reading from a location in an array
- The easiest place to insert a new item in a linked list with only a front pointer is at the front of the linked list, and the easiest place to remove a new item is also from the front of a linked list

Queues

A queue is a data structure that imitates a lineup

Holds a FIFO structure -> First in First Out
Adding a value to a queue is called ***enqueue/insert***
Removing a value from a queue is called ***dequeue/remove***

new data is added to the end of the queue, and data is retrieved from the front of the queue

Queues as Linked Lists

What do you need to implement it as a linked list?

- There should be a way to quickly access the front of the linked list – to remove items – and the end of the linked list – to add items

- At least a singly linked list with a phead and pback pointer or a pback pointer where the last element points to the front

Queues as Arrays

What do you need to implement a queue as an array?

- Implemented as an array with a front and back pointer/variable

Considerations for implementation as an array:

- the code must be able to detect that the queue has become full

- must be able to deal with wrap-around – where the front (or the end) index gets to the end of the array and must be wrapped around to the start of the array – if there is room

Deques

- A deque or double-ended queue is a queue where insertions can take place at both the front and the end, and deletions can take place at both the front and the end

Implementation:

- Double-linked list – a linked list where there is both a forward pointer and a backwards pointer in each node
- The forward links allow a traversal from the front to the end, and the backward links allow a traversal from the end to the front

Example 1

- In accounting, there are two methods for valuing an organization's inventory. In an organization that sells items, they often buy more items for sale before they have sold all the items they currently have. When they sell an item, the question is how much did they make?

- In a way, the question could be interpreted as – did you just sell the first item you bought for inventory or did you just sell the last one? If you assume that prices generally rise over time, you can see where that decision – first one or last one sold – could give significantly different answers.

Example 1

- Suppose you have data as follows:

  BUY 20 1000

  BUY 25 900

  BUY 10 850

  SELL 30 1200

  BUY 15 1050

  SELL 15 1350

- Develop a model to compute earnings under both the LIFO and FIFO accounting methods. Your code will use stacks and queues, and you will need routines that access the top-of-stack and the front-of-the queue.

Example 1 - LIFO

- Use a stack
- Each node is the amount paid for an item

- When a “BUY” is input, add a node to the top of the stack where its value is the amount paid
- When a “SELL” is input, remove the top node of the stack, return the profit = sell price – node cost

Example 1 - FIFO

- Use a queue
- Each node is the amount paid for an item

- When a "BUY" is input, add a node to the back of the queue where its value is the amount paid
- When a "SELL" is input, remove the front node of the queue, return the profit = sell price – node cost

Scheduling Problem

- In a simple task management scheme, jobs that arrive are processed in order. So, for example, if there is a single clerk at a store, then customers line up and each one is served as soon as the previous one finishes with the server (the clerk).
- The total time taken in the queue will depend on how long it takes to serve all the previous jobs that were in the queue.

Scheduling Problem - Setup

- You have a list of arrival and server times (process duration). For example:

Job ID	Arrival	Duration
1	20	5
2	22	4
3	23	6

At that time, a job enters the queue and is immediately removed and sent to the server. The server is now busy until time 25, Next event will be the completion of job 1 at time 25. We can remove job 1 from the server.

the server will become free at time 25 and the next job will arrive at time 23 (The server will be busy until time 29).

The total wait time was 3 minutes for job 2 and 6 minutes for job 3, for a total wait time of 9. This means there is an average wait time in the queue to see a server of 3.

Scheduling Problem – A

- Given conditions:
  - The jobs arrival time
  - How long each job will take
  - One job is processed at a time
  - They are processed in order of arrival.
- **A queue would be an appropriate data structure to use to hold the input data.**
- **Solution:**
  1. Get all the data into a queue. Pull first one out so it can be inspected.
  2. While there is more to do, do it.
  3. Report results.

Scheduling Problem - B

Requirements to implement a simple solution:

- A data structure to hold data entries (arrival and duration)
- A queue to hold a collection of these data structures
- A variable to hold the first one removed from the queue (e.g. first-in-line)
- Completion-time: A server not doing anything has a completion time of negative 1 (-1) to indicate it is idle.
- A variable to hold "*now*"!
- Total wait time
- Count: number of jobs

Scheduling Problem - C

If the server is **idle** and **there is a job in first-in-line**, then we look at the first-in-line arrival time and compare it with *now*.

We have a queue with the first one removed and placed in another variable called "first-in-line".

Items Queue

Job ID	Arrival	Duration
1	20	5
2	22	4
3	23	6

First-in-line

Job ID	Arrival	Duration

Scheduling Problem - D

If the server is **idle** and **there is a job in first-in-line**, then we look at the *first-in-line* arrival time and compare it with *now*.

what we have is a queue with the first one removed and placed in a separate place called "*first-in-line*".

Items Queue

Job ID	Arrival	Duration
2	22	4
3	23	6

First-in-line

Job ID	Arrival	Duration
1	20	5

Scheduling Problem - E

If the server is idle and there is a job in first-in-line, then we look at the *first-in-line*'s arrival time and compare it to *now*.

If *now* <= *first-in-line*'s arrival time: advance *now* to the arrival time; as soon as the arrival time becomes equal to *now,* the job has arrived and the server immediately given to the job with no wait. Note how long the server will be busy for. Add 1 to job count and 0 to total *wait* time.

First-in-line

Job ID	Arrival	Duration
1	20	5

Now = 15

Now = 20
Count++
Total-wait-time = 0

Scheduling Problem - F

If the server is idle and there is a job in first-in-line, then we look at the *first-in-line*'s arrival time and compare it to *now*.

If *now* > *first-in-line*'s arrival time: then move the *first-in-line* to the server, noting how long the server will be busy. Add 1 to job count and store "*now – arrival*" to the total *wait* time.

First-in-line

Job ID	Arrival	Duration
1	20	5

Now = 25

Now = 25
Count++
Total-wait-time = 5

completion time = now + server busy time.

Scheduling Problem - G

If the queue is not empty, then we must figure out what happens next. We have some notion of what time it is now, say a variable called now.

With a busy server, nothing will change until server finishes. So, with busy server, **just advance the clock to the completion time – now + server busy time.**
Now we have one of the cases we have already covered, and so we proceed until the queue is empty

Write a simple simulation of this scenario with single cashier. Develop your own set of data and see how one big job early in the day has a big effect on wait times.

Reverse Polish Notation (RPN)

- RPN – "Reverse Polish Notation" (or postfix notation) writes expressions differently
  - E.g.: (4+5)*(8+9) → 4 5 + 8 9 + *
- We have prefix, infix, and postfix notations.

How to evaluate the postfix expression?

- Save 4, save 5
- Get most recent pair of saved values and add them – save result
- Save 8, save 9
- Get most recent pair of saved values and add them – save result
- Get most recent pair of saved values and multiply them – save result
- When we're done, the answer is the last (only) saved value.

RPN Problem

- This notation is useful as compilers process expressions and generate code to evaluate expressions. They essentially translate the normal algebraic expression into an RPN version, then generate the code based on the RPN version. The result from above would be:

```
push 4
push 5
add
push 8
push 9
add
multiply
```

RPN Problem

Step 1.

Fetch a string containing an RPN expression.

- 1.1 Prompt user to enter a string
- 1.2 Read user's input.

RPN Problem

Step 2.

Step 2 says “If possible, compute expression and display answer.”

- 2.1 result = compute (input string, &answer)
- 2.2 if (result == OK) print answer else print error message

RPN Problem

Operations example for compute:

```
bool add ()
{
    int a, b, answer;
    // assume pop() returns TRUE if it works
    // better remember this when designing stack

    if (pop(&a) && pop(&b)) {
        answer = a + b;
        push (answer);
        return TRUE;
    }
    return FALSE;
}
```

RPN Problem

```
Bool error = FALSE; // None yet, anyway
while (!error) {
    // a token is a number or an operator, in an RPN
    error = get next token from string;
    if (!error) {
        if (token is a number)
            push number;
        else (token is operator) {
            switch (operator type) {
                case addoperator:
                    err = !add();
                case suboperator:
                    etc;
            }//ends switch
        }
    }
    else { return error; }
}
```

Recursion – Problem 1

The Fibonacci numbers are defined as follows:

Fibonacci (0) = 1

Fibonacci (1) = 1

Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

Implement a function to compute the first 100 Fibonacci numbers, using the formula as shown.

Recursion – Problem 1

Implement a function to compute the first 100 Fibonacci numbers, using the formula as shown.

```
int  fib(int n) {
      if (n == 0) {
            return 0;
      } else if (n == 1) {
            return 1;
      } else {
            return fib(n – 1) + fib(n – 2);
      }
}
```

Base cases: must have them to stop a recursive algorithm