Stacks and Queues

- Stacks and queues ubiquitous data-structure in computing.
- Part of many important algorithms.
- Good example of abstract data types.
- Good example to practice programming with arrays.
- Good example to practice programming with linked lists.
a stack is a collection of items such that the last item to enter is the first one to exit
“last in, first out” (LIFO)
based on the idea of a stack of books, or plates
essential Stack operations:

- push() // add new item to stack
- pop() // remove top item from stack

additional Stack operations:

- top() // fetch top item (but don’t remove it)
- size() // number of items
- is_empty()
Stack Applications

- page-visited history in a Web browser
- undo sequence in a text editor
- checking for balanced brackets
- HTML tag matching
- postfix (RPN) calculator
- chain of function calls in a program
typedef struct stack *stack_t;
stack_t stack_create(void);
void stack_free(stack_t stack);
void stack_push(stack_t stack, int item);
int stack_pop(stack_t stack);
int stack_is_empty(stack_t stack);
int stack_top(stack_t stack);
int stack_size(stack_t stack);
Stack - Abstract Data Type - using C Interface

```c
stack_t s;
s = stack_create();
stack_push(s, 10);
stack_push(s, 11);
stack_push(s, 12);
printf("%d\n", stack_size(s));  // prints 3
printf("%d\n", stack_top(s));  // prints 12
printf("%d\n", stack_pop(s));  // prints 12
printf("%d\n", stack_pop(s));  // prints 11
printf("%d\n", stack_pop(s));  // prints 10
```

- Implementation of stack is **opaque** (hidden from user).
- User programs can not depend on how stack is implementated.
- Stack implementation can change without risk of breaking user programs.
- This type of **information hiding** is crucial to managing complexity in large software systems.
Queue Abstract Data Type

- a *queue* is a collection of items such that the *first* item to enter is the *first* one to exit, i.e. “first in, first out” (FIFO)
- based on the idea of queuing at a bank, shop, etc.
- Essential Queue operations:
  - enqueue() // add new item to queue
  - dequeue() // remove front item from queue
- Additional Queue operations:
  - front() // fetch front item (but don’t remove it)
  - size() // number of items
  - is_empty()
Queue Applications

• waiting lists, bureaucracy
• access to shared resources (printers, etc.)
• phone call centres
• multiple processes in a computer
queue_t queue_create(void);
void queue_free(queue_t queue);
void queue_enqueue(queue_t queue, int item);
int queue_dequeue(queue_t queue);
int queue_is_empty(queue_t queue);
int queue_front(queue_t queue);
int queue_size(queue_t queue);
Queue - Abstract Data Type - C Interface

```c
queue_t q;
q = queue_create();
queue_enqueue(q, 10);
queue_enqueue(q, 11);
queue_enqueue(q, 12);
printf("%d\n", queue_size(q)); // prints 3
printf("%d\n", queue_front(q)); // prints 10
printf("%d\n", queue_dequeue(q)); // prints 10
printf("%d\n", queue_dequeue(q)); // prints 11
printf("%d\n", queue_dequeue(q)); // prints 12
```
• Again implementation of stack is **opaque**.

• Queue implementation can change without risk of breaking user programs.
• A stack can be implemented using a linked list, by adding and removing at the head [push() and pop()]
• For a queue, we need to either add or remove at the tail
  ▶ Can either of these be done efficiently?
Adding to the Tail of a List

- adding an item at the tail is achieved by making the last node of the list point to the new node
- we first need to scan along the list to find the last item
Adding to the Tail of a List

```
struct node *add_to_tail( *new_node, struct node *head) {
    if (head == NULL) {
        // list is empty
        head = new_node;
    } else {
        // list not empty
        struct node *node = head;
        while (node->next != NULL) {
            node = node->next; // scan to end
        }
        node->next = new_node;
    }
    return head;
}
```
Efficiency Issues

Unfortunately, this implementation is very slow. Every time a new item is inserted, we need to traverse the entire list (which could be very large).
We can do the job much more efficiently if we retain a direct link to the last item or “tail” of the list:

```c
if (tail == NULL) {  // list is empty
    head = node;
} else {              // list not empty
    tail->next = node;
}
tail = node;
```

Note: there is no way to efficiently remove items from the tail. (Why?)
• a *queue* is a collection of items such that the *first* item to enter
  is the *first* one to exit, i.e. “first in, first out” (FIFO)
• based on the idea of queueing at a bank, shop, etc.
We can use this structure to implement a queue efficiently:

```c
typedef struct queue Queue;

struct queue {
    struct node *head;
    struct node *tail;
    int size;
};
```
Making a new Queue

```c
queue_t *makeQueue() {
    queue_t *q = (queue_t *)malloc(sizeof (queue_t));
    if (q == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    q->head = NULL;
    q->tail = NULL;
    q->size = 0;
    return q;
}
```
Adding a new Item to a Queue

```c
void enqueue(struct node *new_node, queue_t *q) {
    if (q->tail == NULL) { // queue is empty
        q->head = new_node;
    } else { // queue not empty
        q->tail->next = new_node;
    }
    q->tail = new_node;
    q->size++;
}
```
Removing an Item from a Queue

```c
struct node *dequeue(queue_t *q) {
    struct node *node = q->head;
    if (q->head != NULL) {
        if (q->head == q->tail) { /* only one item */
            q->tail = NULL;
        }
        q->head = q->head->next;
        q->size--;
    }
    return node;
}
```
Example: queue.c

```c
int main(void) {
    queue_t *q = makeQueue();
    struct node *node;
    int ch;

    while ((ch = getchar()) != EOF) {
        if (ch == '-') {
            node = dequeue(q);
            if( node != NULL ) {
                printf("Dequeueing %c\n", node->data );
                free( node );
            }
        }
    }
}
```
... else if (ch == '\n') {
    print_list(q->head);
}
else {
    enqueue(makeNode(ch), q);
}
}
free_list(q->head);
return 0;
Reverse Polish Notation

Some early calculators and programming languages used a convention known as *Reverse Polish Notation* (RPN) where the operator comes after the two operands rather than between them:

1 2 +
result = 3
3 2 *
result = 6
4 3 + 6 *
result = 42
1 2 3 4 + * +
result = 15
A calculator using RPN is called a *Postfix Calculator*; it can be implemented using a stack:

- when a number is entered: push it onto the stack
- when an operator is entered: pop the top two items from the stack, apply the operator to them, and push the result back onto the stack.
int main(void) {
    struct node *list = NULL;
    int num;
    int a,b, num;
    while ((ch = getc(stdin)) != EOF) {
        if (ch == '
') {
            printf("Result: %d\n", list->data);
        }
        else if (isdigit(ch)) {
            ungetc(ch, stdin); // put first digit back
            scanf("%d", &num); // now scan entire number
            list = push(makeNode(num), list);
        }
    }
}

else if (ch == '+' || ch == '-' || ch == '*') {
    if (list != NULL) {
        a = list->data;  // fetch top item
        list = pop(list);
        if (list != NULL) {
            b = list->data;  // fetch 2nd item
            list = pop(list);
            switch (ch) {
                case '+': num = b + a;  break;
                case '-': num = b - a;  break;
                case '*': num = b * a;  break;
            }
            list = push(make_node(num), list);
        }
    }
}