Stacks and Queues

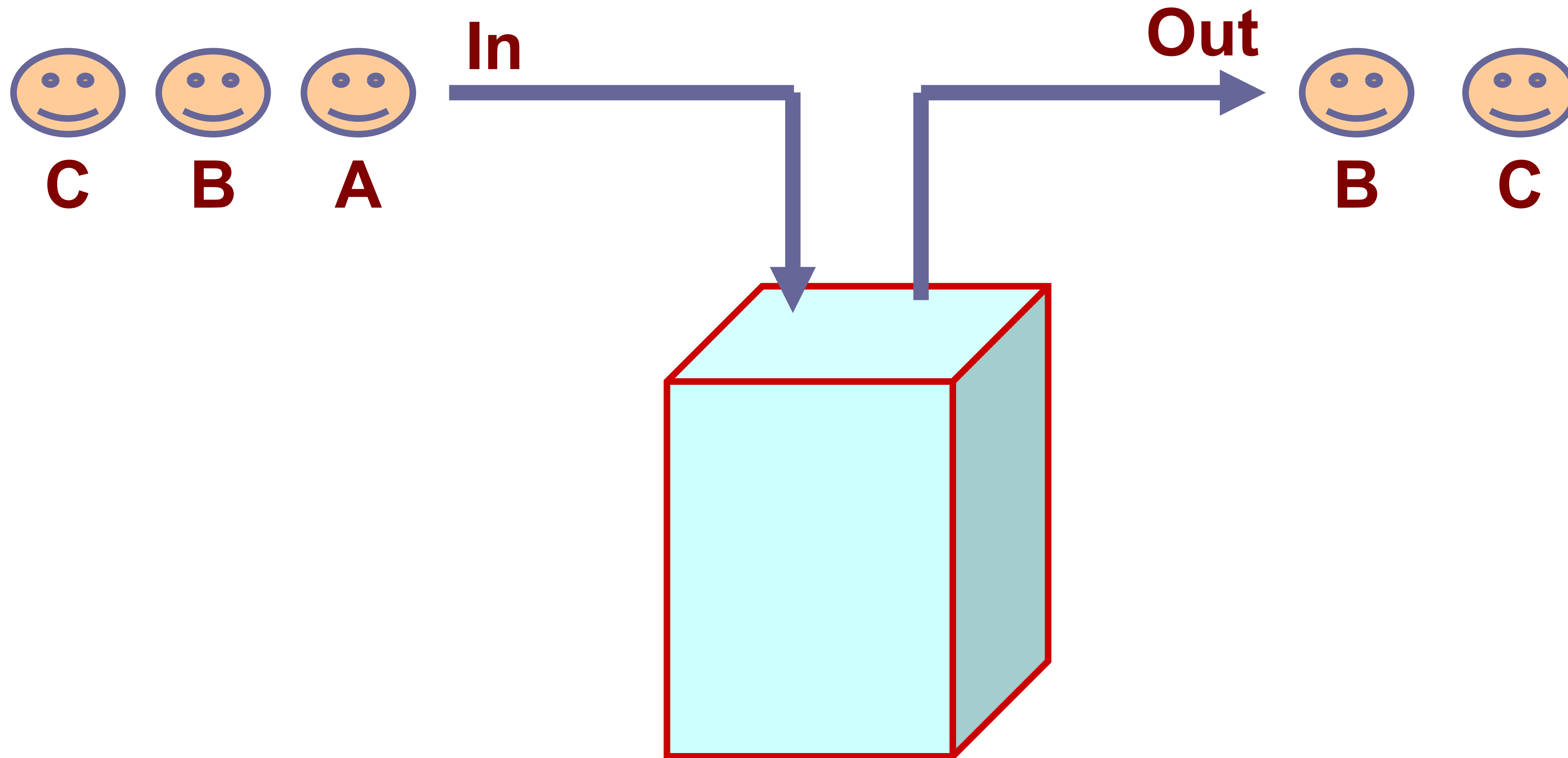
Two useful ADTs

Stacks

Basic Definitions

- A stack is an ordered list in which all insertions and all deletions occur at one end of the list.
- That end is called the top of the stack.
- Insertion is called push.
- Deletion is called pop.
- A stack is called a last-in-first-out (LIFO) list.

Visualization of a Stack (Last In First Out)



The interface for a stack

A stack ADT can be specified by the following basic operations. We assume that we are maintaining a stack of characters. In practice, the data type for each element of a stack can be of any data type.

S = init()	Initialize S to an empty stack.
isEmpty(S)	Returns "true" if and only if the stack S is empty
isFull(S)	Returns "true" if and only if the stack S has a bounded size and holds the maximum number of elements that it can
top(S)	Return the element at top of the stack S, or error if the stack is empty
S = push(S, ch)	Push the character ch at the top of the stack S
S = pop(S)	Pop an element from the top of the stack S
print(S)	Print the elements of the stack S from top to bottom

Stacks can be implemented

a) Using arrays
b) Using linked list

. . .

Array implementation: Basic Idea

In the array implementation, we would:

- Declare an array of fixed size (which determines the maximum size of the stack).
- Keep a variable top which always points to the “top” of the stack (contains the array index of the “top” element)
- Both push and pop will happen at the top of the stack
- If we assume that the stack elements are stored in the array starting from the index 0, it is convenient to take the top as the maximum index of an element in the stack.
- The other boundary 0, can in principle be treated as the top, but insertions and deletions at the location 0 call for too many relocations of array elements.

Declaration and Initialization

```
#define MAXLEN 100

typedef struct {
      char element [MAXLEN];
      int top;
} stack;
```

```
stack init ( )
{
      stack S;

      S.top = -1;
      return S;
}
```

Check for empty stack

```
#define MAXLEN 100

typedef struct {
        char element [MAXLEN];
        int top;
} stack;
```

```
int isEmpty (stack S)
{
        return (S.top == -1);
}
```

Check if stack is full

```
#define MAXLEN 100

typedef struct {
        char element [MAXLEN];
        int top;
} stack;
```

```
int isFull (stack S)
{
    return (S.top == MAXLEN - 1);
}
```

Obtain the top of the stack

```
#define MAXLEN 100

typedef struct {
        char element [MAXLEN];
        int top;
} stack;
```

```
char top (stack S)
{
        if (isEmpty(S)) {
                printf("top: Empty 
stack\n");
                return '\0';
        }
        return S.element [S.top];
}
```

Push

```
#define MAXLEN 100

typedef struct {
      char element [MAXLEN];
      int top;
} stack;
```

```
stack push (stack S, char ch)
{
    if (isFull(S)) {
       printf("push: Full stack\n");
             return S;
    }
    ++S.top;
    S.element[S.top] = ch;
    return S;
}
```

Pop

```
#define MAXLEN 100

typedef struct {
        char element [MAXLEN];
        int top;
} stack;
```

```
stack pop ( stack S )
{
     if (isEmpty(S)) {
                printf ("pop: Empty
stack");
                return S;
     }
     --S.top;
     return S;
}
```

Print the contents of the stack

```
#define MAXLEN 100

typedef struct {
        char element[MAXLEN];
        int top;
} stack;
```

```
void print (stack S)
{
    int i;

    for (i = S.top; i >= 0; --i)
                printf ("%c",
S.element[i]);
}
```

Example main function and its output

```
int main ()
{
   stack S;

   S = init(); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'d'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'f'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'a'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'x'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
}
```

Output:

```
top: Empty stack
Current stack :  with top = .
Current stack : d with top = d.
Current stack : fd with top = f.
Current stack : afd with top = a.
Current stack : fd with top = f.
Current stack : xfd with top = x.
Current stack : fd with top = f.
Current stack : d with top = d.
top: Empty stack
Current stack :  with top = .
pop: Empty stack
top: Empty stack
Current stack :  with top = .
```

Implementation of a stack using a linked list

- A linked list is an ordered list, and can be used to implement a stack.
- A linked list has two ends: head and tail.
- Both insertion and deletion are easy at the head.
- Insertion is easy at the tail if we maintain a tail pointer.
- Deletion is time-consuming even in the presence of a tail pointer.
- So the obvious choice is: The top of the stack is at the head of the linked list.

Implementation of the stack ADT functions

```
typedef struct _node {
   char element;
   struct _node *next;
} node;

typedef node *stack;
```

```
stack init ()
{
      return NULL;
}

int isEmpty ( stack S )
{
      return (S == NULL);
}
```

```
int isFull ( stack S )
{
      return 0;
}

char top ( stack S )
{
      if (isEmpty(S)) return '\0';
      return S -> element;
}

void print ( stack S )
{
      while (S) {
            printf("%c", S -> element);
            S = S -> next;
      }
}
```

Implementation of the stack ADT functions (continued)

```
stack push ( stack S, char ch )
{
        node *p;

        p = (node *)malloc(sizeof(node));
        p -> element = ch;
        p -> next = S;
        return p;
}
```

```
stack pop ( stack S )
{
        node *p;

    if (isEmpty(S)) {
        printf("pop: empty stack\n");
        return S;
    }
    p = S;
    S = S -> next;
    free(p);
    return S;
}
```

Exactly the same main function works

```
int main ()
{
   stack S;

   S = init(); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'d'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'f'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'a'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = push(S,'x'); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
   S = pop(S); printf("Current stack : "); print(S); printf(" with top = %c.\n", top(S));
}
```

Output:

```
top: Empty stack
Current stack :  with top = .
Current stack : d with top = d.
Current stack : fd with top = f.
Current stack : afd with top = a.
Current stack : fd with top = f.
Current stack : xfd with top = x.
Current stack : fd with top = f.
Current stack : d with top = d.
top: Empty stack
Current stack :  with top = .
pop: Empty stack
top: Empty stack
Current stack :  with top = .
```

Implementation of a stack using dynamic arrays

```
typedef struct {
        int allocsize;
        char *element;
        int top;
} stack;
```

```
stack init ( int n )
{
        stack S;
        S.element = (char *)malloc(n * sizeof(char));
        S.allocsize = n;
        S.top = -1;
    return S;
}
```

Note: This init() function has a behavior different from the original specifications of the stack ADT. The main function given earlier will not work. If you need to follow the original ADT specification, you can choose an initial size (like 64) of the element array, so no user input is required. This implementation is meaningful to the users who accept the new specification of init().

Implementation of a stack using dynamic arrays

```
typedef struct {
        int allocsize;
        char *element;
        int top;
} stack;
```

```
stack push ( stack S, char ch )
{
        ++S.top;
        if (S.top >= S.allocsize) {
                S.allocsize *= 2;
                S.element = (char *)realloc(S.element,
                                S.allocsize * sizeof(char));
    }
        S.element[S.top] = ch;
        return S;
}
```

An Application of Stacks: Parenthesis matching

The Basic Problem

Given a parenthesized expression, to test whether the expression is properly parenthesized.

- Whenever a left parenthesis is encountered, it is pushed in the stack.
- Whenever a right parenthesis is encountered, pop from stack and check if the parentheses match.
- Works for multiple types of parentheses

  ( ), { }, [ ]

Pseudocode

```
while (not end of string) do {
     a = get_next_token();
     if (a is '(' or '{' or '[') push (a);
     if (a is ')' or '}' or ']') {
              if (isempty()) {
             printf ("Not well formed");
             exit();
              }
                     x = pop();
                     if (a and x do not match) {
             printf ("Not well formed");
             exit();
         }
     }
}
if (not isempty()) printf ("Not well formed");
```

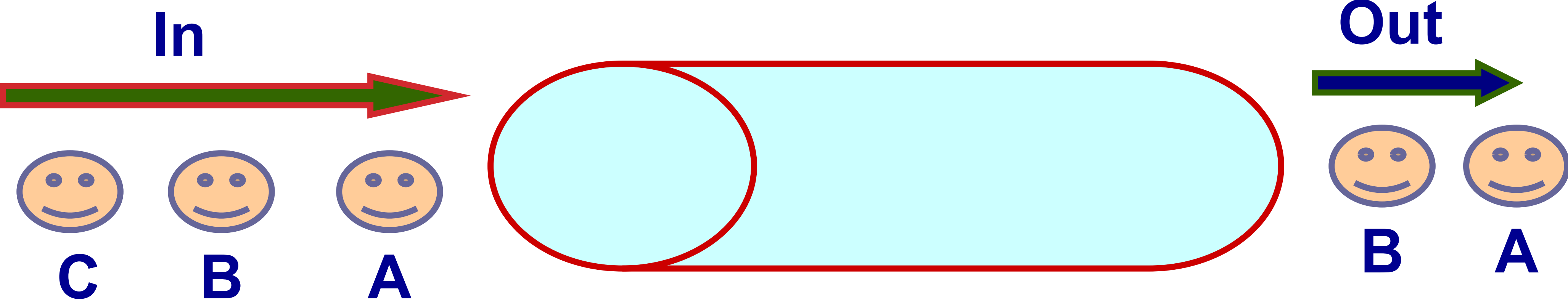
```
Given expression: (a + (b - c) * (d + e))

Search string for parenthesis from left to right:

    (:        push ('(')    Stack: (
    (:        push ('(')    Stack: ( (
    ):        x = pop() = ( Stack: (          MATCH
    (:        push ('(')    Stack: ( (
    ):        x = pop() = ( Stack: (          MATCH
    ):        x = pop() = ( Stack: EMPTY      MATCH


Given expression: (a + (b - c)) * d)

Search string for parenthesis from left to right:

    (:        push ('(')    Stack: (
    (:        push ('(')    Stack: ( (
    ):        x = pop() = ( Stack: (          MATCH
    ):        x = pop() = ( Stack: EMPTY      MATCH
    ):        x = pop() = ( Stack: ?          MISMATCH
```

Queues

Basic Definitions

- A queue is an ordered list in which all insertions occur at one end of the list, and all deletions occur at the other end of the list.
- The insertion end is called the back or the rear of the queue.
- The deletion end is called the front or the head of the queue.
- Insertion is called enqueue.
- Deletion is called dequeue.
- A queue is called a first-in-first-out (FIFO) list.

Visualization of a Queue (First In First Out)



The interface for a queue

The following functions specify the operations on the queue ADT. We are going to maintain a queue of characters. In practice, each element of a queue can be of any well-defined data type.

Q = init()	Initialize the queue Q to the empty queue.
isEmpty(Q)	Returns "true" if and only if the queue Q is empty.
isFull(Q)	Returns "true" if and only if the queue Q is full, provided that there is a limit on the maximum size of the queue.
front(Q)	Returns the element at the front of queue Q or error if the queue is empty.
Q = enqueue(Q,ch)	Inserts the element ch at the back of the queue Q. Insertion request in a full queue leads to failure along with some appropriate error message.
Q = dequeue(Q)	Delete one element from the front of the queue Q. A dequeue attempt from an empty queue should lead to failure and appropriate error messages.
print(Q)	Print the elements of the queue Q from front to back.

Queues can be implemented

a) Using arrays
b) Using linked list
. . .

Array implementation of a queue: The basic idea

- We maintain two indices to represent the front and the back of the queue.
- For an enqueue operation, the back index is incremented and the new element is written in this location.
- For a dequeue operation, on the other hand, the front is simply advanced by one position.
- As elements are enqueued and dequeued, the entire queue moves down the array and the back index may hit the right end of the array, even when the size of the queue is smaller than the capacity of the array.

Array implementation of a queue: Some points to consider

- To avoid waste of space, we allow our queue to *wrap* at the end. This means that after the back pointer reaches the end of the array and needs to proceed further down the line, it comes back to the zeroth index, provided that there is space at the beginning of the array to accommodate new elements.

- Thus, the array is now treated as a circular one with index MAXLEN treated as 0, MAXLEN + 1 as 1, and so on. That is, index calculation is done modulo MAXLEN.

- We still don't have to maintain the total queue size. As soon as the back index attempts to collide with the front index modulo MAXLEN, the array is considered to be full.

Array implementation of a queue: More points to consider

- A little thought reveals that under this wrap-around technology, there is no difference between a full queue and an empty queue with respect to arithmetic modulo MAXLEN.

- This problem can be overcome if we allow the queue to grow to a maximum size of MAXLEN – 1. This means we are going to lose one available space, but that loss is inconsequential.

- Now the condition for full array is that the front index is two locations ahead of the back modulo MAXLEN, whereas the empty array is characterized by that the front index is just one position ahead of the back again modulo MAXLEN.

Declaration and Initialization

```
#define MAXLEN 100

typedef struct {
      char element [MAXLEN];
      int front;
      int back;
} queue;
```

```
queue init ()
{
      queue Q;
      Q.front = 0;
      Q.back = MAXLEN - 1;
      return Q;
}
```

Check for empty queue

```
#define MAXLEN 100

typedef struct {
        char element [MAXLEN];
        int front;
        int back;
} queue;
```

```
int isEmpty (queue Q)
{
   if (Q.front == (Q.back + 1) % MAXLEN)
               return 1;
   else
               return 0;
}
```

Check for full queue

```
#define MAXLEN 100

typedef struct {
        char element [MAXLEN];
        int front;
        int back;
} queue;
```

```
int isFull (queue Q)
{
   if (Q.front == (Q.back + 2) % MAXLEN)
              return 1;
   else
              return 0;
}
```

Obtain the front of the queue

```
#define MAXLEN 100

typedef struct {
        char element [MAXLEN];
        int front;
        int back;
} queue;
```

```
char front (queue Q)
{
        if (isEmpty(Q)) {
             printf("front: empty Queue");
             return '\0';
        }
        return Q.element [Q.front];
}
```

Enqueue

```
#define MAXLEN 100

typedef struct {
        char element
[MAXLEN];
        int front;
        int back;
} queue;
```

```
queue enqueue (queue Q, char ch) {

        if (isFull(Q)) {
                printf("enqueue: Queue is full\n");
                return Q;
        }

        ++Q.back;
        if (Q.back == MAXLEN) Q.back = 0;
        Q.element[Q.back] = ch;
        return Q;
}
```

Dequeue

```
#define MAXLEN 100

typedef struct {
      char element [MAXLEN];
      int front;
      int back;
} queue;
```

```
queue dequeue (queue Q) {

      if (isEmpty(Q)) {
         printf("dequeue: empty Queue\n");
         return Q;
      }

      ++Q.front;
      if (Q.front == MAXLEN) Q.front = 0;
      return Q;
}
```

Print the contents of the queue

```
#define MAXLEN 100

typedef struct {
        char element [MAXLEN];
        int front;
        int back;
} queue;
```

```
void print (queue Q) {
        int i;
        if (isEmpty(Q)) return;

        i = Q.front;
        while (1) {
                printf ("%c", Q.element[i]);
                if (i == Q.back) break;
                if (++i == MAXLEN) i = 0;
        }
}
```

Example main function and its output

```
int main ()
{
   queue Q;

   Q = init(); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'h'); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'w'); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'r'); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'c'); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
}
```

Output:

```
Current queue :
Current queue : h
Current queue : hw
Current queue : hwr
Current queue : wr
Current queue : r
Current queue : rc
Current queue : c
Current queue :
dequeue: Queue is empty
Current queue :
```

Linked-list implementation of a queue

- We need to maintain two pointers to the head and the tail nodes of the list.
- Insertion is easy at both the ends.
- Deletion is easy at the front, but costly at the tail.
- So the natural choice is:
  - The head of the list is the front of the queue.
  - The tail of the list is the back of the queue.

```
typedef struct _node {
        char element;
        struct _node *next;
} node;

typedef struct {
        node *front;
        node *back;
} queue;

queue init ()
{
        return (queue){NULL,NULL};
}
```

Enqueue and Dequeue

```
queue enqueue ( queue Q, char ch )
{
        node *p;
        p = (node *)sizeof(node);
        p -> element = ch;
        p -> next = NULL;
        if (Q.back == NULL) {
                Q.front = Q.back = p;
        } else {
                Q.back -> next = p;
                Q.back = p;
        }
        return Q;
}
```

```
queue dequeue ( queue Q )
{
        node *p;

        if (isEmpty(Q)) {
                printf("empty
queue\n");
                return Q;
        }

        p = Q.front;
        Q.front = Q.front -> next;
        if (Q.front == NULL)
                Q.back = NULL;
        free(p);
        return Q;
}
```

Exercise: Write the other ADT functions.

Practice exercises

1. Make an implementation of the queue ADT using dynamic and realloc()-able array.
2. A *double-ended queue* (deque) is a queue with the exception that it supports insertion and deletion at both the ends. Each insert/delete operation must specify the end at which the operation is to be performed. Implement initialization, insertion, and deletion functions on a double-ended queue using:
   a. Static arrays
   b. Dynamic arrays
   c. Linked lists
3. A dictionary of integers is a set (an unordered collection) of integers that supports initialization, insertion, deletion, and searching. Implement the dictionary ADT using:
   a. Unsorted arrays
   b. Unsorted linked lists
   c. Sorted arrays
   d. Sorted linked lists
4. Suppose that in the dynamic-array implementation of the stack ADT, we want to ensure that the element array is never less than half full. The init function (no input size) would set element to a NULL pointer. Rewrite the ADT operations so that the never-less-than-half-full condition is always maintained.

Practice exercises

5. Let w be an alphanumeric string.
   a. Use the stack ADT calls to check whether a string is of the form $w\#w^r$ for some w, where w^r is the reverse of w.
   b. Use the queue ADT calls to check whether a string is of the form w#w for some w.
6. A (univariate) polynomial with integer coefficients is an ADT that supports initialization (from an array), check for zero, and adding, subtraction, and multiplication of two polynomials. Note that in order to store a polynomial, it suffices to know its degree d, and the coefficients of 1, x, x^2, x^3, . . . , x^d. Implement the polynomial ADT using:
   a. Arrays
   b. Linked lists
7. A sparse polynomial consists of a very few non-zero terms. We store a sparse polynomial as a list of (degree, coefficient) pairs for the non-zero terms. The operations to be supported are initialization (from an array of pairs), addition, subtraction, and multiplication. Implement the sparse polynomial ADT using:
   a. Arrays
   b. Linked lists