Assignment 2

- Make sure that you understand the tasks of A2, know what to do, ask questions if in doubt.
- Do assignment 2, further questions, and/or
- Review complexity, recurrences, and other parts.

A2.P3 and A2.P2: Questions?

Pseudocode in A2.P2:

For simplicity you can suppose that a tree node include

- array `child[0..3]` of children
- array `val[1..3]` of keys

(but if you do so, you should clearly state the supposition).

Pseudocode should be clear and precise (ie. programable), but also be concise (ie. with no/minimal redundancy).

Other questions?

sample pseudocode: insertTree of the skeleton

Notes

```
# suppose tree node is {nVal,val[1..3],child[0..3]} where
#     - nVal is the number of keys in the node
#     - val[1..3] is array of the node's keys
#     - child[0..3] is array of the node's children
# also suppose that "-" denotes an unused slot in the array val[]
```

`insert` = skeleton's insertTree

```
# insert val into t, where t is the root node of a 234 tree
function insert(t, val)
  if t = nil then
    return {1, {val,-,-},{nil, nil, nil, nil}}
  if t.nVal = 3 then
    # root is full, split and build new root
    left  := {1, {t.val[1],-,-}, {t.child[0], t.child[1], nil, nil}}
    right := {1, {t.val[3],-,-}, {t.child[2], t.child[3], nil, nil}}
    t := {1, {t.val[2],-,-}, {left, right, nil, nil}}
  #now root has at most 2 keys, do recursive insertion
  return insertRec(t, val)
```

PS: I might have made mistakes in this and the next slide, please advise if you catch any.

insertRec = **insertTreeRecursive** + **insertIntoNode** + **splitAndInsert**

```
# insert val to tree t, where node t has at most 2 keys
# OR t has 3 keys but will further insert into a child with a single key
function insertRec(t, val)
  if t.child[0]=nil
    # t is a leaf with at most 2 keys, just insert "val"
    increase t.nVal by 1
    t.val[t.nVal] := val
    sort array t.val[1 .. t.nVal] in increasing order
    return t
  for c:= 0 to 2 do   # here, node can only have at most 3 children
    if c=2 or val < t.val[c+1] then
      # val belongs to child[c]
      x := t.child[c]
      if x.nVal=3 then
        # x is full, fisrt split it and promote its middle key to t
        left  := {1,{x.val[1],-,-}, {x.child[0], x.child[1], nil, nil}}
        right := {1,{x.val[3],-,-}, {x.child[2], x.child[3], nil, nil}}
        t.child[c] := right
        insert left into position c of array t.child[0..2]
        insert x.val[2] into position c+1 of array t.val[1..2]
        increase t.nVal by 1
        # then re-insert val to t
        # it will further insert to left or right, each has a single key
        return insertRec(t, val)
      else
        # insert into child c
        t.child[c] := insertRec(t.child[c], val)
        return t
```

A2.P1: more on C

C bitwise operators: AND `&`, OR `|`, XOR `^`, … operate on integers or unsigned integers at binary level, ie. by processing each bit of the binary representations.

With unsigned integers there are less confusion. Example of unsigned int datatypes in C:

- `uint64_t` : 8-byte non-negative int, convenient for very big numbers (up to $2^{64}-1$)
- `uint8_t` : 1-byte non-negative int. A text of `N` characters can be declared as

```
uint8_t text[N];
```

and represented as a pair `(uint8_t *text, unit64_t N)` [no \0 at the end].

Exclusive OR ^ and some important properties:

```
a ^ a  =  0
a ^ b  =  b ^ a
(a ^ b) ^ c  =  a ^ (b ^ c)
```

So:

```
c  =  s^t   →   t  =  s ^ c
```

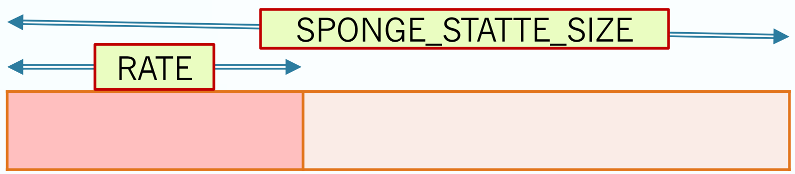
a	b	a^b	example	
0	0	0	10001101	141
0	1	1	10001010	138
1	0	1	--------------	-----
1	1	0	00000111	7

Other C facilities: not crucial, but it would be better if you `google` or use `man` to know about `memset` and `memcpy`.

A2.P1: Sponge

A sponge is defined by its state, which is array of SPONGE_STATTE_SIZE bytes:

uint8_t state[SPONGE_STATE_SIZE];



`init`: zeroing the whole `state`

`read`: copying the first `num` bytes of state into a buffer `uint8_t *dest`

`write`: using the first `num` bytes of the text `uint8_t *src` and

- copying it into `state[0..num-1]` if the flag `bw_xor` is `false`
- `XOR` it into `state[0..num-1]` if the flag `bw_xor` is `true`

`demarcate`: XOR a byte `uint8_t delimiter` into `state[i]`

`permute`: using function *`permutation_384`* to replace state with a sequence of `SPONGE_STATTE_SIZE` bytes in a deterministic and reversible way

A2.P1: Hashing

// Hashes an input text `T` of `len` bytes to produce the hash value `H` of `H_len` bytes.

```
void hash(uint8_t *H, uint64_t H_len, uint8_t const *T, uint64_t len);
```

Step 0: start with a zeroed sponge `s`

Step 1 – Absorb `T` into `s`:

divide `T` into blocks T_0, T_1, T_2, … T_k of `RATE` bytes (T_k has $0 \le r \le RATE-1$ bytes)

for each block T_i in that order:

absorb T_i into `s` using `sponge_write` with `bw_xor= true`

```
//read block into ciphetext Ci
  // Ci=  Ti ^ s  ➔ Ti= Ci ^ s
```

permute `s` (note: special treatment for the last block)

Step 2 – Demarcation: do 2 specific demarcations

Step 3 – Squeezing: getting value for H block-by-block

while (`tag` is not full)

read `tag` bytes from `s` and append it to `H`

permute `s`

A2.P1: MAC

// Creates authentication tag of size tag_len bytes from key of size CRYPTO_KEY_SIZE bytes and an input text T msg len bytes.

```
void mac(uint8_t *tag, uint64_t tag_len, uint8_t const *key,
            uint8_t const *T, uint64_t len);
void hash(uint8_t *H, uint64_t H_len, uint8_t const *T, uint64_t len);
```

Step 0:

start with a zeroed sponge `s`

absorb key (of `CRYPTO_KEY_SIZE` bytes) into `s`

Step 1,2,3:

same as teps 1,2,3 of hashing,

but in Step 3, squeeze into `tag` (of `tag_len` bytes) instead of `H`

A2.P1: encrypt is similar to MAC

For simplicity you can suppose

```
void mac(                      tag, tag_len, key, msg,        msg_len);
void auth_encr(ciphertext, tag, tag_len, key, plaintext, text_len);
```

the only difference is in Step 1 (absorbing), when we build the encrypted `ciphertext` which also has length of `text_len` bytes

A2.P1: decrypt should mimic the encrypt

and you should make sure that the sponge is the same in both encryption and decryption before each permutation. Read the XOR properties again and figure out how to get back the original plain_text.

```
void auth_encr(ciphertext, tag, tag_len, key, plaintext, text_len);
int auth_decr (plaintext, key, ciphertext, text_len, tag, tag_len);
```

?