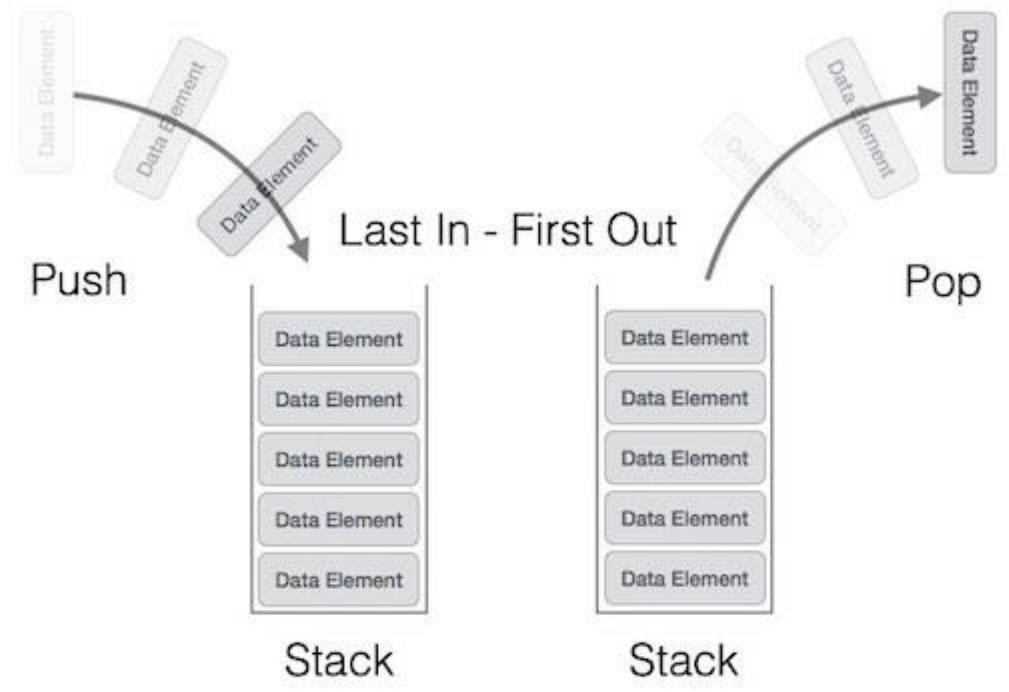
Data Structures and Algorithms

Trong-Hop Do
University of Information Technology, HCM city

Stack vs Queue

- Stack is a linear data structure that are inserted and removed according to the last-in first-out (LIFO) principle.
- Queue is a linear data structure that are inserted and removed according to the first-in first-out (FIFO) principle.



Stack vs Queue applications

- Stack
  - Expression Handling
    - Infix to Postfix or Infix to Prefix Conversion
    - Postfix or Prefix Evaluation
  - Backtracking Procedure
  - Restore the recursive function and its argument(s).
- Queue
  - Resource scheduling
  - Asynchronous data transfer (IO Buffers, pipes, file IO)

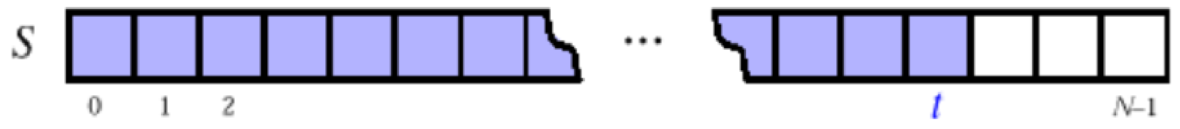
Operation on stack

- Push(o): pushing (storing) an element on the stack.
- Pop(): removing (accessing) an element from the stack.
- isEmpty(): check if Stack is empty
- isFull(): check if Stack is full
- Top(): get the top data element of the stack, without removing it.

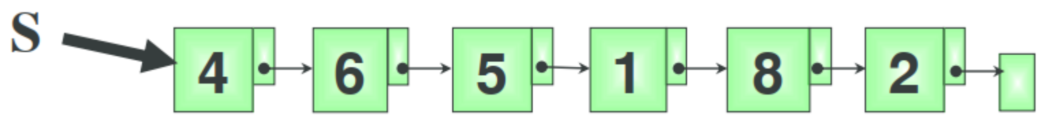
Implementing Stack

- Using array

```
Data    S [N];
int     t;
```



- Using linked list



List S

Implementing Stack using array

- Stack data structure

```
struct Stack {
       int a[MAX];
       int t;
};
```

- Create empty stack

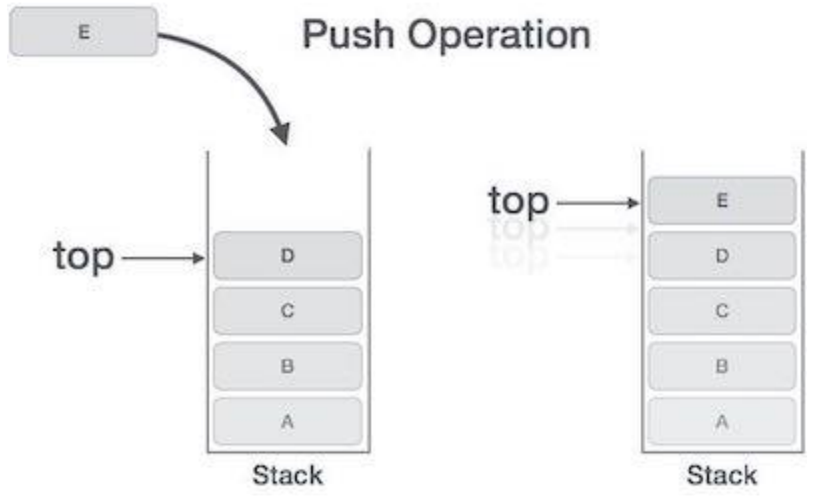
```
void CreateStack(Stack &s) {
       s.t = -1;
}
```

Check if Stack is empty or full

```
bool isEmpty(Stack s) { //check if Stack is empty
      if (s.t == -1)
      return 1;
      return 0;
}
bool isFull(Stack s) { //Check if Stack is full
      if (s.t >= MAX)
      return 1;
      return 0;
}
```

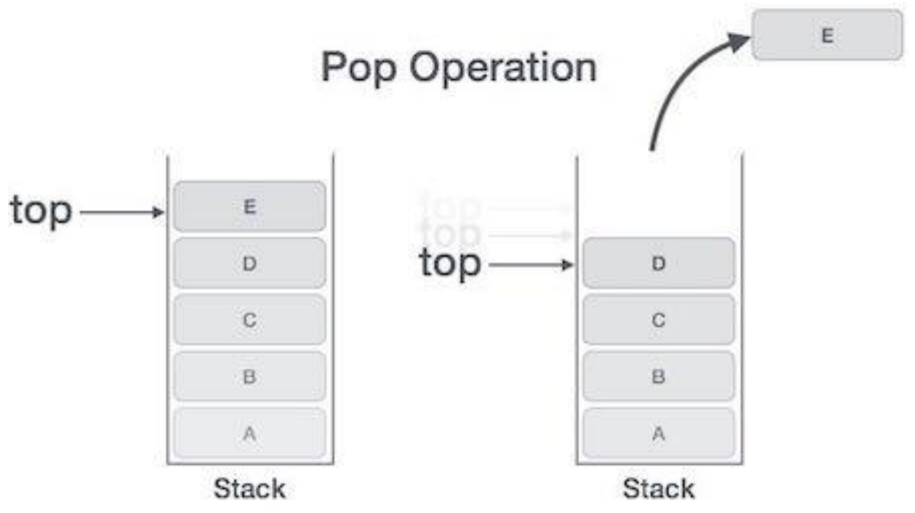
Pushing (storing) an element on the stack.

```
bool Push(Stack &s, int x) {
        if (isFull(s) == 0)
                s.a[++s.t] = x;
                return 1;
        }
        return 0;
}
```



Get the top data element of the stack, without removing it

```
int Pop(Stack &s, int &x) {
        if (isEmpty(s) == 0) {
                x = s.a[s.t--];
                return 1;
        }
        return 0;
}
```



Stack implementation using Linked List

```
typedef struct tagNode {
    int data;
    struct Node* next;
}Node;

typedef struct tagList {
        Node *pHead;
        Node *pTail;
}List;
```

Check empty

```
int isEmpty(LIST &s) {
       if (s.pHead == NULL)//Stack is empty
       return 1;
       return 0;
}
```

Pushing (storing) an element on the stack.

```
void Push(LIST &s, NODE *p) { // AddHead
        if (s.pHead == NULL) {
                s.pHead = p;
                s.pTail = p;
        }
        else {
                p->pNext = s.pHead;
                s.pHead = p;
        }
}
```

Removing (accessing) an element from the stack

```
bool Pop(LIST &s, int &x) {
       NODE *p;
       if (isEmpty(s) != 1) {
              if (s.pHead != NULL) {
                     p = s.pHead;
                     x = p->info;
                     s.pHead = s.pHead->pNext;
                     if (s.pHead == NULL)
                            s.pTail = NULL;
                     return 1;
              }
       }
       return 0;
}
```

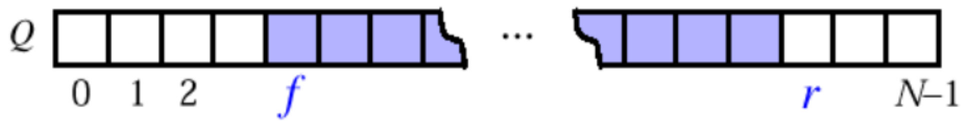
Operation on Queue

- EnQueue(O): add (store) an item to the queue
- DeQueue(): remove (access) an item from the queue
- isEmpty(): check if Queue is empty
- IsFull(): check if Queue is full
- Front(): Gets the element at the front of the queue without removing it

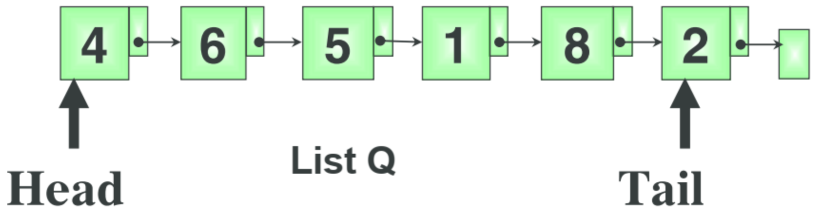
Queue implementation

- Using array

```
Data   Q[N];
int    font, rear;
```



- Using Linked list



Queue implementation using array

- Queue data structure:

```
typedef struct tagQueue {
        int a[MAX];
        int Front; //index of the first element in Queue
        int Rear; //index of the last element in Queue
} Queue;
```

- Create an empty Queue

```
void CreateQueue(Queue &q) {
        q.Front = -1;
        q.Rear = -1;
}
```

Check if Queue is empty or full

```
bool isEmpty(Queue q) { // Queue is empty
      if (q.Front == -1)
            return 1;
      return 0;
}
bool isFull(Queue q) { // Queue is full
      if (q.Rear - q.Front + 1 == MAX)
            return 1;
      return 0;
}
```

Add (store) an item to the queue

```
void EnQueue(Queue &q, int x) {
        int f, r;
        if (isFull(q)) //Queue is full
                printf("queue is full");
        else {
                if (q.Front == -1) {
                        q.Front = 0;
                        q.Rear = -1;
                }
                if (q.Rear == MAX - 1) {//Queue is not full but no empty space at Rear
                        f = q.Front;
                        r = q.Rear;
                        for (int i = f; i <= r; i++)
                                q.a[i - f] = q.a[i];
                        q.Front = 0;
                        q.Rear = r - f;
                }
                q.Rear++;
                q.a[q.Rear] = x;
        }
}
```

Remove (access) an item from the queue

```
bool DeQueue(Queue &q, int &x) {
       if (isEmpty(q)==0) { //Queue is not empty
              x = q.a[q.Front];
              q.Front++;
              if (q.Front>q.Rear) { //Queue has only one element
                     q.Front = -1;
                     q.Rear = -1;
              }
              return 1;
       }
       printf("Queue is empty");
       return 0;
}
```

Queue implementation using linked list

- Check if Queue is empty

```
bool isEmpty(LIST &Q) {
	if (Q.pHead == NULL) //Queue is empty
		return 1;
	return 0;
}
```

Add (store) an item to the queue

```
void EnQueue(LIST &Q, NODE *p) {
	if (Q.pHead == NULL) {
		Q.pHead = p;
		Q.pTail = p;
	}
	else {
		Q.pTail->pNext = p;
		Q.pTail = p;
	}
}
```

Remove (access) an item from the queue

```
int DeQueue(LIST &Q, int &x) {
        NODE *p;
        if (isEmpty(Q) != 1) {
                if (Q.pHead != NULL) {
                        p = Q.pHead;
                        x = p->info;
                        Q.pHead = Q.pHead->pNext;
                        if (Q.pHead == NULL)
                                Q.pTail = NULL;
                        return 1;
                }
        }
        return 0;
}
```

STACKS	QUEUES
Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list.	Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
Insert operation is called push operation.	Insert operation is called enqueue operation.
Delete operation is called pop operation.	Delete operation is called dequeue operation.
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.
Stack is used in solving problems works on recursion.	Queue is used in solving problems having sequential processing.

Reverse a string (or array) using stack

1) Create an empty stack.

2) One by one push all element of string (or array) to stack.

3) One by one pop all element from stack and put them back to string (or array).

Conversion of infix to postfix expression

- Infix vs postfix
  - Infix expression: a+b*c
  - Postfix expression: abc*+
- Example of infix to postfix conversion
  - Input: str = “a+b*c-(d/e+f*g*h)” Output: abc*+de/fg*h*+-
  - Input: a+b*c Output: abc*+

Conversion of infix to postfix expression

- Scan the infix expression from left to right.
- If the scanned character is an operand, output it.
- Else,
  - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack(or the stack is empty or the stack contains a '('), then push it.
  - '^' operator is right associative and other operators like '+','-','*' and '/' are left-associative. Check especially for a condition when both, operator at the top of the stack and the scanned operator are '^'. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack. In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
  - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
- If the scanned character is an '(', push it to the stack.
- If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
- Repeat steps 2-6 until the infix expression is scanned.
- Print the output
- Pop and output from the stack until it is not empty.

Conversion of infix to postfix expression

```
void infixToPostfix(string s){
    stack<char> st;
    string result;
    for (int i = 0; i < s.length(); i++) {
        char c = s[i];
        // If the scanned character is an operand, add it to output string.
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
            result += c;
        // If the scanned character is an '(', push it to the stack.
        else if (c == '(')
            st.push('(');
        // If the scanned character is an ')',pop and to output string from the
stack until an '(' is encountered.
        else if (c == ')') {
            while (st.top() != '(') {
                result += st.top();
                st.pop();
            }
            st.pop();
        }
    // If an operator is scanned
        else {
            while (!st.empty()  && prec(s[i]) <= prec(st.top())) {
                result += st.top();
                st.pop();
            }
            st.push(c);
        }
    }
    // Pop all the remaining elements from the stack
    while (!st.empty()) {
        result += st.top();
        st.pop();
    }
    cout << result << endl;
}
```

Conversion of Infix To Prefix Expression

- Example
  - Input : A * B + C / D Output : + * A B/ C D
  - Input : (A - B/C) * (A/K-L) Output : *-A/BC-/AKL
- Algorithm
  - Step 1: Reverse the infix expression i.e A+B*C will become C*B+A. Note while reversing each '(' will become ')' and each ')' becomes '('.
  - Step 2: Obtain the postfix expression of the modified expression i.e CB*A+.
  - Step 3: Reverse the postfix expression. Hence in our example prefix is +A*BC.

Evaluation of Postfix Expression

1) Create a stack to store operands (or values).

2) Scan the given expression and do following for every scanned element.

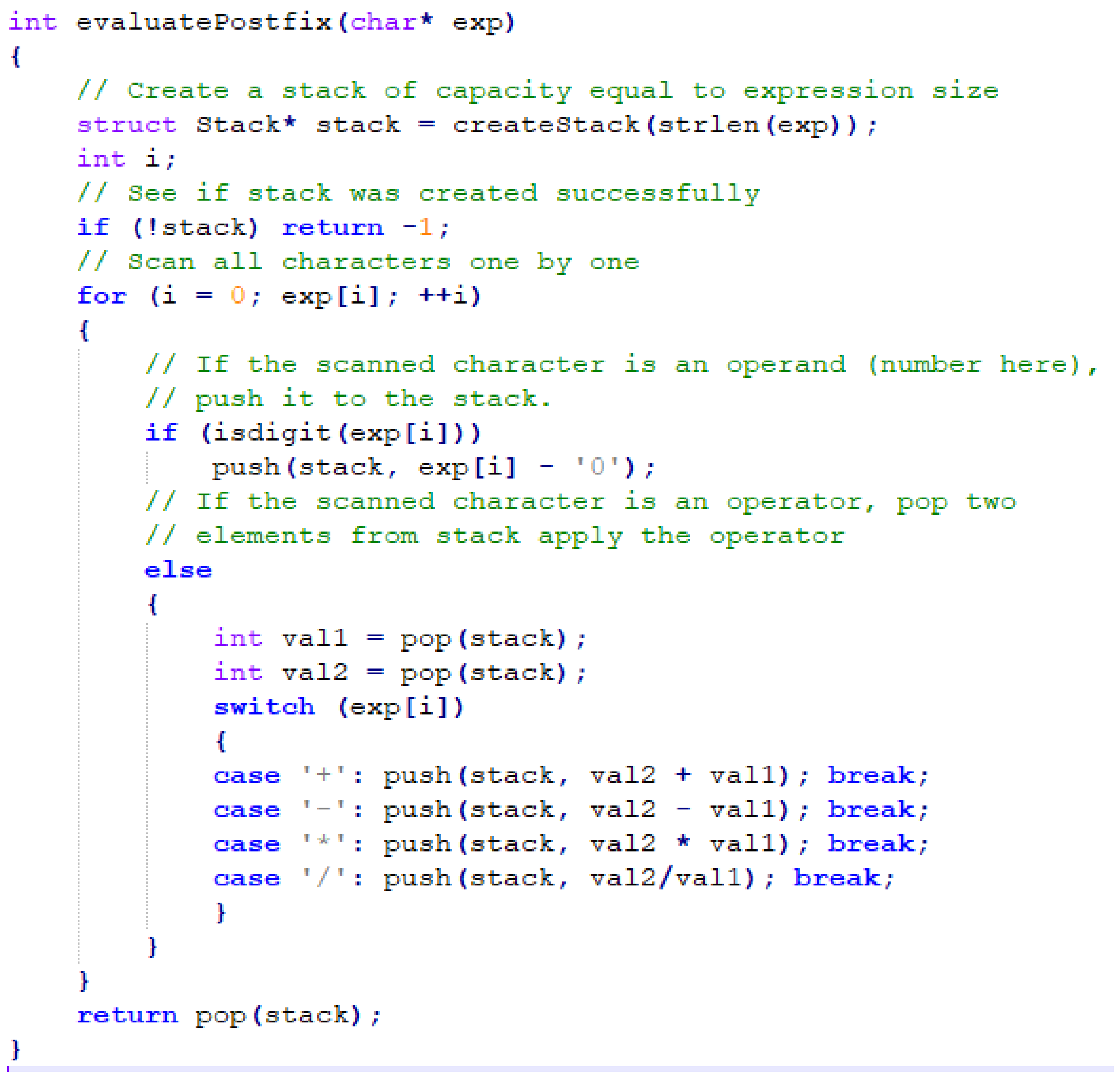
…..a) If the element is a number, push it into the stack

…..b) If the element is a operator, pop operands for the operator from stack.

Evaluate the operator and push the result back to the stack

3) When the expression is ended, the number in the stack is the final answer

Evaluation of Postfix Expression



```c
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;
    // See if stack was created successfully
    if (!stack) return -1;
    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        // If the scanned character is an operand (number here),
        // push it to the stack.
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');
        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);
            switch (exp[i])
            {
            case '+': push(stack, val2 + val1); break;
            case '-': push(stack, val2 - val1); break;
            case '*': push(stack, val2 * val1); break;
            case '/': push(stack, val2/val1); break;
            }
        }
    }
    return pop(stack);
}
```

Applications of Queues

- Scheduler (e.g., in an operating system) for controlling access to shared resources
  - maintains queues of printers' jobs
  - maintains queues of disk input/output requests
- Simulation of real word situations
  - use queues to simulate waiting queues in real scenarios (e.g. find out a good number of tellers, by simulating typical customer transactions)
- Telephone operators
  - queuing system for handling calls to toll-free numbers.

Demerging algorithm using Queue

- Consider a file of person records, each of which contains a person’s name, gender, date-of-birth, etc. The records are sorted by date-of-birth.
- **Task**: Rearrange the records such that females precede males but they remain sorted by date-of-birth within each gender group.

Sorting algorithm ✗	**Demerging algorithm** ✓
Time complexity $O(n \log n)$	Time complexity $O(n)$

Demerging Algorithm

Rearrange a file of person records such that females precede males but their order is otherwise unchanged.

```
1. Make females queue and males queue.
2. For each record in the input file, repeat:
       Let p be the next person read from the file.
       If p is female, adds p to the females queue
       If p is male, adds p to the males queue
3. While females is not empty, repeat:
       Remove the person from the females queue and
       write it out to a file
4. While males is not empty, repeat:
       Remove the person from the males queue and
       write it out to the same file
5. Output the file.
```