

# Notes 19.0: Queues and stacks

COMP9021 Principles of Programming

*School of Computer Science and Engineering  
The University of New South Wales*

2014 session 1

# The queue data type (1)

Queues are lists with two special properties:

- New items can be added only to the end of the list.
- Items can be removed from the list only at the beginning.

They are **first-in first-out (FIFO)** data structures.

The interface is defined in the header file `tailored_queue.h` and implemented in `tailored_queue.c`. A simple test is provided in `test_tailored_queue.c`.

The queue data type can be used to model queues in real-life situations. Key parameters are:

- the average time,  $\lambda$ , between two successive arrivals of customers joining the queue (say in minutes);
- the average time,  $\mu$ , needed to serve a customer when her turn comes (say in minutes).

## The queue data type (2)

When running a simulation, we also need to know how long the simulation should be run (say in hours).

Natural assumptions are that the inter-arrival time between successive customers and the service time for a given customer are modelled by exponentially distributed random variables with an expected value of  $\lambda$  and  $\mu$ , respectively.

Natural and important theoretical and measured entities are:

- the average number of customers in the queue including those being served;
- the average number of customers in the queue waiting to be served;
- the average waiting time for a customer, excluding service time
- the average waiting and service time for a customer.

[simulate\\_queue.c](#) estimates these values both theoretically and using a simulation.

# The stack data type (1)

Stacks are lists, visualised vertically, with two special properties:

- New items can be added only to the top of the list.
- Items can be removed only from the top of the list.

They are **last-in first-out (LIFO)** data structures.

The interface is defined in the header file [tailored\\_stack.h](#) and implemented in [tailored\\_stack.c](#). A simple test is provided in [test\\_tailored\\_stack.c](#).

We implement the data structure using arrays allocated on the heap rather than linked lists.

Popping an element does not deallocate memory: the index indicating the top of the stack is moved down, and the value will just be overwritten if another element is pushed onto the stack.

## The stack data type (2)

When the array becomes too small to push new datum, more memory is allocated with a call to `realloc()`, that tries to extend the chunk of memory and keep the same pointer as passed as first argument, in which case the same pointer is returned; if that is not possible another chunk of memory is put aside, the value stored in the old location are copied, and a pointer to the start of the new chunk of allocated memory is returned.

## Using a stack (1)

Stacks can be used to evaluate postfix expressions. For instance, in order to evaluate `3 4 + 2 x`, we would use a stack as follows.

Symbol processed	Action	Stack
none	none	empty
3	Push 3	3
4	Push 4	4, 3
+	Pop 4, Pop 3, compute $4 + 3 = 7$	empty
None	Push 7	7
2	Push 2	2, 7
x	Pop 2, Pop 7, compute $2 \times 7 = 14$	empty
None	Push 14	14

This is implemented in [postfix.c](#).

## Using a stack (2)

Stacks can also be used to check whether the parentheses in an expression are well balanced. For instance, in order to check that `()((()()))` is well balanced, we would use a stack as follows.

Symbol processed	Action	Stack
none	none	empty
(	Push (	(
)	Pop )	empty
(	Push (	(
(	Push (	((
(	Push (	((
)	Pop )	((
(	Push (	((
)	Pop )	((
)	Pop )	(
)	Pop )	empty

## Using a stack (3)

- A sequence of parentheses is well balanced if starting with an empty stack, and pushing every opening parentheses that is being processed, we can achieve the following.
  - Every time we process a closing parenthesis, we can pop a corresponding opening parenthesis from the stack;
  - we end up with an empty stack.
- A stack is also used to implement function calls.