More on organisation

Tutorials

see your group on Syllabus page in Canvas

Office hours

Tuesday 11:15–12:15 in 242 (after the tutorial)

Assignments deadlines

1st assignment: 12 February 2018

2nd assignment: 19 March 2018

(submit before the lecture)

Shared document

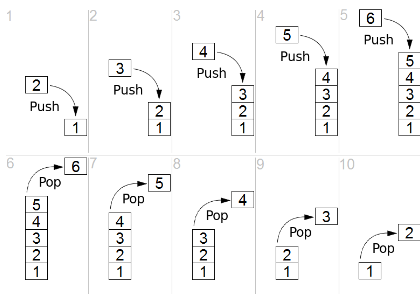
`http://cs.bham.ac.uk/~txj300/dsa2018-doc`

Stacks

Stacks = LIFOs (Last-In-First-Out)

Stack is an abstract data type defined by its three operations:

- `push(x)` puts value `x` on the *top* of the stack
- `pop()` takes out a value from the *top* of the stack
  If there are no values in the stack, it raises
  `EmptyStackException`.
- `is_empty()` says whether the stack is empty



(picture source: wikipedia)

Stack is an abstract data type. A stack is a list of values. The two ends of the list are called the *bottom* and *top*. We *push* a value (add it to the top of the stack) and *pop* a value (remove from the top of the stack). Thus a stack behaves in a Last In First Out (LIFO) manner. If we try to pop from an empty stack, we get an `EmptyStackException`. In theory we should be able to push any number of values, but in practice that won't be possible and we will get an exception if we run out of memory.

We can have a stack of any type of value, e.g. a stack of integers, a stack of strings, a stack of Booleans etc.

As a part of the specification of stacks it is usually also said that `push(x)` followed by `is_empty()` gives `false`, and `push(x)` followed by `pop()` gives `x` back.

Since we see a stack as an abstract data type, we do not specify how it is implemented.

Example

Suppose we create an empty stack and we push 3, push 5, push 2 and pop; we get 2. Suppose we then pop; we get 5. Suppose we push 1, push 8, then pop. Pop again three times, what do we get?

Usage

For example, when we are solving tasks with dependencies. Imagine that we want to complete a task A, `push(A)`, and

1. A depends on B and C $\Longrightarrow$ `push(B)` and `push(C)`
   (ok, we need to solve C first but ...)
2. C depends on D $\Longrightarrow$ `push(D)`.

Once we complete D (`pop()`), C (`pop()`), and B (`pop()`), we can also complete A (`pop()`).

(More verbose example of usage:)

One reason that stacks are useful is that sometimes, in order to complete job A we must first do job B and then job C, but in order to complete job B we must first do job D, and in order to do that we must first do job E and job F. So we have a job stack. While we are doing E it is at the top of the stack, with D below, and B below that, and A at the bottom. When we start a job we push it onto the stack, and when we complete a job we pop it.

Another example: the most natural way of evaluating an expressing written in *reversed polish notation* is by using stacks.

Stacks as linked lists

30
15
11
5

To store a stack as a linked list we pick the faster of the two options:

1. the top is at the beginning, i.e. $\langle 30, 15, 11, 5 \rangle$
2. the top is at the end, i.e. $\langle 5, 11, 15, 30 \rangle$

Question: Which one is better and why?

Since inserting and deleting from the beginning of a linked list is constant, the first option is better. In other words, we take

- `push` = `insert_beg`
- `pop` = `delete_beg`
- `is_empty` (for stacks) = `is_empty` (for linked lists)

This way every operation on stacks takes constant time.

The second option would mean that `push` = `insert_end` and `pop` = `delete_end`. Then, even if we stored the position of the end of the linked list (to make sure `insert_end` is fast), `delete_end` would still be slow (linear time) and so the second option is not reasonable.

Stacks as arrays

Stacks are usually stored as arrays:

```
1 // Initialize an empty stack:
2 stack = new int[MAXSTACK];
3 stack_size = 0;
```

(With the bottom of the stack stored on the 0th position of the array.)

In the tutorials, we show how to implement `push` and `pop` for this representation in constant time.

$\Longrightarrow$ No matter if we store stacks as linked lists or arrays, in both cases, `push`, `pop`, and `is_empty` finish in *constant time*.

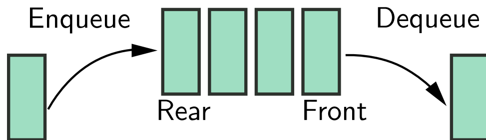
Storing stacks as arrays has the advantage that we avoid calling `allocate_memory` all the time (this takes time, even if it is done automatically for us, like in Java). On the other hand, we need to know the maximum size of the stack in advance.

Queues

Queues = FIFOs (First-In-First-Out)

Queue is an abstract data type defined by its three operations:

- `enqueue(x)` puts value `x` at the *rear* of the queue
- `dequeue()` takes out a value from the *front* of the queue
  If there are no values in the queue, it raises
  `EmptyQueueException`.
- `is_empty()` says whether the queue is empty



Queue is an abstract data type. A queue is a list of values (e.g. integers, Booleans, ...). The two ends of the list are called the *rear* and *front*. We *enqueue* a value (add it to the rear of the queue) and *dequeue* a value (remove it from the front of the queue). Thus a queue behaves in a First In First Out (FIFO) manner. If we try to dequeue from an empty queue, we get an `EmptyQueueException`. In theory we should be able to enqueue any number of values, but in practice that won't be possible and we will get an exception if we run out of memory.

Example

Starting from an empty queue, enqueue 4 and 3, dequeue, then enqueue 1 and dequeue two times. What is the last value you get? What would happen in the next dequeue?

Usage

A typical application of queues is a print queue: files are sent to the queue for printing and are printed in the order in which they were sent.

After sending a file, you know that only the jobs currently in the queue will be printed before yours.

Queues are useful whenever we have need to process tasks in the order in which they came. We demonstrate this in the printer queue but there are many more examples (e.g. web server when serving websites).

Notice that since the tasks in a print queue are executed in the order in which they came, there is no priority. Even as a lecturer I have to wait for all student tasks that came before mine to finish before my file gets printed.

Queue as a linked list

In order to have an efficient implementation we need to store the location of the last element in the linked list.

We have two options again:

1. Front at beginning of the linked list, rear at end
2. Rear at beginning of the linked list, front at end

Question: Which one is better and why?

Think of how would enqueue and dequeue be implemented if we did (1) or (2).

For (1) to enqueue, as long as we keep track of the end, constant time, to dequeue, remove first node and change head poitner, constant time.

For (2) to enqueue, allocate new and change the head pointer, to dequeue, we don't know the address of the penultimate node and so we have to find it first, linear time.

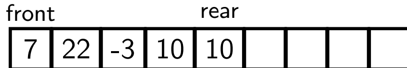
(1) makes sense, (2) doesn't. In other words, we take

- `enqueue` = `insert_end`
- `dequeue` = `delete_beg`

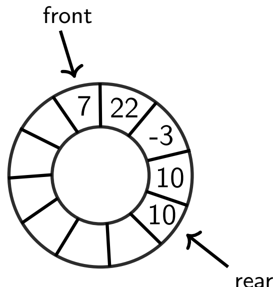
This way both enqueue and dequeue take constant time.

Queue stored in an array (1st attempt)

Whenever we `enqueue` (resp. `dequeue`) we move `rear` (resp. `front`) to the right:



We eventually run out of space! To fix this, every time we `dequeue`, move everything to the left. It works but is slow!



Instead we use a circular storage of a queue. We move Front and Rear clockwise.

Works beautifully in theory but how do we implement it in an array?

If we know that the size of queue is limited during the run of our program, we can store the queue as an array. We store the position of the front and the rear of the queue in variables `front` and `rear`, respectively. Then, values stored in the queue are stored on the positions `front`, `front+1`, ..., `rear`. To `enqueue` we increase `rear` by `1` (move it to the right) and store the new value on the position `rear`. To `dequeue` we read out the value on position `dequeue` and move `dequeue` to the right.

However, this way, we eventually reach the end of the array and we can't move `rear` anymore. If we could store our queue in a "circular" array, we could keep moving `front` and `rear` clockwise forever.

Digression: `div` and `mod` (maths break)

For numbers a, b with $b > 0$ we write `a div b` to mean the result of dividing a by b and discarding the remainder, and we write `a mod b` to mean the reminder.

For example:	$123 \text{ div } 10 = 12$	$58.7 \text{ div } 10 = 5$
	$123 \text{ mod } 10 = 3$	$58.7 \text{ mod } 10 = 8.7$

Example

Formula racing: Assume that one lap in a race is 600 meters. Then a formula which has driven 1500 meters has done `1550 div 600 = 2` laps and `1550 mod 600 = 350` meters in the third lap.

Note that `a div b` is always an integer and $0 \leq$ `a mod b` $< b$ such that

$$(a \text{ div } b) * b + a \text{ mod } b = a.$$

Consequently: $-123 \text{ div } 10 = -13$ and $-123 \text{ mod } 10 = 7$.

`mod` can be used to define flooring (rounding down to an integer), we have

$$\lfloor x \rfloor = x \text{ div } 1 \quad \text{and} \quad x \text{ div } y = \lfloor x/y \rfloor$$

(very useful when we do complexity)

Note that in Java `Math.floorDiv` and `Math.floorMod` behave better than `%` and `/`, respectively, because the latter interpret the operations on negative numbers incorrectly.

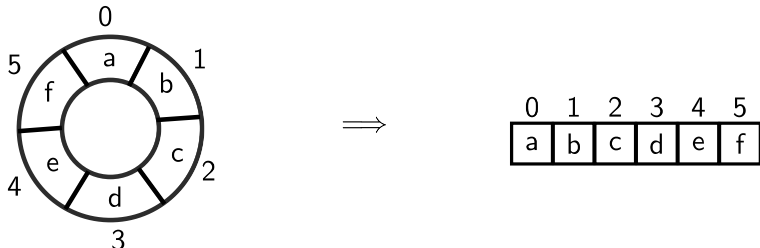
More examples:

- $100 \bmod 10 = 0$ (100 is divisible by 10)
- $-18 \text{ div } 10 = -2$ (move backwards 2 * 10)
- $-18 \bmod 10 = 2$ (and then 2 steps forwards)
  (it has to be in the bounds: $0 \leq -18 \bmod 10 < 10$)

- Floor: $\lfloor 14.3 \rfloor = 14, \quad \lfloor 7.0 \rfloor = 7, \quad \lfloor -5.3 \rfloor = -6$
- Ceiling: $\lceil 14.3 \rceil = 15, \quad \lceil 7.0 \rceil = 7, \quad \lceil -5.3 \rceil = -5$

Representing circles

Positions in the circle, numbered clockwise, correspond to the positions in the array:



Example

For a position `pos` in the circle (e.g. `pos = 5`), moving clockwise by two positions is computed as `(pos + 2) mod 6`.

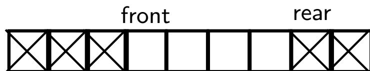
Or in general `(pos + 2) mod circle_length`.

Circular queue

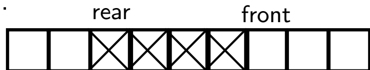
```
1 // Initialize an empty queue:
2 queue = new int[MAXQUEUE];
3 rear  = 0;
4 front = 0;
```

We store values in the queue in between positions marked by `rear` and `front`, that is,

- if $front \leq rear$ then the queue consists of the entries on positions $front, front + 1, ..., rear - 1$:



- if $rear < front$ then the queue consists of the entries on positions $front, front + 1, ..., MAXQUEUE - 1$ and $0, 1, ..., rear - 1$:



Contrary to our first attempt, we store `rear` one position ahead of where the last element of the queue is stored. This is to simplify calculations. For example, the queue is empty whenever `front == rear`.

Consequently, the queue is full whenever `front` is equal to `rear + 1 mod MAXQUEUE` and trying to `enqueue` in such case raises an exception. However, even though the queue is full, one space in the array is not being used. This could be avoided if one really wants but it complicates matters and, for example, checking whether the queue is empty would not be as simple as just checking that `front == rear`.

Circular queue

```
1 // Initialize empty queue:
2 queue = new int[MAXQUEUE];
3 rear  = 0;
4 front = 0;
```

```
1 boolean is_empty () {
2   return rear == front;
3 }
```

```
1 void enqueue (int a) {
2   int newrear = (rear+1) mod MAXQUEUE; // move clockwise
3
4   if (newrear == front)
5     throw OutOfMemoryException;     // the queue is full
6
7   queue[rear] = a;
8   rear = newrear;
9 }
```

We do `mod MAXQUEUE` to make sure we stay in the circle.

Final points

As we will see, stacks and queues are used in many algorithms.

We often just say “make a stack” or “make a queue” and we don’t care how they are implemented.

We know that, whether it is as an array or linked list, it can be done efficiently.