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 single **(but arbitrary)** type
- Operations: CREATE_STACK, PUSH, POP, DELETE_STACK
  Optional: PRINT_STACK, IS_EMPTY

Implementation issues: decide parameters and return values for above functions)

- Programmer should be able to specify initial size for the stack.
- Ideally, stack should grow on demand without overflowing.
  $\Rightarrow$ use `realloc()` to double the stack size as needed
- How should *underflow* be handled?

Some options

Option 1

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

Option 2

```
extern int create_stack(STACK *s, unsigned int capacity);
// OR extern STACK *create_stack(unsigned int capacity);
extern int push(STACK *s, DATA *d);
extern int pop(STACK *s, DATA *d);  // better for handling underflow
```

Generic stacks

```
#ifndef _GSTACK_
#define _GSTACK_

#include "common.h"

typedef void (*PRINTER_FN)(void *);

typedef struct {
    void *elements;
    size_t capacity, top, element_size;
    PRINTER_FN print_element;
} STACK;

extern int create_stack (STACK *s, size_t element_size, PRINTER_FN print_element,
      uint capacity);
extern void delete_stack(STACK *s);
/* extern bool is_empty(const STACK *s); */
extern int push(STACK *s, void *eptr);
extern int pop(STACK *s, void *eptr);
extern void print_stack(STACK *s);

#endif // _GSTACK_
```

Implementation notes

Use `memcpy()` for `push()` and `pop()`

```
stackTopAddress = (char *) s->elements + s->top * s->element_size;

/* PUSH */
memcpy(stackTopAddress, eptr, s->element_size);

/* POP */
memcpy(eptr, stackTopAddress, s->element_size);
```

Your tasks

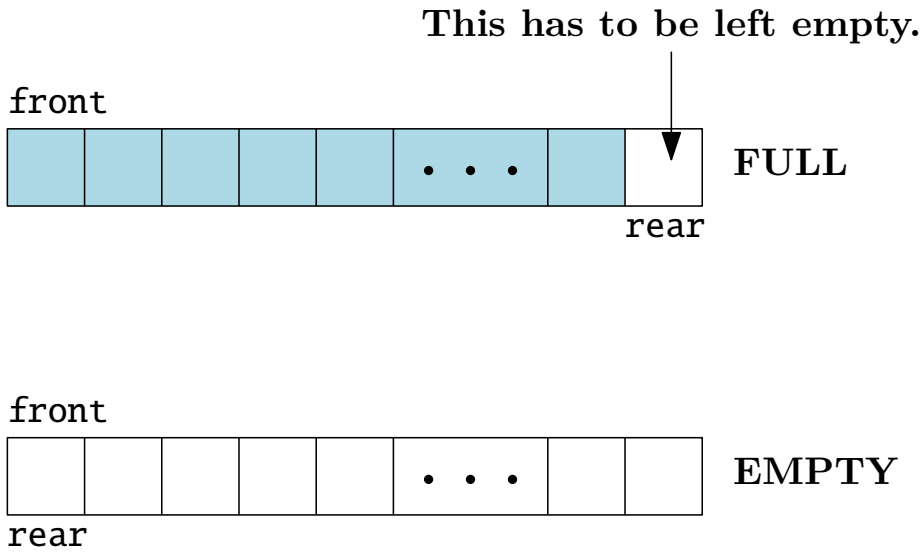
1. Create *library* of data structures:
   `generic-stack.h`, `generic-stack.c`
2. Test your 'library': use `gstack-testing.c`
   - include `generic-stack.h` in `gstack-testing.c`
     (but don't use `#include "generic-stack.c"`! Why??)
   - compilation
     ```
     $ gcc -g -Wall generic-stack.c gstack-testing.c
     ```
   - after testing is complete, compile `generic-stack.c` *separately*
     ```
     $ gcc -g -Wall -c generic-stack.c
     ```
     ⟵ creates `generic-stack.o`
3. Use 'library' to solve problems
   - include `generic-stack.h` in `problem1.c`, `problem2.c`, etc.
   - compilation
     ```
     $ gcc -g -Wall generic-stack.o problem1.c
     ```
     Note extension!

Basic (non-generic) 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.

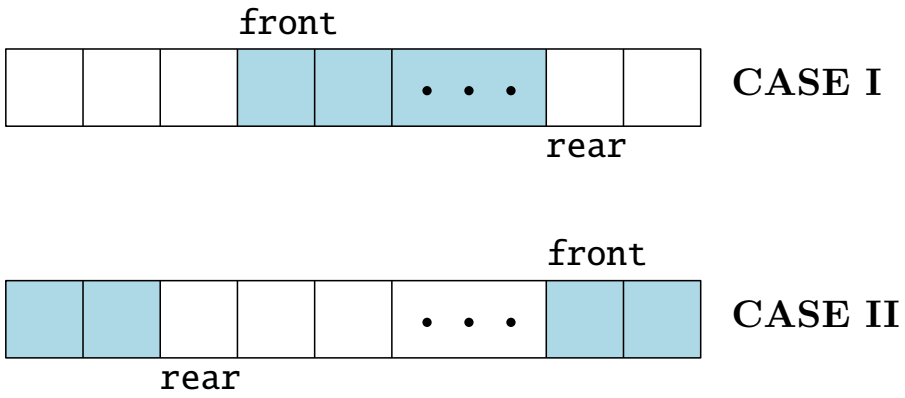
Queues: implementation (contd.)



Empty queue: `rear == front`

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

Queues: implementation (contd.)



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

Exercises I

1. Implement the required functions for the GENERIC-QUEUE.
   Alsopso write `gqueue-testing.c` to test your implementation.
2. Implement a generic data structure SEQUENCE to hold a list of elements of any one of the following types: `int`, `float`, or null-terminated strings (`char *`) (all elements in a particular list will be of the same type). Your data structure should support the operations given below.
   - `void get_element(SEQUENCE s, size_t i)`: prints the numerical value of the i-th element of the sequence `s`, if it exists. The function should print an error message if the i-th element does not exist.
     The numerical value of an integer n is n itself; the numerical value of a floating point number f is the integer nearest to f; the numerical value of a string s of alphanumeric characters is the sum of the ASCII values of all the characters contained in s. Note that the numerical value of an empty string is 0.

Exercises II

- `size_t length(SEQUENCE s)`: returns the number of elements in the sequence `s`.
- `void summation(SEQUENCE s)`: prints the sum of the numerical values of the elements in the sequence `s`. Please see above for the definition of numerical value for sequences of various types. For a sequence containing no elements, `summation(s)` should print 0.

3. Write a program that takes N sequences as input (from stdin), and prints the sequence `s` for which `summation(s)` is the maximum.

   Input format: A positive integer N, followed by N sequences, one per line. Each of these lines will begin with `i`, `f` or `s` to specify whether the sequence on that particular line consists of `int`, `float` or string elements. This single letter will be followed by a non-negative integer that specifies the number of elements in the sequence, which in turn will be followed by the elements of the sequence.

Exercises III

4. 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.
5. Write a program that can accept a generic but homogeneous list of elements from the user and report the next greater element for every element in the list in linear time. Consider a lexicographic comparison of the elements (i.e., use `memcmp()`).

Linked lists

Linked lists

Operations

- `LIST init_list();`
- Insertion
  - at specified index (position)
    `insert(LIST *l, DATA d, unsigned int index);`
  - in the beginning / at the end
- Deletion
  - at specified index (position)
    `delete(LIST *l, unsigned int index);`
  - from the beginning / end
- Searching
  - `DATA find_index(LIST *l, DATA d);`
  - `DATA find_value(LIST *l, unsigned int index);`
- `void print_list(LIST *l);`

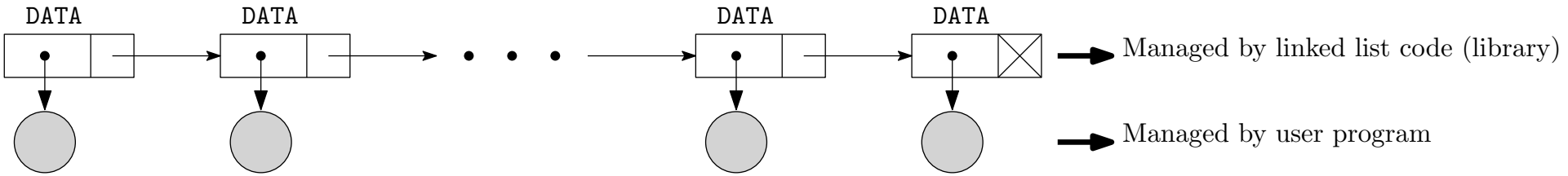
Easy implementation: use arrays

- Insert somewhere in the middle: inefficient; involves moving array elements
- Delete from somewhere in the middle: inefficient: involves moving array elements
- Search by index – very efficient

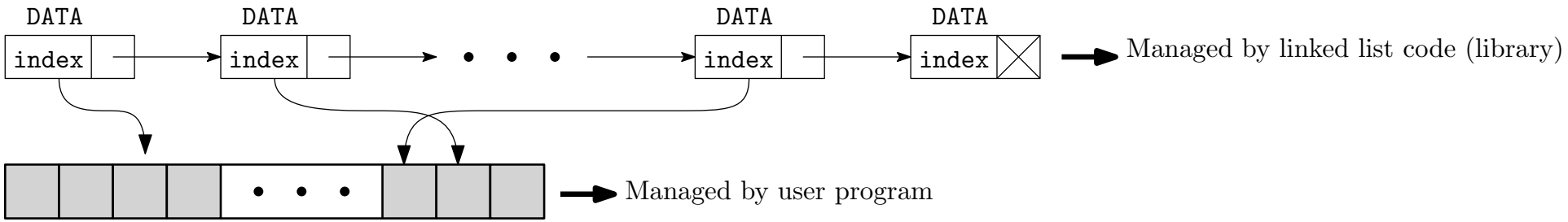
Traditional implementation: using pointers

Option 1:

Option 2: `typedef void *DATA;`



Option 3: `typedef int DATA;`



Traditional implementation: Option 2

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

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

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