TDD in C

Illustrated by the Recently-Used-List kata

Olve Maudal

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

- A recently-used-list is initially empty. As you add unique numbers, the list grows, until capacity is reached.

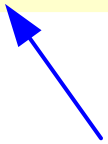
- Numbers in the list can be looked up by index. The zeroth element is the most recently used item.

- Numbers in the list are unique. If you add a number that already exists, the number is moved to the zeroth element in the list.

- When adding a unique number to a full list, the oldest number is dropped to make room for the new entry.

Source: cyber-dojo.org

README.txt <empty> <empty> <empty> <empty>

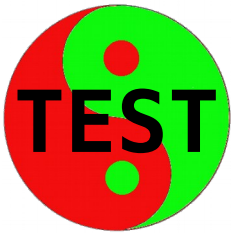


README.txt | rul_tests.c | <empty> | <empty> | <empty>

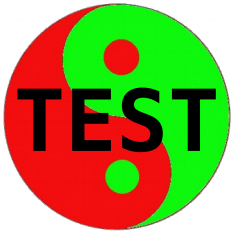
```c
#include <stdio.h>

int main(void)
{
    printf("All tests passed\n");
}
```

```c
#include <stdio.h>

int main(void)
{
    printf("All tests passed\n");
}
```

```c
#include <stdio.h>

int main(void)
{
    printf("All tests passed\n");
}
```



```c
#include <stdio.h>

int main(void)
{
    printf("All tests passed\n");
}
```




```
cc rul_tests.c && ./a.out
All tests passed
```

```c
#include <stdio.h>

int main(void)
{
    printf("All tests passed\n");
}
```

```c
#include <stdio.h>

int main(void)
{
    printf("All tests passed\n");
}
```

```
#include <stdio.h>

int main(void)
{

    printf("All tests passed\n");
}
```

```c
#include <stdio.h>

int main(void)
{
    assert(3 == 4);
    printf("All tests passed\n");
}
```

```
#include <stdio.h>

int main(void)
{
    assert(3 == 4);
    printf("All tests passed\n");
}
```

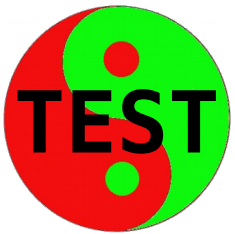
```c
#include <stdio.h>

int main(void)
{
    assert(3 == 4);
    printf("All tests passed\n");
}
```

```
#include <stdio.h>

int main(void)
{
    assert(3 == 4);
    printf("All tests passed\n");
}
```

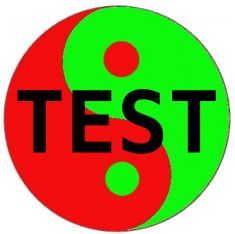
```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 4);
    printf("All tests passed\n");
}
```

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 4);
    printf("All tests passed\n");
}
```

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 4);
    printf("All tests passed\n");
}
```

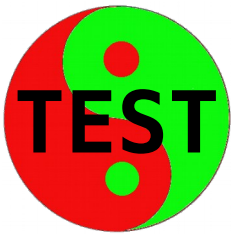


```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 4);
    printf("All tests passed\n");
}
```




```
cc rul_tests.c && ./a.out
a.out: rul_tests.c:6: main: Assertion `3 == 4' failed.
```

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 4);
    printf("All tests passed\n");
}
```




```
cc rul_tests.c && ./a.out
a.out: rul_tests.c:6: main: Assertion `3 == 4' failed.
```

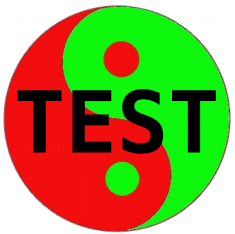
README.txt | rul_tests.c | <empty> | <empty> | <empty>

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 4);
    printf("All tests passed\n");
}
```

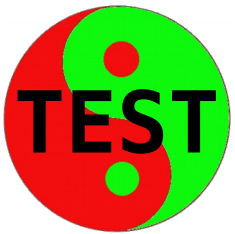
```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 3);
    printf("All tests passed\n");
}
```

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 3);
    printf("All tests passed\n");
}
```

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 3);
    printf("All tests passed\n");
}
```



```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 3);
    printf("All tests passed\n");
}
```




```
cc rul_tests.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | <empty> | <empty> | <empty>

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 3);
    printf("All tests passed\n");
}
```

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 3);
    printf("All tests passed\n");
}
```

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 3);
    printf("All tests passed\n");
}
```

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    assert(3 == 3);
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

int main(void)
{
    
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

int main(void)
{

    printf("All tests passed\n");
}
```

```c
#include <assert.h>
#include <stdio.h>

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

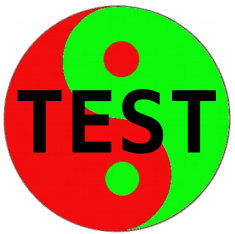
```
#include <assert.h>
#include <stdio.h>
```

```
int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

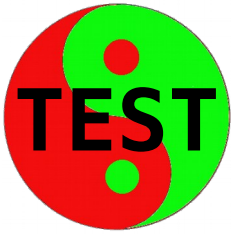
```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 4);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 4);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 4);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```



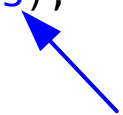
```c
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 4);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```




```
cc rul_tests.c && ./a.out
a.out: rul_tests.c:6: test_list_is_initially_empty: Assertion `3 == 4' failed.
```

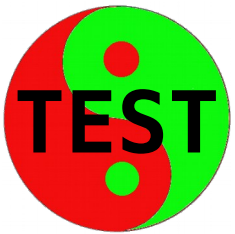
```c
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 4);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 3);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

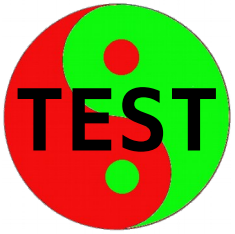


```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 3);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 3);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```



```c
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 3);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```




```
cc rul_tests.c && ./a.out
All tests passed
```

```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 3);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 3);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(3 == 3);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(        );
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(                    );
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

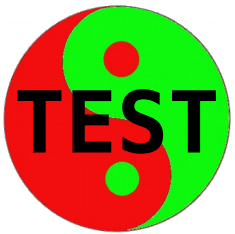
```c
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

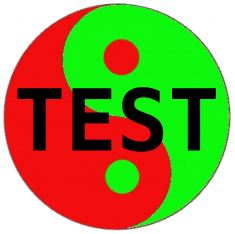
```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```



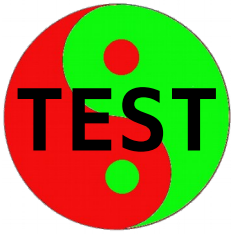
```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```




```
cc rul_tests.c && ./a.out
rul_tests.c:1:17: fatal error: rul.h: No such file or directory
```

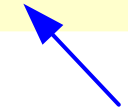
README.txt | rul_tests.c | <empty> | <empty> | <empty>

```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```




```
cc rul_tests.c && ./a.out
rul_tests.c:1:17: fatal error: rul.h: No such file or directory
```

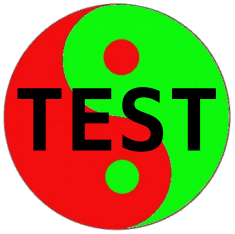
README.txt | rul_tests.c | <empty> | <empty> | <empty>



```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```

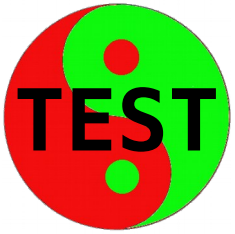
```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```



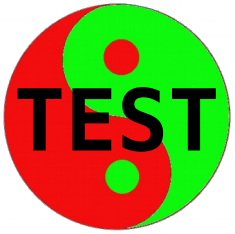
README.txt | rul_tests.c | rul.h | <empty> | <empty>

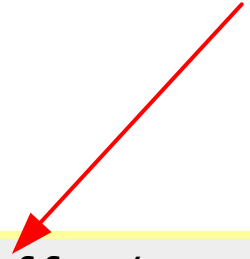
```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```




```
cc rul_tests.c && ./a.out
rul_tests.c:(.text+0xa): undefined reference to `rul_size'
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```

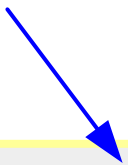




```
cc rul_tests.c && ./a.out
rul_tests.c:(.text+0xa): undefined reference to `rul_size'
```

README.txt | rul_tests.c | rul.h | <empty> | <empty>

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```



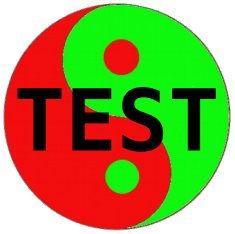
```
cc rul_tests.c && ./a.out
rul_tests.c:(.text+0xa): undefined reference to `rul_size'
```

README.txt | rul_tests.c | rul.h | <empty> | <empty>

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```

```
cc rul_tests.c rul.c && ./a.out
```

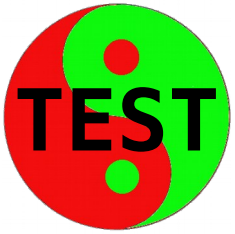
```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```




```
cc rul_tests.c rul.c && ./a.out
```

README.txt | rul_tests.c | rul.h | <empty> | <empty>

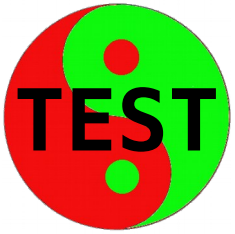
```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```




```
cc rul_tests.c rul.c && ./a.out
cc: error: rul.c: No such file or directory
```

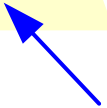
README.txt | rul_tests.c | rul.h | <empty> | <empty>

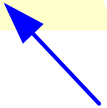
```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```




```
cc rul_tests.c rul.c && ./a.out
cc: error: rul.c: No such file or directory
```

README.txt | rul_tests.c | rul.h | <empty> | <empty>





README.txt | rul_tests.c | rul.h | rul.c | <empty>

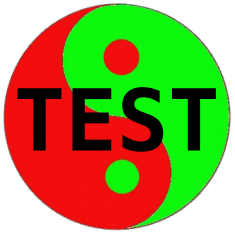
```c
#include "rul.h"

size_t rul_size(void)
{
    return   ;
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

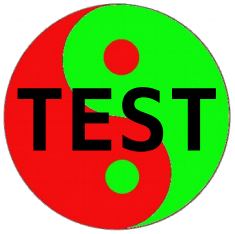
```c
#include "rul.h"

size_t rul_size(void)
{
    return ;
}
```

```c
#include "rul.h"

size_t rul_size(void)
{
    return 42;
}
```

```c
#include "rul.h"

size_t rul_size(void)
{
    return 42;
}
```



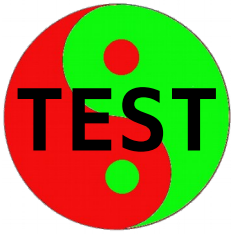
```c
#include "rul.h"

size_t rul_size(void)
{
    return 42;
}
```




```
cc rul_tests.c rul.c && ./a.out
test_list_is_initially_empty: Assertion `rul_size() == 0' failed.
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

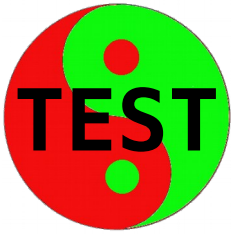
```c
#include "rul.h"

size_t rul_size(void)
{
    return 42;
}
```



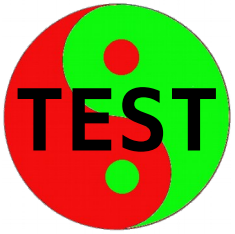

```
cc rul_tests.c rul.c && ./a.out
test_list_is_initially_empty: Assertion `rul_size() == 0' failed.
```

```c
#include "rul.h"

size_t rul_size(void)
{
    return 0;
}
```

```c
#include "rul.h"

size_t rul_size(void)
{
    return 0;
}
```



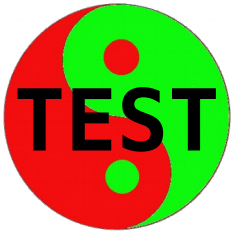
```c
#include "rul.h"

size_t rul_size(void)
{
    return 0;
}
```




```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

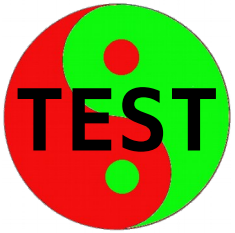
```
#include "rul.h"

size_t rul_size(void)
{
    return 0;
}
```




```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

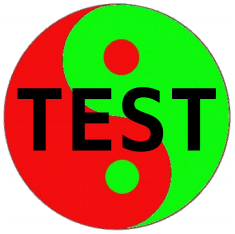
```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```



README.txt | rul_tests.c | rul.h | rul.c | <empty>

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```




```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size() == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul * rul = NULL;
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul * rul = NULL;
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul * rul = NULL;
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

size_t rul_size(void);

#endif
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {};

size_t rul_size(const struct rul * rul);

#endif
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

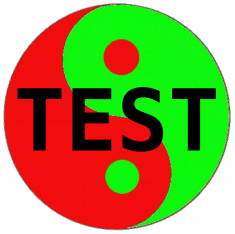
```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {};

size_t rul_size(const struct rul * rul);

#endif
```

```c
#include "rul.h"

size_t rul_size(void)
{
    return 0;
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

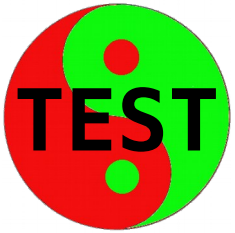
```c
#include "rul.h"

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```

```c
#include "rul.h"

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```



README.txt | rul_tests.c | rul.h | rul.c | <empty>

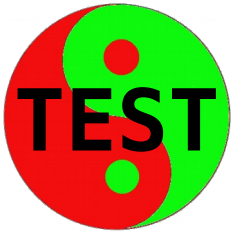
```c
#include "rul.h"

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```




```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```c
#include "rul.h"

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```

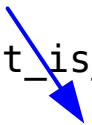



```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul * rul = NULL;
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul * rul = NULL;
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul * rul = NULL;
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```



```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

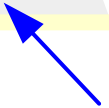
README.txt | rul_tests.c | rul.h | rul.c | <empty>

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {};

size_t rul_size(const struct rul * rul);

#endif
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);

#endif
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>



```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);

#endif
```

```
#include "rul.h"

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```

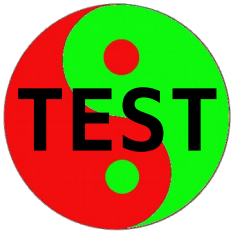
```
#include "rul.h"




size_t rul_size(const struct rul * rul)
{
    return 0;
}
```

```c
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return NULL;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return NULL;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```

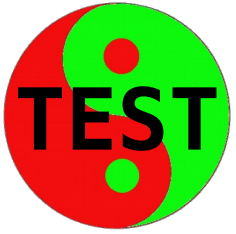
README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return NULL;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```



README.txt | rul_tests.c | rul.h | rul.c | <empty>

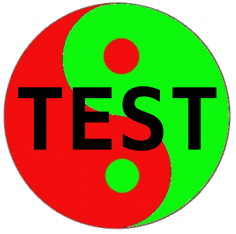
```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return NULL;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```




```
cc rul_tests.c rul.c && ./a.out
test_list_is_initially_empty: Assertion `rul == &myrul' failed.
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

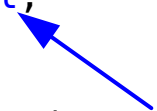
```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return NULL;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```




```
cc rul_tests.c rul.c && ./a.out
test_list_is_initially_empty: Assertion `rul == &myrul' failed.
```

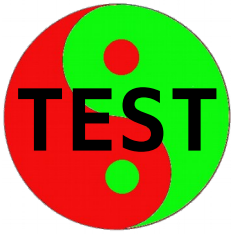
README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```



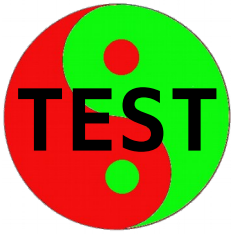
```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```



README.txt | rul_tests.c | rul.h | rul.c | <empty>

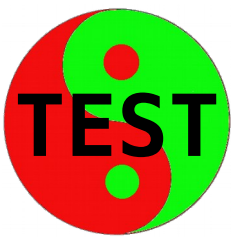
```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```




```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

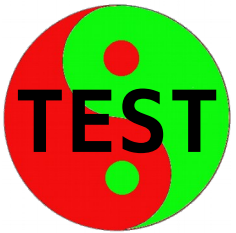
```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```



README.txt | rul_tests.c | rul.h | rul.c | <empty>

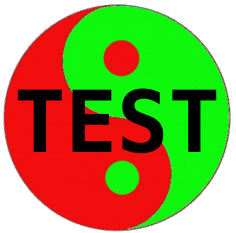
```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```




```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```




```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

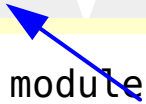
Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.



Develop a recently-used-list module for holding a limited set of unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer than 15 digits. You may also assume that the capacity of the list is 10 items. (Imagine that this code is to be used to show a list of the most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers, the list grows, until capacity is reached.

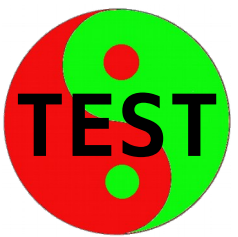
o) Numbers in the list can be looked up by index. The zeroth element is the most recently used item.

o) Numbers in the list are unique. If you add a number that already exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is dropped to make room for the new entry.

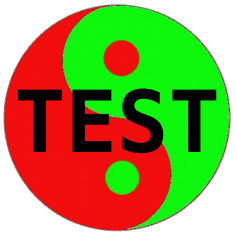
```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```



README.txt | rul_tests.c | rul.h | rul.c | <empty>

```c
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```




```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
#include "rul.h"

#include <assert.h>
#include <stdio.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

...

```
int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

...

```
int main(void)
{
    test_list_is_initially_empty();
    printf("All tests passed\n");
}
```

...

```
int main(void)
{
    test_list_is_initially_empty();

    printf("All tests passed\n");
}
```

...

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    assert(3 == 4);
}









int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    assert(3 == 4);
}









int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

```
cc rul_tests.c rul.c && ./a.out
test_size_of_list_increases_as_we_add_unique_items: Assertion `3 == 4' failed.
```

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    assert(3 == 4);
}









int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    assert(3 == 4);
}









int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(rul_size(rul) == 1);
}




int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(rul_size(rul) == 1);
}
```

```
cc rul_tests.c rul.c && ./a.out
rul_tests.c:(.text+0x92): undefined reference to `rul_add'
```

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(rul_size(rul) == 1);
}
```

```
cc rul_tests.c rul.c && ./a.out
rul_tests.c:(.text+0x92): undefined reference to `rul_add'
```

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);

#endif
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);

#endif
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);

#endif
```

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}
```

```c
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}

void rul_add(struct rul * rul, const char * num)
{
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}

void rul_add(struct rul * rul, const char * num)
{
}
```

```
cc rul_tests.c rul.c && ./a.out
a.out: rul_tests.c:19: test_size_of_list_increases_as_we_add_unique_items:
Assertion `rul_size(rul) == 1' failed.
```

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return 0;
}

void rul_add(struct rul * rul, const char * num)
{
}
```

```c
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
}
```

```c
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```c
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}
```

```c
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);

#endif
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {

};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);

#endif
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {
    size_t n_numbers;
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);

#endif
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {
    size_t n_numbers;
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);

#endif
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {
    size_t n_numbers;
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);

#endif
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>


```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {
    size_t n_numbers;
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);

#endif
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(rul_size(rul) == 1);
}









int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(rul_size(rul) == 1);
    rul_add(rul, "1001");
    assert(rul_size(rul) == 2);
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    rul_add(rul, "1004");
    assert(rul_size(rul) == 5);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(rul_size(rul) == 1);
    rul_add(rul, "1001");
    assert(rul_size(rul) == 2);
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    rul_add(rul, "1004");
    assert(rul_size(rul) == 5);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(rul_size(rul) == 1);
    rul_add(rul, "1001");
    assert(rul_size(rul) == 2);
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    rul_add(rul, "1004");
    assert(rul_size(rul) == 5);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
static void test_size_of_list_increases_as_we_add_unique_items(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(rul_size(rul) == 1);
    rul_add(rul, "1001");
    assert(rul_size(rul) == 2);
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    rul_add(rul, "1004");
    assert(rul_size(rul) == 5);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

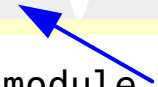
Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.



Develop a recently-used-list module for holding a limited set of unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer than 15 digits. You may also assume that the capacity of the list is 10 items. (Imagine that this code is to be used to show a list of the most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers, the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element is the most recently used item.

o) Numbers in the list are unique. If you add a number that already exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is dropped to make room for the new entry.

...

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    printf("All tests passed\n");
}
```

...

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
}









int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
}









int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
}









int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {
    size_t n_numbers;
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);

#endif
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {
    size_t n_numbers;
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);
const char * rul_get(const struct rul * rul, size_t index);

#endif
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {
    size_t n_numbers;
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);
const char * rul_get(const struct rul * rul, size_t index);

#endif
```

```c
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}
```

```c
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "";
}
```

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "";
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

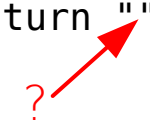
```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "";
}
```

```
cc rul_tests.c rul.c && ./a.out
rul_tests.c:34: test_indexing_into_the_list:
Assertion `strcmp(rul_get(rul, 0), "1000") == 0' failed.
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "";
}
```

```
cc rul_tests.c rul.c && ./a.out
rul_tests.c:34: test_indexing_into_the_list:
Assertion `strcmp(rul_get(rul, 0), "1000") == 0' failed.
```



README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "1000";
}
```

```
cc rul_tests.c rul.c && ./a.out
rul_tests.c:34: test_indexing_into_the_list:
Assertion `strcmp(rul_get(rul, 0), "1000") == 0' failed.
```

```c
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "1000";
}
```

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "1000";
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "1000";
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#include "rul.h"

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "1000";
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
}









int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

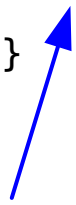
```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
}



int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
}




int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
cc rul_tests.c rul.c && ./a.out
rul_tests.c:36: test_indexing_into_the_list:
Assertion `strcmp(rul_get(rul, 0), "1001") == 0' failed.
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
}


int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```



```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    // rul_add(rul, "1001");
    // assert(strcmp(rul_get(rul, 0), "1001") == 0);
}



int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

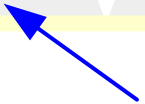
```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    // rul_add(rul, "1001");
    // assert(strcmp(rul_get(rul, 0), "1001") == 0);
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    // rul_add(rul, "1001");
    // assert(strcmp(rul_get(rul, 0), "1001") == 0);
}




int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>



```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    // rul_add(rul, "1001");
    // assert(strcmp(rul_get(rul, 0), "1001") == 0);
}



int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

struct rul {
    size_t n_numbers;
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);
const char * rul_get(const struct rul * rul, size_t index);

#endif
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

#define MAX_NUMLEN 15
#define MAX_NUMBERS 10

struct rul {
    size_t n_numbers;
    char numbers[MAX_NUMBERS][MAX_NUMLEN+1];
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);
const char * rul_get(const struct rul * rul, size_t index);

#endif
```

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

#define MAX_NUMLEN 15
#define MAX_NUMBERS 10

struct rul {
    size_t n_numbers;
    char numbers[MAX_NUMBERS][MAX_NUMLEN+1];
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);
const char * rul_get(const struct rul * rul, size_t index);

#endif
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

#define MAX_NUMLEN 15
#define MAX_NUMBERS 10

struct rul {
    size_t n_numbers;
    char numbers[MAX_NUMBERS][MAX_NUMLEN+1];
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);
const char * rul_get(const struct rul * rul, size_t index);

#endif
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "1000";
}
```

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    strcpy(rul->numbers[0], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "1000";
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    strcpy(rul->numbers[0], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "1000";
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    strcpy(rul->numbers[0], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "1000";
}
```

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    strcpy(rul->numbers[0], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return "1000";
}
```

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    strcpy(rul->numbers[0], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return rul->numbers[0];
}
```

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    strcpy(rul->numbers[0], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return rul->numbers[0];
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    // rul_add(rul, "1001");
    // assert(strcmp(rul_get(rul, 0), "1001") == 0);
}



int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    // rul_add(rul, "1001");
    // assert(strcmp(rul_get(rul, 0), "1001") == 0);
}




int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    // rul_add(rul, "1001");
    // assert(strcmp(rul_get(rul, 0), "1001") == 0);
}



int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
}



int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
}




int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

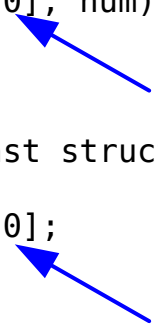
README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
    assert(strcmp(rul_get(rul, 1), "1000") == 0);
}


int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
cc rul_tests.c rul.c && ./a.out
rul_tests.c: test_indexing_into_the_list:
Assertion `strcmp(rul_get(rul, 1), "1000") == 0' failed.
```

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    strcpy(rul->numbers[0], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return rul->numbers[0];
}
```

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    strcpy(rul->numbers[0], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    return rul->numbers[0];
}
```



```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        return;
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    if (index + 1 > rul->n_numbers)
        return "";
    return rul->numbers[rul->n_numbers - (index + 1)];
}
```

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        return;
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    if (index + 1 > rul->n_numbers)
        return "";
    return rul->numbers[rul->n_numbers - (index + 1)];
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        return;
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    if (index + 1 > rul->n_numbers)
        return "";
    return rul->numbers[rul->n_numbers - (index + 1)];
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
    assert(strcmp(rul_get(rul, 1), "1000") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
    assert(strcmp(rul_get(rul, 1), "1000") == 0);
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    assert(strcmp(rul_get(rul, 0), "1003") == 0);
    assert(strcmp(rul_get(rul, 1), "1002") == 0);
    assert(strcmp(rul_get(rul, 2), "1001") == 0);
    assert(strcmp(rul_get(rul, 3), "1000") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
    assert(strcmp(rul_get(rul, 1), "1000") == 0);
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    assert(strcmp(rul_get(rul, 0), "1003") == 0);
    assert(strcmp(rul_get(rul, 1), "1002") == 0);
    assert(strcmp(rul_get(rul, 2), "1001") == 0);
    assert(strcmp(rul_get(rul, 3), "1000") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer than 15 digits. You may also assume that the capacity of the list is 10 items. (Imagine that this code is to be used to show a list of the most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers, the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element is the most recently used item.

o) Numbers in the list are unique. If you add a number that already exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is dropped to make room for the new entry.

...

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    printf("All tests passed\n");
}
```

...

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    test_adding_unique_items_to_full_list();
    printf("All tests passed\n");
}
```

```
...
static void test_adding_unique_items_to_full_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    char num[MAX_NUMLEN+1];
    for (int i = 0; i < MAX_NUMBERS; i++) {
        sprintf(num, "10%02d", i);
        rul_add(rul, num);
    }
    assert(rul_size(rul) == MAX_NUMBERS);
    assert(strcmp(rul_get(rul, MAX_NUMBERS-1), "1000") == 0);
    rul_add(rul, "1234");
    assert(rul_size(rul) == MAX_NUMBERS);
    assert(strcmp(rul_get(rul, 0), "1234") == 0);
    assert(strcmp(rul_get(rul, MAX_NUMBERS-1), "1001") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    test_adding_unique_items_to_full_list();
    printf("All tests passed\n");
}
```

```
...
static void test_adding_unique_items_to_full_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    char num[MAX_NUMLEN+1];
    for (int i = 0; i < MAX_NUMBERS; i++) {
        sprintf(num, "10%02d", i);
        rul_add(rul, num);
    }
    assert(rul_size(rul) == MAX_NUMBERS);
    assert(strcmp(rul_get(rul, MAX_NUMBERS-1), "1000") == 0);
    rul_add(rul, "1234");
    assert(rul_size(rul) == MAX_NUMBERS);
    assert(strcmp(rul_get(rul, 0), "1234") == 0);
    assert(strcmp(rul_get(rul, MAX_NUMBERS-1), "1001") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    test_adding_unique_items_to_full_list();
    printf("All tests passed\n");
}
```

```
cc rul_tests.c rul.c && ./a.out
test_adding_unique_items_to_full_list:
Assertion `strcmp(rul_get(rul, 0), "1234") == 0' failed.
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
static void test_adding_unique_items_to_full_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    char num[MAX_NUMLEN+1];
    for (int i = 0; i < MAX_NUMBERS; i++) {
        sprintf(num, "10%02d", i);
        rul_add(rul, num);
    }
    assert(rul_size(rul) == MAX_NUMBERS);
    assert(strcmp(rul_get(rul, MAX_NUMBERS-1), "1000") == 0);
    rul_add(rul, "1234");
    assert(rul_size(rul) == MAX_NUMBERS);
    assert(strcmp(rul_get(rul, 0), "1234") == 0);
    assert(strcmp(rul_get(rul, MAX_NUMBERS-1), "1001") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    test_adding_unique_items_to_full_list();
    printf("All tests passed\n");
}
```

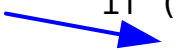
```
cc rul_tests.c rul.c && ./a.out
test_adding_unique_items_to_full_list:
Assertion `strcmp(rul_get(rul, 0), "1234") == 0' failed.
```

...

```
void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        return;
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    if (index + 1 > rul->n_numbers)
        return "";
    return rul->numbers[rul->n_numbers - (index + 1)];
}
```

...

```
void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        return;
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    if (index + 1 > rul->n_numbers)
        return "";
    return rul->numbers[rul->n_numbers - (index + 1)];
}
```



...

```
void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    if (index + 1 > rul->n_numbers)
        return "";
    return rul->numbers[rul->n_numbers - (index + 1)];
}
```

```
...

static void remove_oldest_number(struct rul * rul)
{
    for (size_t i = 0; i < rul->n_numbers - 1; i++)
        strcpy(rul->numbers[i], rul->numbers[i+1]);
    rul->n_numbers--;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    if (index + 1 > rul->n_numbers)
        return "";
    return rul->numbers[rul->n_numbers - (index + 1)];
}
```

```
...

static void remove_oldest_number(struct rul * rul)
{
    for (size_t i = 0; i < rul->n_numbers - 1; i++)
        strcpy(rul->numbers[i], rul->numbers[i+1]);
    rul->n_numbers--;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    if (index + 1 > rul->n_numbers)
        return "";
    return rul->numbers[rul->n_numbers - (index + 1)];
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...

static void remove_oldest_number(struct rul * rul)
{
    for (size_t i = 0; i < rul->n_numbers - 1; i++)
        strcpy(rul->numbers[i], rul->numbers[i+1]);
    rul->n_numbers--;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    if (index + 1 > rul->n_numbers)
        return "";
    return rul->numbers[rul->n_numbers - (index + 1)];
}
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

Develop a recently-used-list module for holding a limited set of
unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer
than 15 digits. You may also assume that the capacity of the list is
10 items. (Imagine that this code is to be used to show a list of the
most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers,
   the list grows, until capacity is reached.

o) Numbers in the list can be looked up by index. The zeroth element
   is the most recently used item.

o) Numbers in the list are unique. If you add a number that already
   exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is
   dropped to make room for the new entry.

Develop a recently-used-list module for holding a limited set of unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer than 15 digits. You may also assume that the capacity of the list is 10 items. (Imagine that this code is to be used to show a list of the most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers, the list grows, until capacity is reached. 

o) Numbers in the list can be looked up by index. The zeroth element is the most recently used item. 

o) Numbers in the list are unique. If you add a number that already exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is dropped to make room for the new entry. 

Develop a recently-used-list module for holding a limited set of unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer than 15 digits. You may also assume that the capacity of the list is 10 items. (Imagine that this code is to be used to show a list of the most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers, the list grows, until capacity is reached. 

o) Numbers in the list can be looked up by index. The zeroth element is the most recently used item. 

o) Numbers in the list are unique. If you add a number that already exists, the number is moved to the zeroth element in the list.

o) When adding a unique number to a full list, the oldest number is dropped to make room for the new entry. 

...

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    test_adding_unique_items_to_full_list();
    printf("All tests passed\n");
}
```

...

```
int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    test_adding_unique_items_to_full_list();
    test_adding_existing_numbers_to_list();
    printf("All tests passed\n");
}
```

```
...
static void test_adding_existing_numbers_to_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    rul_add(rul, "1001");
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    assert(rul_size(rul) == 4);
    rul_add(rul, "1002");
    assert(rul_size(rul) == 4);
    assert(strcmp(rul_get(rul, 0), "1002") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    test_adding_unique_items_to_full_list();
    test_adding_existing_numbers_to_list();
    printf("All tests passed\n");
}
```

```
...
static void test_adding_existing_numbers_to_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    rul_add(rul, "1001");
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    assert(rul_size(rul) == 4);
    rul_add(rul, "1002");
    assert(rul_size(rul) == 4);
    assert(strcmp(rul_get(rul, 0), "1002") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    test_adding_unique_items_to_full_list();
    test_adding_existing_numbers_to_list();
    printf("All tests passed\n");
}
```

```
cc rul_tests.c rul.c && ./a.out
rul_tests.c:73: test_adding_existing_numbers_to_list:
Assertion `rul_size(rul) == 4' failed.
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
static void test_adding_existing_numbers_to_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    rul_add(rul, "1001");
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    assert(rul_size(rul) == 4);
    rul_add(rul, "1002");
    assert(rul_size(rul) == 4);
    assert(strcmp(rul_get(rul, 0), "1002") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_items();
    test_indexing_into_the_list();
    test_adding_unique_items_to_full_list();
    test_adding_existing_numbers_to_list();
    printf("All tests passed\n");
}
```

```
cc rul_tests.c rul.c && ./a.out
rul_tests.c:73: test_adding_existing_numbers_to_list:
Assertion `rul_size(rul) == 4' failed.
```

...

```
void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}
```

...

...

```
void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}
```

...

...

```
void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    size_t i = find_number(rul, num);
    if (i != rul->n_numbers)
        remove_number_at_index(rul, i);
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}
```

...

```
...
static size_t find_number(struct rul * rul, const char * num)
{
    for (size_t i = 0; i < rul->n_numbers; i++)
        if (strcmp(rul->numbers[i], num) == 0)
            return i;
    return rul->n_numbers;
}
```

```
void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    size_t i = find_number(rul, num);
    if (i != rul->n_numbers)
        remove_number_at_index(rul, i);
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}
...
```

```
static size_t find_number(struct rul * rul, const char * num)
{
    for (size_t i = 0; i < rul->n_numbers; i++)
        if (strcmp(rul->numbers[i], num) == 0)
            return i;
    return rul->n_numbers;
}

static void remove_number_at_index(struct rul * rul, size_t index)
{
    for (size_t i = index; i < rul->n_numbers - 1; i++)
        strcpy(rul->numbers[i], rul->numbers[i+1]);
    rul->n_numbers--;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    size_t i = find_number(rul, num);
    if (i != rul->n_numbers)
        remove_number_at_index(rul, i);
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}
...
```

```
...
static size_t find_number(struct rul * rul, const char * num)
{
    for (size_t i = 0; i < rul->n_numbers; i++)
        if (strcmp(rul->numbers[i], num) == 0)
            return i;
    return rul->n_numbers;
}

static void remove_number_at_index(struct rul * rul, size_t index)
{
    for (size_t i = index; i < rul->n_numbers - 1; i++)
        strcpy(rul->numbers[i], rul->numbers[i+1]);
    rul->n_numbers--;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    size_t i = find_number(rul, num);
    if (i != rul->n_numbers)
        remove_number_at_index(rul, i);
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}
...
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

README.txt | rul_tests.c | rul.h | rul.c | <empty>

```
...
static size_t find_number(struct rul * rul, const char * num)
{
    for (size_t i = 0; i < rul->n_numbers; i++)
        if (strcmp(rul->numbers[i], num) == 0)
            return i;
    return rul->n_numbers;
}

static void remove_number_at_index(struct rul * rul, size_t index)
{
    for (size_t i = index; i < rul->n_numbers - 1; i++)
        strcpy(rul->numbers[i], rul->numbers[i+1]);
    rul->n_numbers--;
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    size_t i = find_number(rul, num);
    if (i != rul->n_numbers)
        remove_number_at_index(rul, i);
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}
...
```

```
cc rul_tests.c rul.c && ./a.out
All tests passed
```

Develop a recently-used-list module for holding a limited set of unique phone numbers in a Last-In-First-Out order.

Initially, you may assume that numbers added to the list are no longer than 15 digits. You may also assume that the capacity of the list is 10 items. (Imagine that this code is to be used to show a list of the most recently used phone numbers in a very simple cordless phone.)

o) A recently-used-list is initially empty. As you add unique numbers, the list grows, until capacity is reached. 

o) Numbers in the list can be looked up by index. The zeroth element is the most recently used item. 

o) Numbers in the list are unique. If you add a number that already exists, the number is moved to the zeroth element in the list. 

o) When adding a unique number to a full list, the oldest number is dropped to make room for the new entry. 

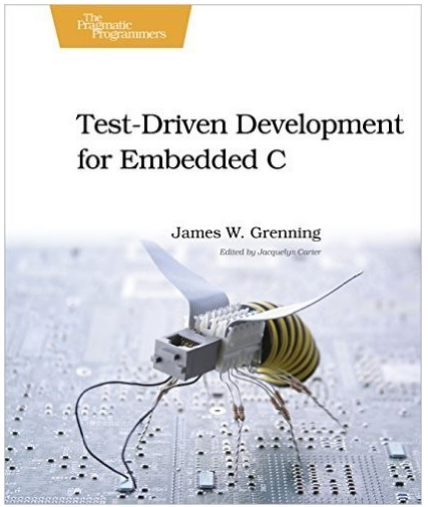
```
#ifndef UTILS_RUL_H_INCLUDED
#define UTILS_RUL_H_INCLUDED

#include <stddef.h>

#define MAX_NUMLEN 15
#define MAX_NUMBERS 10

struct rul {
    size_t n_numbers;
    char numbers[MAX_NUMBERS][MAX_NUMLEN+1];
};

struct rul * rul_init(struct rul * rul);
size_t rul_size(const struct rul * rul);
void rul_add(struct rul * rul, const char * num);
const char * rul_get(const struct rul * rul, size_t index);

#endif
```

```c
#include "rul.h"
#include <string.h>
#include <assert.h>

struct rul * rul_init(struct rul * rul)
{
    rul->n_numbers = 0;
    return rul;
}

size_t rul_size(const struct rul * rul)
{
    return rul->n_numbers;
}

static size_t find_number(struct rul * rul, const char * num)
{
    for (size_t i = 0; i < rul->n_numbers; i++)
        if (strcmp(rul->numbers[i], num) == 0)
            return i;
    return rul->n_numbers;
}

static void remove_number_at_index(struct rul * rul, size_t index)
{
    for (size_t i = index; i < rul->n_numbers - 1; i++)
        strcpy(rul->numbers[i], rul->numbers[i+1]);
    rul->n_numbers--;
}

static void remove_oldest_number(struct rul * rul)
{
    remove_number_at_index(rul, 0);
}

void rul_add(struct rul * rul, const char * num)
{
    if (strlen(num) > MAX_NUMLEN)
        return;
    size_t i = find_number(rul, num);
    if (i != rul->n_numbers)
        remove_number_at_index(rul, i);
    if (rul->n_numbers == MAX_NUMBERS)
        remove_oldest_number(rul);
    assert(rul->n_numbers < MAX_NUMBERS);
    strcpy(rul->numbers[rul->n_numbers], num);
    rul->n_numbers++;
}

const char * rul_get(const struct rul * rul, size_t index)
{
    if (index + 1 > rul->n_numbers)
        return "";
    return rul->numbers[rul->n_numbers - (index + 1)];
}
```

```c
#include "rul.h"
#include <assert.h>
#include <stdio.h>
#include <string.h>

static void test_list_is_initially_empty(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    assert(rul == &myrul);
    assert(rul_size(rul) == 0);
}

static void test_size_of_list_increases_as_we_add_unique_numbers(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(rul_size(rul) == 1);
    rul_add(rul, "1001");
    assert(rul_size(rul) == 2);
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    rul_add(rul, "1004");
    assert(rul_size(rul) == 5);
}

static void test_indexing_into_the_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    assert(strcmp(rul_get(rul, 0), "1000") == 0);
    rul_add(rul, "1001");
    assert(strcmp(rul_get(rul, 0), "1001") == 0);
    assert(strcmp(rul_get(rul, 1), "1000") == 0);
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    assert(strcmp(rul_get(rul, 0), "1003") == 0);
    assert(strcmp(rul_get(rul, 1), "1002") == 0);
    assert(strcmp(rul_get(rul, 2), "1001") == 0);
    assert(strcmp(rul_get(rul, 3), "1000") == 0);
}

static void test_adding_unique_items_to_full_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    char num[MAX_NUMLEN+1];
    for (int i = 0; i < MAX_NUMBERS; i++) {
        sprintf(num, "10%02d", i);
        rul_add(rul, num);
    }
    assert(rul_size(rul) == MAX_NUMBERS);
    assert(strcmp(rul_get(rul, MAX_NUMBERS-1), "1000") == 0);
    rul_add(rul, "1234");
    assert(rul_size(rul) == MAX_NUMBERS);
    assert(strcmp(rul_get(rul, 0), "1234") == 0);
    assert(strcmp(rul_get(rul, MAX_NUMBERS-1), "1001") == 0);
}

static void test_adding_existing_numbers_to_list(void)
{
    struct rul myrul;
    struct rul * rul = rul_init(&myrul);
    rul_add(rul, "1000");
    rul_add(rul, "1001");
    rul_add(rul, "1002");
    rul_add(rul, "1003");
    assert(rul_size(rul) == 4);
    rul_add(rul, "1002");
    assert(rul_size(rul) == 4);
    assert(strcmp(rul_get(rul, 0), "1002") == 0);
}

int main(void)
{
    test_list_is_initially_empty();
    test_size_of_list_increases_as_we_add_unique_numbers();
    test_indexing_into_the_list();
    test_adding_unique_items_to_full_list();
    test_adding_existing_numbers_to_list();

    printf("All tests passed\n");
}
```

!

To learn more about Test-Driven Development in C:



https://www.amazon.com/Driven-Development-Embedded-Pragmatic-Programmers/dp/193435662X

by James W. Grenning

http://cyber-dojo.org/

by Jon Jagger

more presentations like this can be found on http://www.olvemaudal.com/talks