5. Abstract Data Type (ADT) Stack, Queue

- What is an abstract data type (ADT)?
- Stack as an ADT, its data structure, operations, and error conditions.
- Stack implementations and applications.
- Queue as an ADT, its data structure, operations, and error conditions.
- Queue implementations and applications.

Abstract Data Types (ADTs)

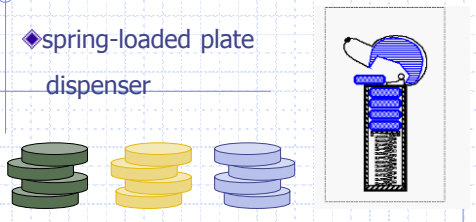
- An abstract data type (ADT) is an abstraction of a data structure.
- ADT refers to a way of packaging some intermediate-level data structures and their operations into a useful collection whose properties have been carefully studied.
- An ADT has a clean and simple interface.
- An ADT specifies:
  - Data stored
  - Operations on the data (clean, simple interface)
  - Error conditions associated with operations

An ADT Example

- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

Stacks

- spring-loaded plate dispenser



The Stack ADT

- The Stack ADT stores arbitrary elements
- Insertions and deletions follow the last-in first-out scheme
- Main stack operations:
  - push(element): inserts an element
  - element pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - element top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Applications of Stacks

- Direct applications
  - Undo sequence in a text editor
  - Chain of function calls in any language runtime system
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Execution Stack in C

- The C runtime system keeps track of the chain of active functions with a stack
- When a function is called, the runtime system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function ends, its frame is popped from the stack and control is passed to the function on top of the stack

```
main() {
  int i = 5;
  foo(i);
  }
foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }
bar(int m) {
  …
  }
```

bar
PC = 1
m = 6

foo
PC = 3
j = 5
k = 6

main
PC = 2
i = 5

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

```
function size()
  return top + 1;

function pop(S)
  if (isEmpty(S))
    error ("EmptyStack");
  else
    top = top – 1;
    return S[top + 1];
```

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then send FullStack error message
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

```
function push(S, val)
  if (isFull(S))
    error("FullStack");
  else {
    top = top + 1;
    S[top] = val;
  }
```

Array-based Stack

```
typedef struct {
   int array[MAX];
   int top;
} stack;

    void push(stack *s, int val){
        if (s->top >= MAX-1) error("Stack is full");
        s->top++;
        s->array[s->top] = val;
    }

    void pop(stack *s, int *val){
        if (s->top < 0) error("Stack is empty");
        *val = s->array[s->top];
        s->top--;
    }
```


Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific error

Stack ---- behavior

- A stack is defined by how it is used, not by its underlying structure.
- We can implement a stack by different data structures.
  - linked lists
  - arrays
  - what else?
- The only requirement for a stack is the ability to store elements in order of insertion, so that we can get the **LIFO** behavior.

List-based Stack:

- Linked list can be used to implement stack data structure: add and remove node from the "top".
  - Property: Last In First Out (LIFO).
  - Operations: is_empty, push, pop

An example of stack application:

$\pi * 10^{128} + e * 10^{128}$ Each operand has 128 digits

Read an operand(from left to right): 3141592653.....
27182818.....

Calculate the sum(from right to left): xx...xxxxx
+ yy...yyyyy

= ss...sssss

We need three stacks where two for operands and one for the sum.

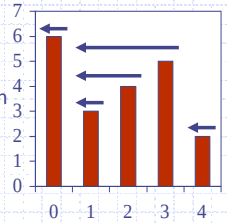

```
typedef struct node  node;
struct node{
    int n;
    node* next;
};
#define empty(s)  (!(s))

void push(node** top, int n){
    node* new = malloc(sizeof(node));        // create a new node
    if (!new) exit(-1);
    new->n = n;
    new->next = *top;
    *top = new;                              // set up stack top
}

int pop(node** top){                         // return a value (not a node)
    int n;
    node* temp;
    if (empty(*top)) return 0;
    temp = *top;                             // save top node
    *top = temp->next;                       // set up stack top
    n = temp->n;
    free(temp);
    return n;
}
```



```
node* get_operand(){ // read an operand and store on a stack
    node* s = NULL;
    int n;
    while (1){
        n = getchar();
        if (n < '0' || n > '9') return s;      // return if input char is not a digit
        push(&s, n – '0');
    }
}

main(){
    node *a, *b, *sum = NULL;
    int sumdig, carry = 0;
    printf("input two operands\n");
    a = get_operand();
    b = get_operand();
    while (!empty(a) || !empty(b)){            // perform addition
        sumdig = pop(&a) + pop(&b) + carry;
        push(&sum, sumdig%10);
        carry = sumdig/10;
    }
    if (carry != 0) push(&sum, carry);
    printf("the sum is:");
    while (!empty(sum)) printf("%c", pop(&sum));   // output result
}
```


Example: Computing Spans

- We show how to use a stack as an auxiliary data structure in an algorithm
- Given an an array X, the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ immediately preceding $X[i]$ and such that $X[j] \le X[i]$
- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

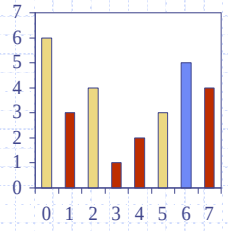
Quadratic Algorithm

```
function spans1(X, n)
    Input: X[n] integers
    Output: array S[n] of spans of X                    #
    int s;
    for (int i = 0; i < n; i++){                        n
        s = 1;                                          n
        while (s ≤ i && X[i – s] ≤ X[i]){               1 + 2 + …+ (n – 1)
            s = s + 1;                                  1 + 2 + …+ (n – 1)
        }
        S[i] = s;                                       n
    }
    return S                                            1
}
```

- Algorithm *spans1* runs in $O(n^2)$ time

Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
  - Let i be the current index
  - We pop indices from the stack until we find index j such that $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push i onto the stack



Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly once
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times
- Algorithm *spans2* runs in $O(n)$ time

```
function spans2(X, n)                          #
Input and output: same as span1
    stack A;
    for (i =0; i < n; i++){                    n
        while (!A.isEmpty() &&
                X[A.top()] ≤ X[i] ) {          n
            A.pop();                           n
        }
        if (A.isEmpty())                       n
            S[i] = i + 1;                      n
        else
            S[i] = i – A.top();                n
        A.push(i);                             n
    }
    return S                                   1
```


Queues

The Queue ADT

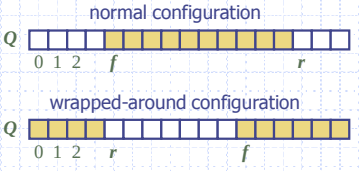
- The Queue ADT stores arbitrary elements
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(element): inserts an element at the end of the queue
  - element dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - element front(): returns the element at the front without removing it
  - int size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Errors
  - Attempting the execution of dequeue or front on an empty queue, or enqueue on a full queue(?)

Applications of Queues

- Direct applications
  - Waiting lists
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Queue

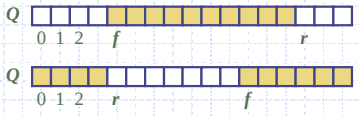
- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty



Queue Operations

- We use the modulo operator (remainder of division)

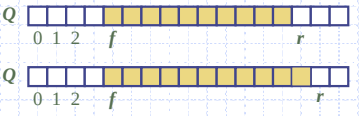
```
function size()
  return (N – f + r) mod N

function isEmpty()
  return (f = r)
```



Queue Operations (cont.)

- Operation enqueue reports an error if the array is full
- This error is implementation-dependent

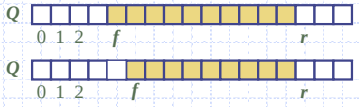
```
function enqueue(val)
  if (size() == N – 1)
    error("Full Queue");
  else{
    Q[r] = val;
    r = (r + 1) mod N;
  }
```



Queue Operations (cont.)

- Operation dequeue reports an error if the queue is empty
- This error is specified in the queue ADT

```
function dequeue()
  if (isEmpty()) then
    error("Empty Queue");
  else{
    val = Q[f];
    f = (f + 1) mod N;
    return val;
  }
```



Array-based Queue Declarations

```
#define MAX 1000

typedef struct {
        int count;
        int front;
        int rear;
        int data[MAX];
} queue;

void createQueue(queue *q)
{
   q->front  = 0;
   q->rear   = 0;
   q->count  = 0;
};
```


Circular-Array-based: Enqueue

```
int queueFull(queue *q)
{
   return q->count >= MAX;
}

void enqueue(int x, queue *q)
{
    if (queueFull(q))
        error("QUEUE IS FULL");
    q->count++;
    q->data[q->rear] = x;
    /* Move to next open position */
    q->rear = (q->rear + 1) % MAX;
}
```


Circular-Array-based: Dequeue

```
void dequeue(int *x,  queue *q)
{
    if (queueEmpty(q))
        error("QUEUE IS EMPTY");
    q->count--;
    *x = q->data[q->front]; /* data from front */
    /* Move to the next slot to dequeue */
    q->front = (q->front + 1)  % MAX;
}
```