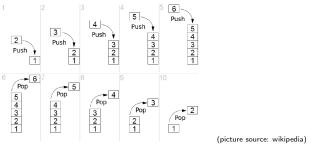
Stacks and queues

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`.
- `isEmpty()` says whether the stack is empty



(picture source: wikipedia)

1

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 `isEmpty()` 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.

Examples

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

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

2

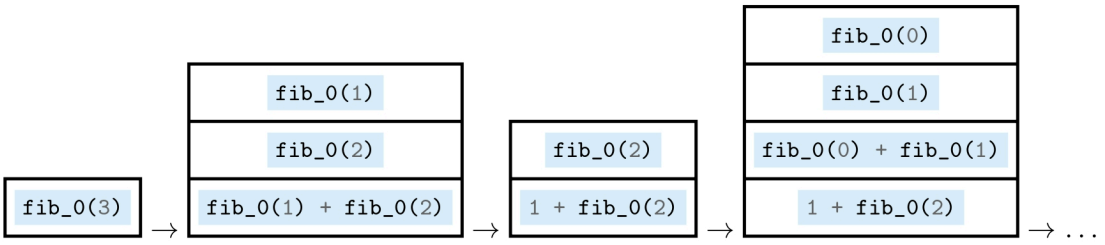
(More verbose example of usage:)

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. Using a stack, we can push the primary job onto the stack first and each time we push any job onto the stack, we follow it by pushing the jobs that it depends on. So long as you then execute the jobs in the order that pop retrieves them, all the jobs will only be executed when the jobs they depend upon are complete.

Space analysis of `fib_0`

```
1  int fib_0 (int n) {
2    if (n < 2)
3      return n;
4    else
5      return fib_0(n-1) + fib_0(n-2);
6  }
```

Each recursive call is put onto a stack.
How big will the stack grow?
For example, computing `fib_0(3)` . . .



3

Another example: T

- The most natural way of evaluating an expression written in reverse Polish notation is by using stacks.
- This is because sub-expressions have to be evaluated before the higher level expressions that must use them, hence evaluating a sub-expression is a job that must be completed before we can evaluate the higher level expressions

- Stacks are heavily used in applications that involve exhaustive searches of some problem space and in constructing and searching tree structures.

Stack ADT

Here is a possible list of operations for a Stack ADT (many variations are possible).

- `EmptyStack` constructor that returns an empty stack.
- `stack.push(element)` pushes an element on top of the given stack.
- `stack.pop()` returns the element at the top of the stack removing it from the stack.[1]
- `stack.isEmpty()` reports whether the stack is empty.

[1] Triggers error if the stack is empty

4

Stacks as linked lists

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?

5

- Since inserting and deleting from the beginning of a linked list is constant, the first option is better.
- In other words, we take
  - $push = insert\_beg$

    - $pop = delete\_beg$
    - $isEmpty(\text{for stacks}) = isEmpty(\text{for linked lists})$
- 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.

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`
- `isEmpty` (for stacks) $=$ `isEmpty` (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

One can implement Stacks using a simple array in Java:

```
// Initialize an empty stack and set the maximal size
int MAXSTACK = 1024;
stack = new int[MAXSTACK];
stack_size = 0;
```

Here the bottom of the stack is stored on position `0` of the array and the top of the stack in position `stack_size-1`. We can 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 `isEmpty` finish in *constant time*.

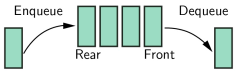
- Storing stacks as arrays has the advantage that we avoid calling $allocate\_memory$ all the time
  - (this takes time, even if it is done automatically for us, like in Java).
- On the other hand, we need to know the maximum size of the stack in advance.
- In practice, in Java, we normally use the library class $Deque < .... >$
  - which implements the Stack ADT as well as some others we will discuss and adjusts to grow as the stack increases in size.
  - We will see this later in this lecture.

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`.
- `isEmpty()` says whether the queue is empty



7

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.

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.

8

- 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 ADT

Here is a possible list of operations for a Queue ADT (many variations are possible)[2]
Constructors and Accessors:

- `EmptyQueue`: returns an empty Queue
- `push(element, queue)`: (also called `enqueue`) pushes an element onto the back of the given queue.
- `top(queue)`: (also called `front`) returns the value at the front of the queue without changing the queue[3]
- `pop(queue)`: (also called `dequeue`) returns the queue with the front element removed[3]
- `isEmpty(queue)`: reports whether the queue is empty

[2]Read chapters 1 and 2 of the module handouts
[3]Triggers error if the queue is empty

9

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.

Remember: dequeue from **front**, enqueue to **rear**

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?

10

Option 1 is better because:

How would enqueue and dequeue be implemented if we did (1) or (2)?

For (1) as long as we use 2 pointers in the head node, one to the first node of the linked list, one to the last node, it will take constant time to dequeue (remove from the start of the Linked List) and to enqueue (add a node at the end of the linked list).

For (2), again with 2 pointers in the head node, we can enqueue in constant time (insert a node at the start of the linked list). However, to dequeue, we now need the address of the penultimate node of the linked list in order to remove the last node, and to find that address, we will need to iterate through the whole linked list, costing linear time.

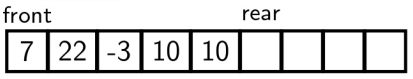
Thus we can ensure constant time enqueues and dequeues in case (1)

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

Case (2) gives us constant time enqueues but linear time dequeues.

Queue stored in an array (1st attempt)

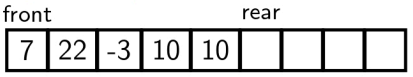
Whenever we `enqueue` (resp. `dequeue`) we increment `rear` (resp. `front`):



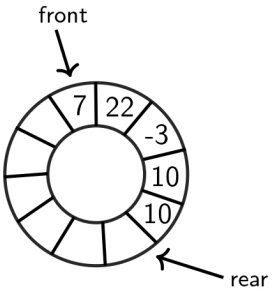
We eventually run out of space! To fix this, every time we `dequeue`, move everything to the left. It works but is slow!

Queue stored in an array (1st attempt)

Whenever we `enqueue` (resp. `dequeue`) we increment `rear` (resp. `front`):



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?

11

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 front position and size in variables `front` and `size`, respectively. Then, values stored in the queue are stored on the positions `front`, `front+1`, ..., `front+size-1`. We must maintain the invariant `front+size` $\leqslant$ `MAXQUEUE`.

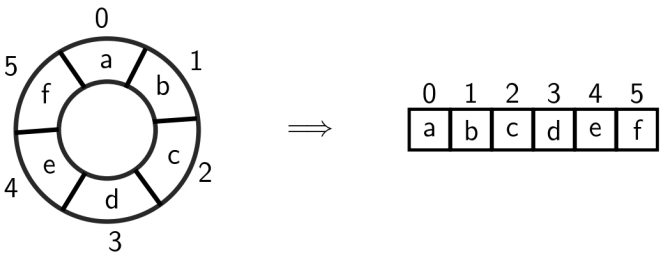
To `enqueue` we store the new value in position `front+size` and increment `size`. To `dequeue` we read out the value on position `front` and increment `front`.

However, this way, we eventually reach the end of the array and we can't continue enqueueing elements even if there is space in the array to do so, that is, the space in the array where old values were which have since been dequeued.

We can fix this by, everytime we dequeue a value, copying all the elements in the queue down to the start of the array so that `front` is back to 0. But this makes dequeue very slow. There is a better solution using a *Circular Array Queue Implementation*

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 one positions is computed as `(pos + 1) mod 6`; in general `(pos + 1) mod circle_length`.

We use `mod` sign to represent the modulo operator, e.g., 11 mod 6 = 5.

Mod and Div in Java

The math operators mod and div are defined so that $x \bmod y \in \{0, \ldots, y-1\}$, $x \operatorname{div} y \in \mathbb{Z}$, and

$$x = (x \operatorname{div} y) \times y + (x \bmod y).$$

Java's `%` and `/` interpret the operations on negative numbers incorrectly (Java, and C, allows negative results for mod). Instead, use `Math.floorDiv` and `Math.floorMod`.

For example:

- `127 % 10 == 7` and $127 \bmod 10 = 7$,
- `-18 % 10 == -8` and `-18 / 10 == -1`, but
- but $-18 \bmod 10 = 2$ and $-18 \operatorname{div} 10 = -2$.

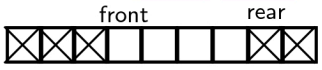
13

Circular queue: Array implementation

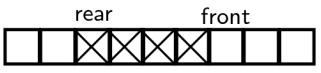
```
1  // Initialize an empty queue:
2  queue = new int[MAXQUEUE];
3  front = 0;
4  size = 0;        // rear = (front + size) mod MAXQUEUE
```

We store values in the queue in between positions marked by `rear` and `front`, i.e.,

- if `front` + `size` < `MAXQUEUE` then `rear` ⩾ `front` and the queue consists of the entries on positions `front`, `front` + 1, ..., `front` + `size` − 1:



- if `front` + `size` ≥ `MAXQUEUE` then `rear` ⩽ `front` and the queue consists of the entries on positions `front`, `front` + 1, ..., `MAXQUEUE` − 1 and 0, 1, ..., `rear`:



14

Digression: Invariants

An invariant is a condition on code. It must always be true during the execution of some section of code, e.g., a loop, or during the execution of a method, or, if it applies to a class, then it can be a condition that must be met by all objects of the class on every entry to and exit from the methods of the class even if it can be temporarily false during the execution of those methods.

Invariants are important because they specify conditions that must be met and maintained in parts of the code. So not only do they communicate information to the reader of the code, but they are tools that can be used both to identify and debug errors in the code (e.g. print an error message if this invariant is broken, stop in the debugger at any point when this invariant becomes false, etc.), and can be used to mathematically prove that the program is correct.

15

Circular queue: Array implementation

The invariants maintained in this implementation are:

- $0 \leqslant$ `front` $<$ `MAXQUEUE`
- $0 \leqslant$ `size` $\leqslant$ `MAXQUEUE`

Note that rather than using a *rear* index variable, we will always calculate it when we need it from *front*, *size* and *MAXQUEUE*. Thus

```
rear = (first + size) % MAXQUEUE
```

The queue is full if `size == MAXQUEUE` and empty when `size == 0`

To enqueue, we first check that the queue is not full, put the new value at index position `rear`, and increment `size` by adding 1 to it.

To dequeue, we first check that the queue is not empty, get the value at index position `front`, increment `front` by calculating `front = (front + 1) % MAXQUEUE` and decrement `size` by subtracting 1 from it.

16

Circular queue: Array implementation

```
// Initialize empty queue:
queue = new int[MAXQUEUE];
front  = 0;
size = 0;

boolean isEmpty () {
  return size == 0;
}

boolean isFull () {
  return size == MAXQUEUE;
}
```

```
void enqueue (int val) {
  if (size == MAXQUEUE) {
    throw new QueueFullException;
  }
// rear = (front + size) mod MAXQUEUE
// % is okay here since front + size >= 0
  queue[(front + size) % MAXQUEUE] = val;
  size++;
}

int dequeue () {
  if (size == 0) {
    throw new QueueEmptyException;
  }
  int val = queue[front];
  front = (front+1) % MAXQUEUE;
  size--;
  return val;
}
```

17

Double Ended Queues: Deques

While the Java library does have a `Stack<...>` class, it is an old design that has been kept for backwards compatibilty purposes and should not normally be used. For both `Stack` and `Queue` classes, you should use the `Deque<...>` class, which implements a double-ended queue data type. This has implementations `ArrayDeque<...>` and `LinkedList<...>` and supports inserting at and removing from both ends.

Actually, the `Deque<...>` class is really a Java Interface, rather than a full Class. This will be covered in your Java programming module but the distinction is not important for the purposes of this module.

18

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.

19