Stacks, Queues, Linked Lists

Computing Laboratory

Indian Statistical Institute

Your tasks during the rest of the semester

1. Create *library* of data structures. (Why?)
2. Test your library.
3. Use library to solve problems.

Stacks: creating the library

Abstract data type

- Collection of elements of some fixed type (say `DATA`)
- Operations: CREATE_STACK, PUSH, POP, DELETE_STACK
  Optional: PRINT_STACK, IS_EMPTY

Implementation issues

- Should the programmer be able to specify size for the stack?
- Should the stack have a fixed capacity, or should the specified size only be used during initialisation?
- How should *overflow* and *underflow* be handled?

Implementation: stack.h

```
typedef int DATA;
/* Some other possibilities:
 * typedef char *DATA;
 * typedef char[BUF_LEN] DATA;
 * typedef struct { ... } DATA;
 */
```

```
typedef struct {
    unsigned int capacity, top;
    DATA *contents;
} STACK;
```

Some options

Option 1

```
extern STACK create_stack();
extern void push(STACK *s, DATA d);
extern DATA pop(STACK *s);
```

Option 2

```
extern STACK *create_stack(unsigned int capacity);
extern int push(STACK *s, DATA *d);
extern int pop(STACK *s, DATA *d);
```

Options: discussion

- `create_stack()` **VS.** `create_stack(unsigned int capacity)`
  - allow user to *suggest* initial capacity, `BUT`
  - stack should grow on demand without overflowing
- `void push(STACK *s, DATA d)` **VS.** `void push(STACK *s, DATA *d)`
  - second option (marginally) preferable
- `DATA pop(STACK *s)` **VS.** `int pop(STACK *s, DATA *d)`
  - second option better for properly handling underflow

Your tasks

1. Create *library* of data structures: `stack.h`, `stack.c`
2. Test your 'library': use `stack-testing.c`
   - include `stack.h` in `stack-testing.c`
   - compilation
     ```
     $ gcc -g -Wall stack.c stack-testing.c
     ```
   - after testing is complete, compile `stack.c` *separately*
     ```
     $ gcc -g -Wall -c stack.c
     ```
     ⟵ creates `stack.o`
3. Use 'library' to solve problems
   - include `stack.h` in `problem1.c`, `problem2.c`, etc.
   - compilation
     ```
     $ gcc -g -Wall stack.o problem1.c
     ```

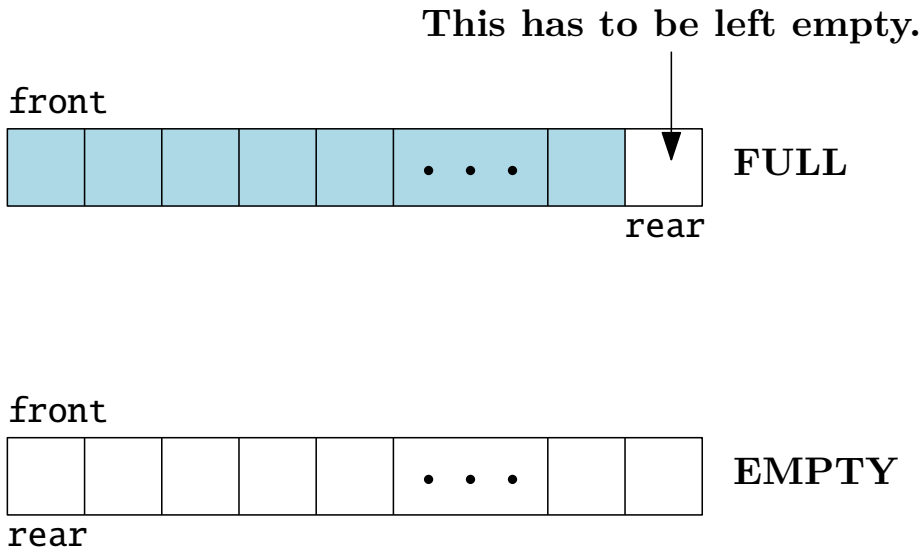
Note extension!

Queues: implementation

```
typedef struct {
    int capacity, num_elements, front, rear;
    DATA *elements;
} QUEUE;
```

- Naive implementation: `elements[0]` is always the `front` of the queue
  - DEQUEUE(): $O(n)$ operation
- Better implementation: use 'circular' array: when index reaches end, wrap around to beginning.

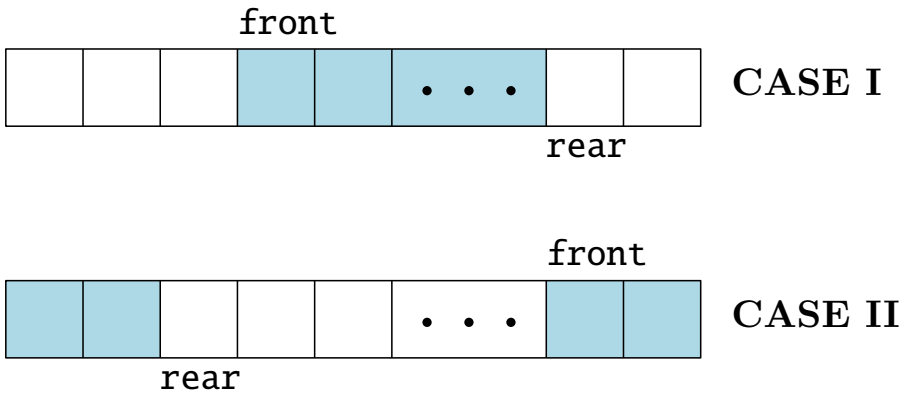
Queues: implementation (contd.)



Empty queue: `rear == front`

Full queue: `(rear + 1) % capacity == front`

Queues: implementation (contd.)



Length of queue: `rear - front` OR `capacity - (front - rear)`

Lists

Operations

- `LIST init_list();`
- `void insert(LIST *l, DATA d, unsigned int index);`
- `DATA delete(LIST *l, unsigned int index);`
- `DATA find_index(LIST *l, DATA d);`
- `DATA find_value(LIST *l, unsigned int index);`
- `void print_list(LIST *l);`

Easy implementation: use arrays

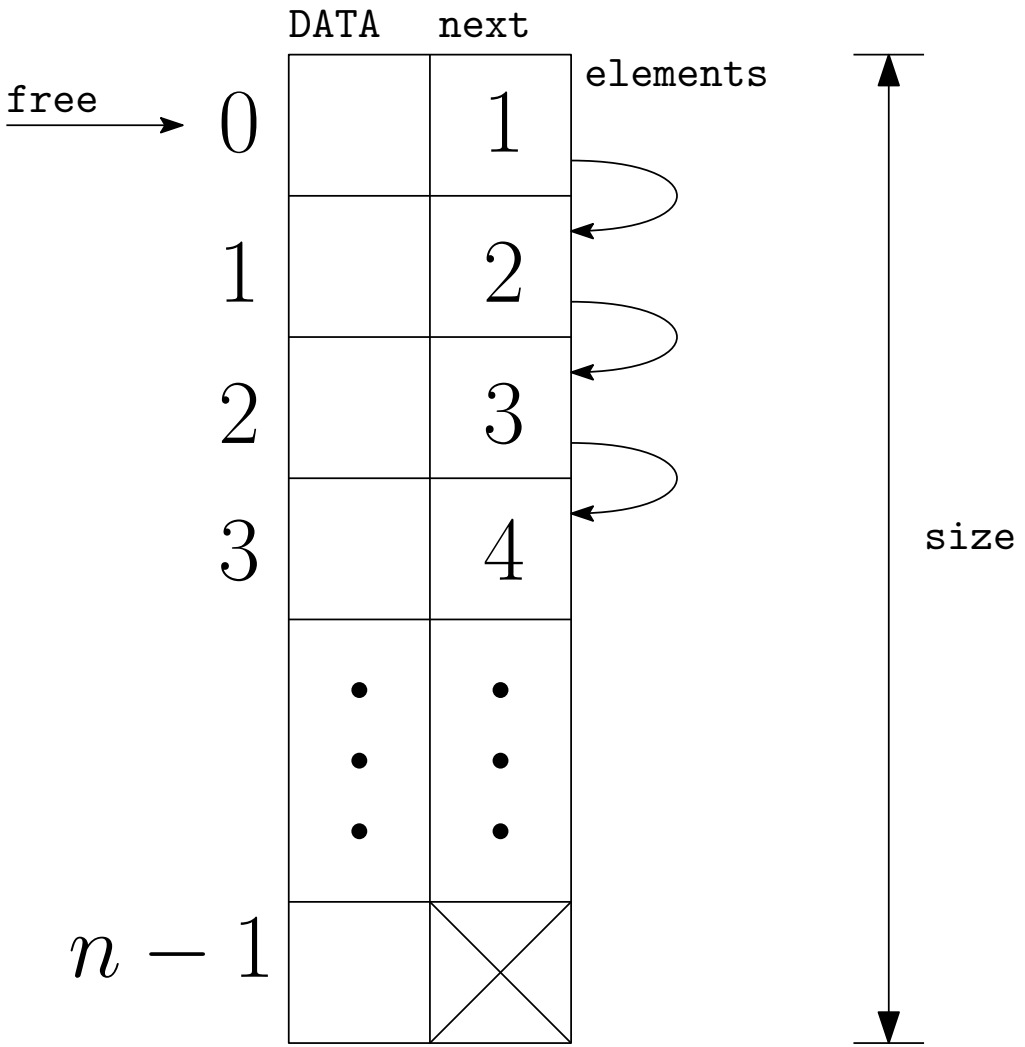
- `insert(LIST *l, DATA d, unsigned int index);` – inefficient: involves moving array elements
- `delete(LIST *l, unsigned int index);` – inefficient: involves moving array elements
- `find_index(LIST *l, DATA d);`
- `find_value(LIST *l, unsigned int index);` – very efficient

Traditional implementation: using pointers


```
typedef struct node {
    DATA data;
    struct node *next, *prev;
} NODE;

typedef struct {
    unsigned int length;
    NODE *head, *tail;
} LIST;

NODE *create_node(DATA d) {
    NODE *nptr;
    if (NULL == (nptr = Malloc(1, NODE)))
       ERR_MESG("out of memory");
    nptr->data = d;
    nptr->next = NULL;
    return nptr;
}
```

Alternative implementation: using arrays

```
typedef struct {
    DATA data;
    int next;
} NODE;

typedef struct {
    int head, free;
    int length, size;
    NODE *elements;
} LIST;
```



Alternative implementation

```
LIST create_list(int n) {
    int i;
    LIST l;
    if (NULL ==
        (l.elements = Malloc(n, NODE)))
       ERR_MESG("out of memory");
    for (i = 0; i < n-1; i++)
        l.elements[i].next = i+1;
    l.elements[n-1].next = -1;
    l.size = n;
    l.free = 0;
    l.head = -1;
    l.length = 0;
    return l;
}
```

Exercises I

1. Implement the required functions for the STACK and QUEUE ADTs in `stack.c` and `queue.c`.
   Also write `stack-testing.c` and `queue-testing.c` to test your implementation. You may use the provided files `stack-ops.txt` and `queue-ops.txt` for testing.
2. Given a list of numbers (provided as command line arguments), write a program to compute the nearest larger value for the number at position i (nearness is measured in terms of the difference in array indices). For example, in the array $[1, 4, 3, 2, 5, 7]$, the nearest larger value for 4 is 5. Implement a naive, $O(n^2)$ time algorithm, as well as an $O(n)$ time algorithm for this problem.

Exercises II

3. Given a set of sorted numbers as user input, write a program that both removes the duplicate elements, and returns the number of distinct elements.