Informatics II, Spring 2024, Exercise 07

Publication of exercise: April 15, 2024
Publication of solution: April 22, 2024
Exercise classes: April 22 - April 26, 2024

Learning Goal

- Learn how to implement stacks and queues as arrays and singly linked lists
- Being able to use pointers-to-pointers in order to allow multiple instances of data structures.
- Being able to apply stacks, queues, and principles "last in, first out", "first in, first out" to solve problems.

Task 1 [Easy]

Task 1.1 (Implementation of Stacks)

The goal of this task is to implement stacks in C, using arrays and linked lists.

Assume that we use a single, global stack `S`. The implementation should support the following operations:

- `new()` initializes the stack `S`.
- `is_empty()` returns `True` if the stack is empty.
- `pop()` pops the topmost element of the stack and returns it. If there is no element on the stack, return `-1`.
- `push(x)` takes as argument a positive integer `x`, and pushes `x` on top of the stack.

These operations should have a runtime complexity of $O(1)$ in your implementation.

a. Implement the operations described above, using stacks represented as arrays.

b. Implement the operations described above, using stacks represented as singly linked lists.

c. Adapt the implementation of stacks from b) to allow support for multiple instances of stacks. More precisely, your implementation should now support the operations:

   - `new()` creates a new (empty) stack and returns it.
   - `is_empty(S)` takes as argument a stack `S` and returns `True` is the stack is empty.

- `pop(S)` takes as argument a stack `S`, pops the topmost element of the stack and returns it. If there is no element on the stack, return `-1`.
- `push(S, x)` takes as arguments a stack `S` and an integer `x`, and pushes `x` on top of the stack.

Task 1.2 (Implementation of Queues)

The goal of this task is to implement queues in C, using arrays and linked lists.

Assume for now that we use a single, global queue. The implementation should support the following operations:

- `new()` initializes the global queue.
- `is_empty()` returns `True` if the queue is empty.
- `dequeue()` removes the front element of the queue and returns it. If there is no element in the queue, return `-1`.
- `enqueue(x)` takes as argument a positive integer `x`, and inserts `x` at the back of the queue.

These operations should have a runtime complexity of $O(1)$ in your implementation.

a. Implement the operations described above, using queues represented as circular arrays.

b. Implement the operations described above, using queues represented as singly linked lists.

c. Adapt the implementation of queues from b) to allow support for multiple instances of queues. More precisely, your implementation should now support the operations:

   - `new()` creates a new (empty) queue and returns it.
   - `is_empty(Q)` takes as argument a queue `Q` and returns `True` if it is empty.
   - `dequeue(Q)` takes as argument the node pointing to queue `Q`. The function removes the front element and returns it. If there is no element in the queue, return `-1`.
   - `enqueue(Q, x)` takes as arguments the queue `Q` and an integer `x`, and inserts `x` at the back of the queue.

Task 2 [Medium]

You are given a string containing open and closing parentheses. The goal of this task is to determine whether the given sequence of parentheses is valid.

A sequence of parentheses is *valid* if every opening parenthesis has a corresponding closing parenthesis, and matching parentheses are in the right order: first an open parenthesis, then a closed parenthesis.

Examples of valid sequences are:

- `(())`
- `()()()`
- `(())((()))`

Examples of invalid sequences are:

- `(()` (the first open parenthesis does not have a corresponding closing parenthesis)
- `(()))()` (one closing parenthesis has no corresponding opening parenthesis)
- `)(` (wrong order of parentheses)

a. Write a function `bool validate_parentheses(char *s)` which takes a string `s`, representing the parentheses sequence, and returns `True` if the sequence is valid. Use a stack in your solution.

   Hint: Go through the input from left-to-right, and try to track the parentheses using a stack.

b. Consider now an *extended* parenthesis sequence, which additionally allows square brackets: `[]`. The rules from a) for a valid sequence still apply here, with the additional constraint that each opening parenthesis must have a corresponding closing parenthesis of the **same** type.

   Examples of valid extended parenthesis sequences are:

   - `[()][]()`
   - `[[()[()]]]`

   Examples of invalid sequences are:

   - `[[()]`
   - `[(])` (incorrect type of opening and closing parentheses)

   Write a function that checks whether an given extended sequence is valid. Can you extend your approach from a) to also solve b)?

Task 3 [Medium]

We define a set of numbers C to be a *Collatz* set if it fulfills the following properties:

- $1 \in C$
- If $x \in C$, then also $3x + 1 \in C$ and $3x + 2 \in C$.

A Collatz set has infinitely many numbers. The first few terms of a Collatz set are:

$$1, 4, 5, 13, 14, 16, 17, 40, ...$$

We can generate the terms of a Collatz set as follows: start from the number 1, then insert the elements $4 = 3 \cdot 1 + 1$ and $5 = 3 \cdot 1 + 2$ into the set, then generate the elements derived from 4 and 5 and so on.

Write a function `void generate(int n)` that prints the first (smallest) `n` elements of a Collatz set. Use a queue to solve this task.

Task 4 [Medium]

Write a C function `sort_stack(S)` that receives a stack of integers `S` as argument, sorts the values in `S` such that the lowest value is at the top, and returns the sorted stack. You are **only** allowed to use stacks to sort the input stack (do not use arrays, queues etc.). You can assume that the stacks you are allowed to use support the following operations:

- `new()` creates a new (empty) stack and returns it.
- `is_empty(S)` takes as argument a stack `S` and returns `True` is the stack is empty.
- `pop(S)` takes as argument a stack `S`, pops the topmost element of the stack and returns it. If there is no element on the stack, return `-1`.
- `push(S, x)` takes as arguments a stack `S` and an integer `x`, and pushes `x` on top of the stack.

Example:

```
// If S is [2, 3, 1] from the bottom to the top, then:
sort_stack(S) -> [3, 2, 1]
```