

Unit 2.1 Stack, Queue and Trees

Algorithms

EE/NTHU

Mar. 10, 2021

Stacks

- A **stack** is a linear list that can store elements to be fetched later, and the element fetched from the stack is the last one stored.
 - **Last In First Out** (LIFO).
- Stack can be implemented using a simple array and an integer that represents the top position.
- Assume the array is $stack[1 : n]$ with n elements and the stack index is top , which is initialized to 0.
- The following algorithm inserts an element into the stack.

Algorithm 2.1.1. Stack Push – Array

```
// Push an element into the stack.
// Input: item to be inserted
// Output: none.
1 Algorithm StkPush(item)
2 {
3     if ( $top \geq n$ ) then error (" Stack is full!");
4     else {
5          $top := top + 1$ ;
6          $stack[top] := item$ ; // Store item.
7     }
8 }
```

Stack — Pop

- To fetch an item from the stack.

Algorithm 2.1.2. Stack Pop – Array

```
// Pop the top element from the stack and return its value.
// Input: none
// Output: item on top of the stack.
1 Algorithm StkPop()
2 {
3     if (top < 1) then error (" Stack is empty!");
4     else {
5         item := stack[top];
6         top := top - 1;
7         return item ;
8     }
9 }
```

- Both `StkPush` and `StkPop` algorithms have the time complexity of $\mathcal{O}(1)$
 - It is independent of the size of the stack, n .
 - And also independent of the number of items stored, top .

Stack — Status Check

- Two functions are useful to check the status of the stack.

Algorithm 2.1.3. Stack Empty Check

```
// Check if the stack is empty.
// Input: none
// Output: true if stack empty otherwise false.
1 Algorithm StkEmpty()
2 {
3     if (top = 0) then return true ;
4     else return false ;
5 }
```

Algorithm 2.1.4. Stack Full Check

```
// Check if the stack is Full.
// Input: none
// Output: true if stack full otherwise false.
1 Algorithm StkFull()
2 {
3     if (top = n) then return true ;
4     else return false ;
5 }
```

Stack — Dynamically Allocated Array

- The array *stack* can be either a static array or a dynamically allocated array.
- Using static array, then the number of items to be stored is limited by the size, n , of the array.
- Using a dynamically allocated array, the array size, n , can be enlarged and then employ the `realloc` function to adjust the stack space.
 - This is more flexible to handle problems in different sizes.
- Stack can also be implemented using **linked list**
- Assuming **NODE** is a **structure** defined as

```
struct NODE {  
    TYPE data;           // for data storage  
    struct NODE *link;   // pointer to the next node  
}
```

- NODE pointer *LStack* is now the linked list to store the items.
 - *LStack* is initialized to **NULL**.
- The variable *top* is no longer needed.

Stacks in Linked List

Algorithm 2.1.5. Stack Push – Linked List

```
// Push an element into the stack, LStack.  
// Input: item to be inserted  
// Output: none.  
1 Algorithm LStkPush(item)  
2 {  
3     temp := new NODE;  
4     temp → data := item; temp → link := LStack;  
5     LStack := temp;  
6 }
```

Algorithm 2.1.6. Stack Pop – Linked List

```
// Pop the top element from the stack, LStack, and return its value.  
// Input: none  
// Output: item on top of the stack.  
1 Algorithm LStkPop()  
2 {  
3     if (LStack = NULL) then error (" Stack is empty!");  
4     else {  
5         item := LStack → data; temp := LStack; LStack := temp → link;  
6         free temp; return item;  
7     }  
8 }
```

- With enough computer resources, stack implemented using linked list should not have stack full issue.
 - Thus, no **StkFull** check is needed.
- Stack empty check is equivalent to checking if *LStack* is **NULL**.
- Again, either **LStkPush** or **LStkPop** algorithm is of $\mathcal{O}(1)$ time complexity.
 - Independent to stack size or the number of items stored.
- The space complexity of the array stack is $\Theta(n)$, where n is the size of the array.
- The space complexity of linked list stack is $\Theta(m)$, where m is the number of items stored.
- The linked list stack appears to be more memory efficient, since $m \leq n$.

Queue

- **Queue** is another linear list to store data, but the data fetched is the first one stored.
 - **First in First out** (FIFO).
- Queue can also be implemented using simple array.
- Assume the array is $Q[1 : n]$ with n elements.
 - Two integer variables: *head* for the front of the queue, and *tail* for the rear of the queue.
- The following algorithm stores an item onto the queue.

Algorithm 2.1.7. Enqueue.

```
// Insert the item into the queue.
// Input: item to be inserted
// Output: none.
1 Algorithm Enqueue(item)
2 {
3     tail := (tail + 1) mod n;
4     if (head = tail) then error (" Queue is full! ");
5     else {
6         Q[tail] := item;
7     }
8 }
```

Algorithm 2.1.8. Queue Empty.

```
// Check if the queue is empty or not.
// Input: none
// Output: true if queue is empty otherwise false.
1 Algorithm EmptyQ()
2 {
3     if (head = tail) then return true ;
4     else return false ;
5 }
```

Algorithm 2.1.9. Dequeue.

```
// Retrieve the item from the queue.
// Input: none
// Output: the first item of the queue.
1 Algorithm Dequeue()
2 {
3     if EmptyQ() then error (" Queue is empty! ");
4     else {
5         head := (head + 1) mod n ;
6         item := Q[head] ;
7         return item ;
8     }
9 }
```

Stack and Queue

- Time complexities of both `Enqueue()` and `Dequeue()` algorithms are $\mathcal{O}(1)$.
 - Space complexities are $\Theta(n)$, n is the size of the array Q .
- Queue also can be implemented using linked list
- Both stack and queue are useful data structures to store temporary data.
 - Storing and retrieving data are very efficient.
- Stack is Last In First Out
 - A simple array with an addition variable is sufficient.
- Queue is First In First Out
 - An simple array with two additional variables.
 - The array elements are used in a circular fashion.
 - Enlarging queue size is a little more complicated than stack.
- Both can also be implemented using linked lists.
 - Space utilization is more efficient.
 - Time complexity remains the same.

Celebrity Problem

- A group of n persons have been gathered. There might be a celebrity in the group such that everyone knows the celebrity while the celebrity knows no one. Is there a way to identify the celebrity quickly?
- The relationship of the persons of the group can be represented by an $n \times n$ matrix, A , such that if person i knows person j then $A[i, j] = 1$, otherwise $A[i, j] = 0$. For simplicity, $A[i, i] = 1$ is also assumed.
- If person k is the celebrity, then we have $A[i, k] = 1, 1 \leq i \leq n$, and $A[k, j] = 0, 1 \leq j \leq n$ and $j \neq k$.

$$A[i, k] = 1, \quad 1 \leq i \leq n, \quad (2.1)$$

$$A[k, j] = 0, \quad 1 \leq j \leq n \text{ and } j \neq k. \quad (2.2)$$

- The brute force approach is to check all $A[i, j], 1 \leq i, j \leq n$ against the equations (2.1) and (2.2).
- It is apparent the brute force approach is $\mathcal{O}(n^2)$.

Celebrity Problem, II

- An alternative to identifying the celebrity is

Algorithm 2.1.10. Celebrity Identification – Generic Algorithm

```
// Given  $n \times n$  matrix  $A$  find the celebrity satisfies Eqs (2.1) and (2.2).
// Input: Array  $A$ , int  $n$ ;
// Output: Celebrity  $k$ , or "None".
1 Algorithm Celebrity( $A, n$ )
2 {
3     Form a set  $S := \{1, 2, \dots, n\}$ ; //  $S$  initialized to  $n$  elements.
4     while  $|S| > 1$  do { //  $S$  has more than one element.
5         choose two elements  $u, v \in S$ ;
6         if  $A[u, v] = 1$  then  $S := S - \{u\}$ ; // Remove  $u$ .
7         else  $S := S - \{v\}$ ; // Remove  $v$ .
8     }
9     let  $k$  be the only element in  $S$ ; //  $k$  is the candidate for celebrity.
10    for  $i := 1$  to  $n$  do // Verify  $k$  is the celebrity.
11        if  $i \neq k$  then {
12            if  $A[i, k] \neq 1$  or  $A[k, i] \neq 0$  then return "None";
13        }
14    return  $k$ ;
15 }
```


Celebrity Problem, III

- In Algorithm (2.1.10) line 6, $A[u, v]$ is checked.
 - If $A[u, v] = 1$ then u cannot be the celebrity therefore it is removed from set S ;
 - On the other hand, if $A[u, v] = 0$ then v is not the celebrity and is removed.
- Therefore, each iteration of the loop (lines 4–8) one element is removed from S .
 - After $n - 1$ iterations, one element is left and it should be a candidate for the celebrity.
 - The complexity is $\Theta(n)$.
- Lines 10–13 verify if the candidate is, indeed, the celebrity.
 - The complexity is $\Theta(n)$.
- Thus, the total complexity is $\Theta(n)$.
- In fact, matrix A is accessed $3(n - 1)$ times over all.

Celebrity Identification using Array

- Algorithm (2.1.10) can be implemented using array for S as

Algorithm 2.1.11. Celebrity Identification – Using Array

```
// Given  $n \times n$  matrix  $A$  find the celebrity satisfies Eqs (2.1) and (2.2).
// Input: Array  $A$ , int  $n$ ;
// Output: Celebrity  $k$ , or "None".
1 Algorithm Celebrity_A( $A, n$ )
2 {
3     for  $i := 1$  to  $n$  do  $S[i] := i$ ; // Initialize array  $S$ .
4      $u := 1$ ;  $v := n$ ;
5     while  $u < v$  do { //  $S$  has more than one element left.
6         if  $A[u, v] = 1$  then  $u := u + 1$ ; // Remove  $u$ .
7         else  $v := v - 1$ ; // Remove  $v$ .
8     }
9      $k := u$ ; //  $k$  is the candidate for celebrity.
10    for  $i := 1$  to  $n$  do // Verify  $k$  is the celebrity.
11        if  $i \neq k$  then {
12            if  $A[i, k] \neq 1$  or  $A[k, i] \neq 0$  then return "None";
13        }
14    return  $k$ ;
15 }
```

Celebrity Identification using Stack

- Algorithm (2.1.10) can be implemented using stack for S as

Algorithm 2.1.12. Celebrity Identification – Using Stack

```
// Given  $n \times n$  matrix  $A$  find the celebrity satisfies Eqs (2.1) and (2.2).
// Input: Array  $A$ , int  $n$ ;
// Output: Celebrity  $k$ , or "None".
1 Algorithm Celebrity_S( $A, n$ )
2 {
3     for  $i := 1$  to  $n$  do StkPush( $i$ ); // Initialize stack.
4     for  $i := 1$  to  $n - 1$  do // Repeat  $n - 1$  times
5          $u := \text{StkPop}()$ ;  $v := \text{StkPop}()$ ;
6         if  $A[u, v] = 1$  then StkPush( $v$ ); // Remove  $u$ .
7         else StkPush( $u$ ); // Remove  $v$ .
8     }
9      $k := \text{StkPop}()$ ; //  $k$  is the candidate for celebrity.
10    for  $i := 1$  to  $n$  do // Verify  $k$  is the celebrity.
11        if  $i \neq k$  then {
12            if  $A[i, k] \neq 1$  or  $A[k, i] \neq 0$  then return "None";
13        }
14    return  $k$ ;
15 }
```

Celebrity Identification using Queue

- Algorithm (2.1.10) can be implemented using queue for S as

Algorithm 2.1.13. Celebrity Identification – Using Queue

```
// Given  $n \times n$  matrix  $A$  find the celebrity satisfies Eqs (2.1) and (2.2).
// Input: Array  $A$ , int  $n$ ;
// Output: Celebrity  $k$ , or "None".
1 Algorithm Celebrity_Q( $A, n$ )
2 {
3     for  $i := 1$  to  $n$  do Enqueue( $i$ ); // Initialize stack.
4     for  $i := 1$  to  $n - 1$  do // Repeat  $n - 1$  times
5          $u := \text{Dequeue}()$ ;  $v := \text{Dequeue}()$ ;
6         if  $A[u, v] = 1$  then Enqueue( $v$ ); // Remove  $u$ .
7         else Enqueue( $u$ ); // Remove  $v$ .
8     }
9      $k := \text{Dequeue}()$ ; //  $k$  is the candidate for celebrity.
10    for  $i := 1$  to  $n$  do // Verify  $k$  is the celebrity.
11        if  $i \neq k$  then {
12            if  $A[i, k] \neq 1$  or  $A[k, i] \neq 0$  then return "None";
13        }
14    return  $k$ ;
15 }
```


- Algorithms [Celebrity_A \(2.1.11\)](#), [Celebrity_S \(2.1.12\)](#) and [Celebrity_Q \(2.1.13\)](#) implement Algorithm [Celebrity_G \(2.1.10\)](#) using different data structures for S .
- All of them have the same time complexity $\Theta(n)$.
- In Algorithm [Celebrity_A \(2.1.11\)](#) if u or v is the candidate, then it is not changed for the rest of the iterations.
- In Algorithm [Celebrity_S \(2.1.12\)](#) if the candidate has been popped from the stack, it also remains on top of the stack for the rest of the iterations.
- In Algorithm [Celebrity_Q \(2.1.13\)](#), however, the candidate is enqueued to the end of the queue.
 - The candidate is evaluated at most $\lfloor \lg n \rfloor$ times.

Summary

- Stacks and queues
 - Insert, delete and status check
 - Array and linked list representations
- Celebrity problem
 - With array
 - With stack
 - With queue