Informatics II, Spring 2024, Solution 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.

This solution presents a high-level idea of the solution. See the attached code (`task01_1a.c`, `task01_1b.c`, `task01_1c.c`) for implementation details.

a. Store a global array `S` and the top of the stack (noted in the implementation with the term `top`). The `top` will show us the position of the first free cell in the array, where we can put the next element that will be push onto the stack.

b. The top of the stack will be the head of the list. When we make a `push` operation, we will create a node with the pushed number and insert that node at the beginning of the list. When we do a `pop`, we erase the first node of the list.

c. Functions cannot directly modify the value of their given arguments. Since the root of the list can change after an operation (from non-`NULL` to `NULL`, for example), we cannot directly use simple pointers as function arguments. To keep a constant function argument but allow modification, we use pointers to pointers: we give to the functions a double pointer (which will stay constant) that points to the list root, which can change. The implementation idea is otherwise very similar to subtask b).

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.

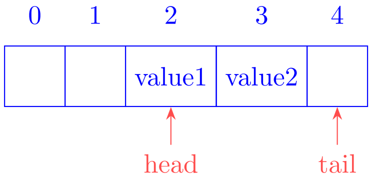
This solution presents a high-level idea of the solution. See the attached code (`task01_2a.c`, `task01_2b.c`, `task01_2c.c`) for implementation details.

a.. Store a global array `Q` and two variables `head` and `top` which are the indexes of the array cells where the queue starts and ends, respectively. `head` refers to the first cell of the queue; `tail` refers to the next empty cell that will be written when doing an `enqueue` (so `tail-1` is the tail cell of the queue).

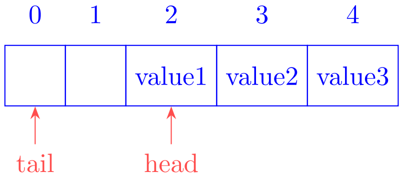
We use a circular array for implementing the queue. This is still a regular array, but instead of simply incrementing the head/tail, we add a modulo operation, which allows the head and tail to "wrap around" the array. For example, if the array has a size of 5 and:

```
head = 2
tail = 4
```

then the array looks like this:



If we now enqueue (add one element to the queue), `tail` becomes `(4+1) % 5 = 0` and wraps around the array:



b.. Store the head and tail of the queue as pointers to the first and last node of the linked list respectively. When enqueuing, insert a node at the beginning of the list. When dequeuing, remove the node from the end of the list.

c.. Similarly to subtask c) for implementing stacks, we will need to use pointers to pointers. However, since we have to store now two pointers (a head and a tail), we now also create a `struct` that stores the head and tail pointers. This also allows us to (easily) return a newly-created queue from the function `new()`. Otherwise, the idea is the same as in the previous subtask.

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)?

Idea: iterate through the string from left to right.

Consider this parenthesis sequence:

$$((())())()$$

We seek to first validate the first pair of parentheses (highlighted in red):

$$((())())()$$

But, in order to validate that pair, we first have to validate the parenthesis inside (highlighted in blue)

$$((())())()$$

We can see here a sort of recursive mechanism: to validate the outer parentheses, we have to first validate the inner part, which on its own may recursively have to be validated.

This allows two ways to solve this problem:

- a recursive function
- a stack

This solution will describe the stack solution[1].

To validate the first pair in parenthesis (in red), we first have to validate pairs of parenthesis that start later (in blue). In other words, the first opening parenthesis which is encountered is the last to be paired with its matching closing parenthesis. In other words, the problem follows a first-in-last-out structure.

We use a stack of characters as follows:

- whenever we encounter an opening parenthesis, push it onto the stack;
- whenever we have a closing parenthesis, check first if there is an opening parenthesis on the stack (at the top). This is our corresponding opening parenthesis. If this is the case, then pop from the stack. Otherwise, there is no open parenthesis for this current closing parenthesis, so the sequence is invalid.
- After going through the entire string, check at the end if the stack is empty. If it is not empty, we still have unmatched opening parenthesis, like in this case:

$$(()$$

For subtask b), the algorithm is mostly the same, except that for closing parenthesis you must additionally check that the stack top has an opening parenthesis of the same type.

[1] Actually, any recursive solution can be solved using a stack; stacks are basically the way recursion is implemented on computers.

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.

Implement the generation idea described in the task: start from the number 1, then generate $4 = 3 \cdot 1 + 1$ and $5 = 3 \cdot 1 + 2$, then generate the numbers derived from 4 and 5 and so on.

To generate the numbers, use a queue. Initially, this queue will contain the number 1. Then, generate as follows: Take the element from the head of the queue (dequeue); let's call it x. Then, push the numbers $3x + 1$ and $3x + 2$ into the queue. Whenever we take one number from the queue, print it; stop after we printed `n` numbers.

To see why it works, execute a few steps of this generation: first, only the number 1 will be in the queue. Then, we pop 1, print it, then add the next Collatz terms in the queue: 4 and 5. Then, pop 4, print it, and add $13 = 3 \cdot 4 + 1$ and $14 = 3 \cdot 4 + 2$. Then, pop 5, print it, and add the numbers $16 = 3 \cdot 5 + 1$ and $17 = 3 \cdot 5 + 2$. Now, the queue contains the numbers $13, 14, 16, 17$.

The terms in the queue are always in increasing order. Popping from the queue means looking at the smallest element which is not yet printed and used for generation; the elements generated from the popped element are always larger than all other previous numbers in the queue. This happens because of the generation rules ($x \to 3x + 1, x \to 3x + 2$).

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]
```

The idea is to simulate an already existing sorting algorithm, but use stack operations instead of swaps or assignments.

Look at the basic sorting algorithms: bubble sort, selection sort and insertion sort. Given enough helper stacks, all of these can be simulated. However, swaps in the middle of the array (which are used in bubble sort and selection sort) make a stack solution very messy. So, we are left with insertion sort.

The idea of the insertion sort is to always keep a sorted prefix, and to try inserting new numbers one by one into this prefix while keeping it sorted. We will store this prefix in the given stack S. Since the task asks us for the smallest number to be at the top, we will store the prefix in descending order (starting from the bottom to the top).

If the prefix is currently

$$9, 7, 5, 2$$

and we want to insert 6 into this prefix, it is enough to "move away" the sequence of numbers smaller than 6, insert 6, then put back those smaller numbers:

$$\begin{gathered} 9, 7 \quad 5, 2 \\ 9, 7, 6 \quad 5, 2 \\ 9, 7, 6, 5, 2 \end{gathered}$$

Let's call x the number that we want to insert into the prefix. This "moving away" of smaller numbers can be done the following way:

1. Move the numbers smaller than x from the stack S into a temporary stack.
2. Push x into the stack S.
3. Move the numbers from the temporary stack back into S.

Since the smaller numbers are pushed two times (first into the temporary stack, then back into S), their final order will not be changed, because two pushes means inverting their order twice (similarly to $-(-x) = x$ not changing the sign of a number x).

Thus the stack S after inserting x will look like this:

$$\underbrace{9, 7,}_{\geq x} \underbrace{6}_{x} \underbrace{5, 2}_{\leq x}$$

Since all three parts are in decreasing order and the left part has numbers larger than the right part, the prefix will remain sorted in descending order also after inserting the value x.