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

Stack - Abstract Data Type

- 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

Stack - Abstract Data Type - C Interface

```c
typedef struct stack internals *stack;
stack stack_create(void);
void stack_free(stack stack);
void stack_push(stack stack, int item);
int stack_pop(stack stack);
int stack_is_empty(stack stack);
int stack_top(stack stack);
int stack_size(stack stack);
```
Stack - Abstract Data Type - using C Interface

```c
stack 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 implemented.
- 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 queueing 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 - Abstract Data Type - C Interface

```c
queue queue_create(void);
void queue_free(queue queue);
void queue_enqueue(queue queue, int item);
int queue_dequeue(queue queue);
int queue_is_empty(queue queue);
int queue_front(queue queue);
int queue_size(queue queue);
```
Queue - Abstract Data Type - C Interface

```c
queue 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.

Implementing A Stack with a Linked List

• 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

```c
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?)

Queues

- 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.

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.

```c
#include <stdio.h>
#include <ctype.h>
#include "stack.h"

int main(void) {
    int ch;
    stack s = stack_create();
    while ((ch = getc(stdin)) != EOF) {
        if (ch == 'n') {
            printf("Result: %d\n", stack_pop(s));
        } else if (isdigit(ch)) {
            ungetc(ch, stdin); // put first digit back
            int num;
            scanf("%d", &num); // now scan entire number
            stack_push(s, num);
        } else if (ch == '+' || ch == '-' || ch == '*') {
            int a = stack_pop(s);
            int b = stack_pop(s);
            int result;
            if (ch == '+') {
                result = b + a;
            } else if (ch == '-') {
                result = b - a;
            } else {
                result = b * a;
            }
            stack_push(s, result);
        }
    }
}
```