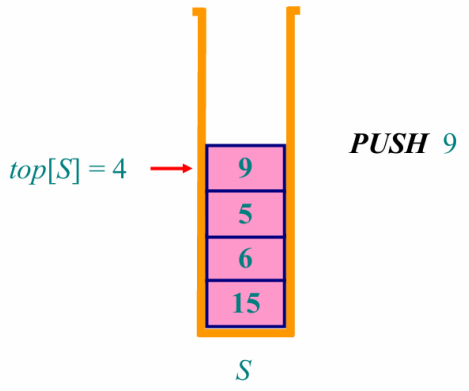
Data Structure Lecture 3: Basic Data Structures and Sorting

Part I. Basic Data Structures

1.Stack

First in, last out(FILO)



- Operation:
  - **Push**: push the new element into the end of the array
  - **Pop**: pop the first element of the end of the array
- Implementation:
  - **stack_pointer**: point to the **top of the stack**(last element's index + 1)

my_stack

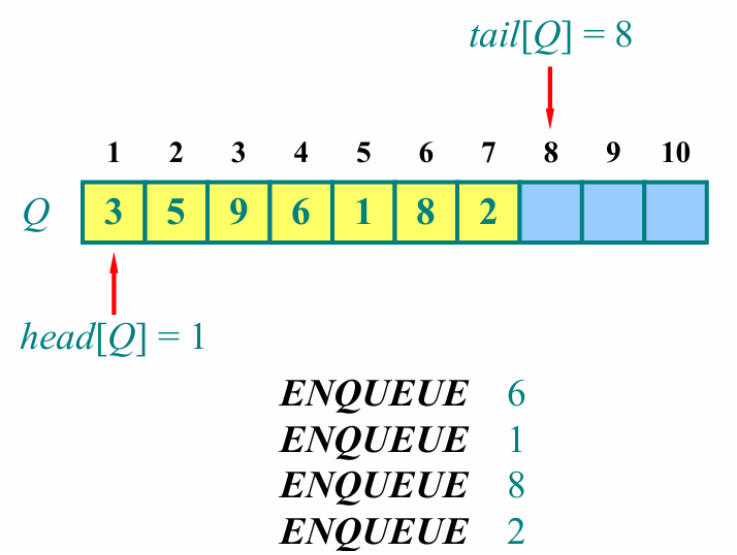
```
template<typename T>
class my_stack{
public:
    int max_num;
    int stack_pointer;
    vector<T> stack_arr;
    //Constructor
    my_stack(){
        stack_pointer = 0;
        max_num = 100;
        stack_arr = vector<T>(max_num);
    }
    my_stack(int num){
        stack_pointer = 0;
```

```
15            max_num = num;
16            stack_arr = vector<int>(max_num);
17        }
18
19        // Push Method
20        void push(T element){
21             if(stack_pointer >= max_num){
22                 cout << "stack overflow" << endl;
23                 return;
24             }
25             stack_arr[stack_pointer] = element;
26             stack_pointer++;
27        }
28
29        // Pop Method
30        T pop(){
31            if(stack_pointer <= 0){
32               cout << "stack downflow" << endl;
33               return NULL;
34            }
35            stack_pointer--;
36            return stack_arr[stack_pointer];
37        }
38    }
```

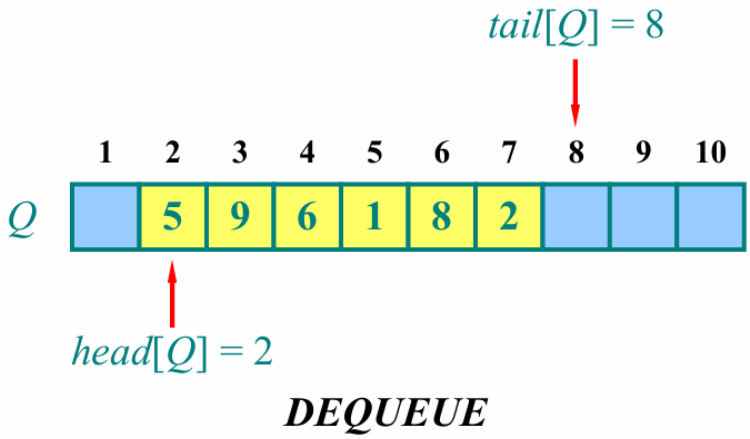
2.Queue

First in, first out(FIFO)

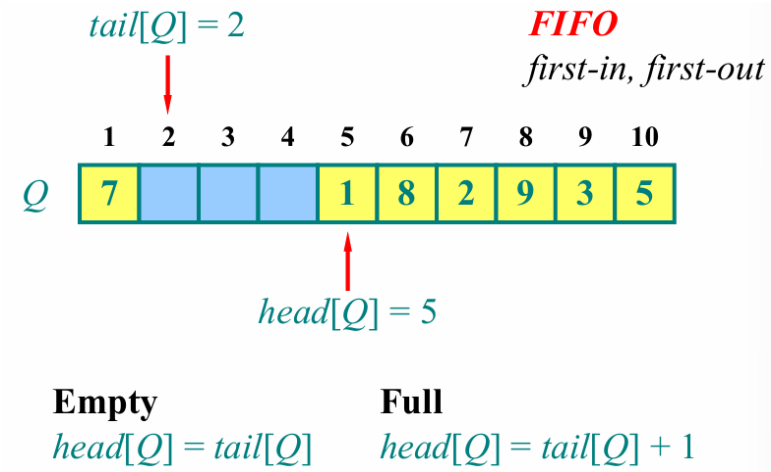
- Operation:
  - **Enqueue:** push the new element into the end of the array
  - **Dequeue:** pop the first element of the start of the array
- Implemenation:
  - **Head:** first element's index
  - **Tail:** Last element's index + 1
  - Idea : **Cyclical array**(Suppose the end of the array is connected to the start of the array)



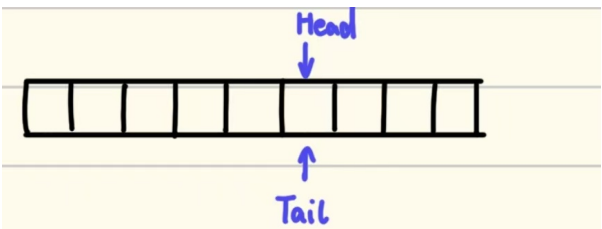
ENQUEUE 6
ENQUEUE 1
ENQUEUE 8
ENQUEUE 2

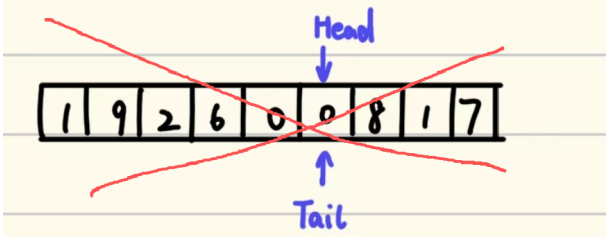


DEQUEUE

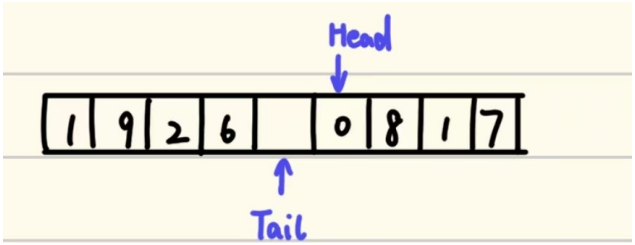


- `Head == Tail` : The queue is **empty**





- `Head == Tail + 1` : The queue is **full**(There remains one grid not filled)



> Understanding:
>
> Suppose that the `Head` is fixed, and we only remove the `Tail` .There are `n` possible results, but we want to reprent that the queue have $0, 1, 2, 3, ..., n$ elements, totally `n+1` states. This means there is one state that we cannot represent.We choose to sacrifice the state where the queue has `n` elements.

my_queue

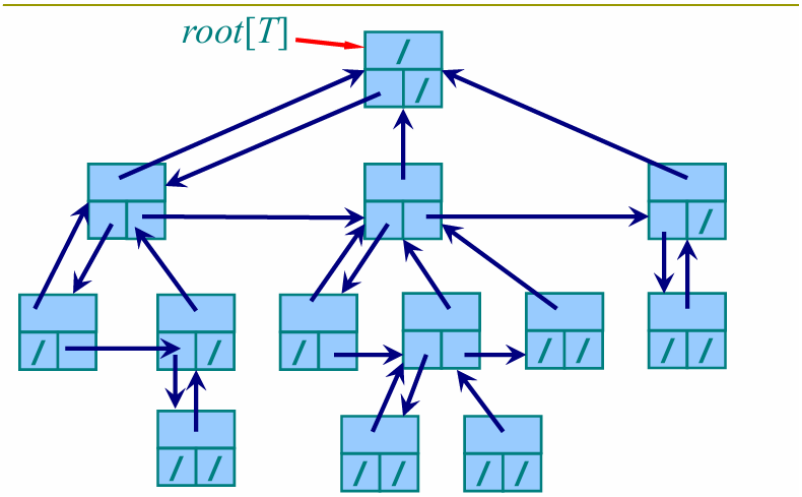
```
1   template<typename T>
2   class my_queue{
3   public:
4       int max_num;
5       int head;
6       int tail;
7       vector<T> queue_arr;
8       //Constructor
9       my_stack(){
10          head = 0;
11          tail = 0;
12          max_num = 100;
13          queue_arr = vector<T>(max_num);
14      }
15      my_stack(int num){
16          head = 0;
17          tail = 0;
18          max_num = num;
19          stack_arr = vector<int>(max_num);
20      }
21
22      // index up
23      int index_up(int index){
24          return index < max_num - 1 ? index + 1 : 0;
```

```
25        }
26
27        // Enqueue Method
28        void enqueue(T element){
29             if(tail == head - 1){
30                  cout << "queue overflow" << endl;
31                  return;
32             }
33             queue_arr[tail] = element;
34             tail = index_up(tail);
35        }
36
37        // Deque Method
38        T deque(){
39            if(head == tail){
40               cout << "queue downflow" << endl;
41               return NULL;
42            }
43            T ret = queue_arr(head);
44            head = index_up(head);
45            return ret;
46        }
47    }
```

3.Linked List

4.Tree

所有的树可以用”**儿子兄弟表示法**“转化为二叉树。一个节点的左子节点表示他的”长子“， 右子节点表示比他”最大的弟弟“。向左的边表示父子关系，向右的边表示兄弟关系。

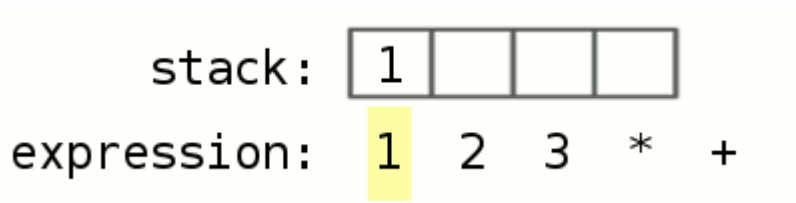


5.Graph

6.Postfix(reverse Polish notation)：Application of Stack

> Infix: `9 + 2 * 7 + (2 * 5 + 6) * 2`
>
> Postfix: `9 3 7 * + 4 5 * 6 + 2 * +`

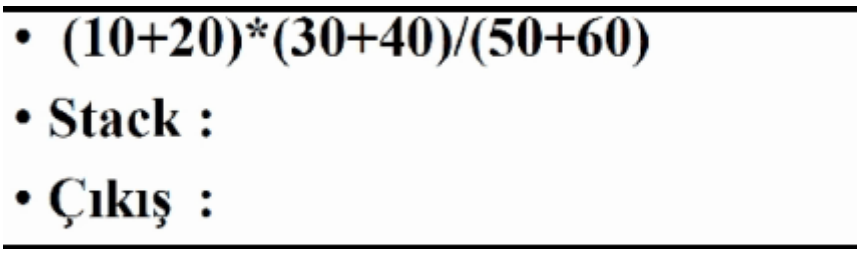
Postfix calculation:



📢 Rules:

1. 每遇到1个运算数，将这个数字放入stack
2. 遇到一个二元运算符，从stack里面弹出2个运算数，进行该运算以后，将结果放入堆栈
3. 当所有运算符都已经用掉，stack里面只有一个元素时，停止计算，这个元素就是结果。

Infix to prefix:



📢 Rules:

1. 遇到了算术运算符 `+`, `-`, `*`, `/`, `(` , `)` 的时候，尝试把运算符放入stack
   - 如果是 `(` ,直接放入
   - 如果是 `)` ,依次弹出它与stack里面最近的 `(` 之间的所有运算符。
   - 如果是 `+`, `-`, `*`, `/`
     - 如果stack的top元素运算符优先级高于它，弹出top元素以后，放入它
     - 如果stack的top元素运算符优先级较低或者是 `(` ，直接放入它。
2. 遇到一个运算数，把它放入后缀表达式
3. 当所有运算数都用掉以后，依次弹出stack里面的所有运算符

Part II. Sorting

📢 **Sorting Algorithms:**

- Sorting with Comparison:

  lower bound: $O(NlogN)$

  - Insertion Sort:

    $O(N^2)$

    Space Complexity: $O(1)$

  - Merge Sort:

    $O(NlogN)$

    Space Complexity: $O(N)$

  - Heap Sort:

    $O(NlogN)$

  - Quick Sort:

    Expected case: $\Theta(NlogN)$

    Worst case: $\Theta(N)$

    Space Complexity: $O(1)$

- Sorting without Comparison:

  - Radix Sort

1.Quick Sort

> Basic Idea of Quick Sort
>
> 1. Divide: **Partition** the array into two subarrays around a **pivot** `x` such that **elements in lower subarray** $\leq x$, **elements in upper subarrys** $\geq x$.
>
> 
>
>
> 2. Conquer: Recursively sort the two subarrays
>
>   Expected conquer cost: $O(n)$
>
> 3. Combine: Trivial(微不足道的)

1. Time complexity analysis:

   - Expected Case:

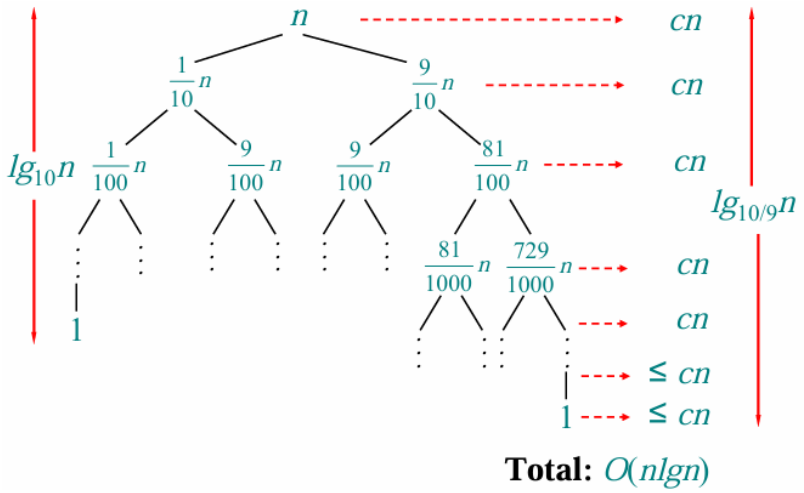
     We suppose we can get the ideal case, in which the pivot divides the array into exactly two identically half subarrays.

$$T(n) = 2T(\frac{N}{2}) + O(N)$$

We can use the master method to analyze the time complexity.

$$T(N) = Nlog(N)$$

Or we can suppose that pivot divides the array into exactly one subarray with `k` of the total elements and the other with `1-k` of the elements, $0 < k < 1$



We assume $\lambda = max\{k, 1-k\}$

Tree Height: $log_{\frac{1}{\lambda}} N$

Cost for each layer: cn

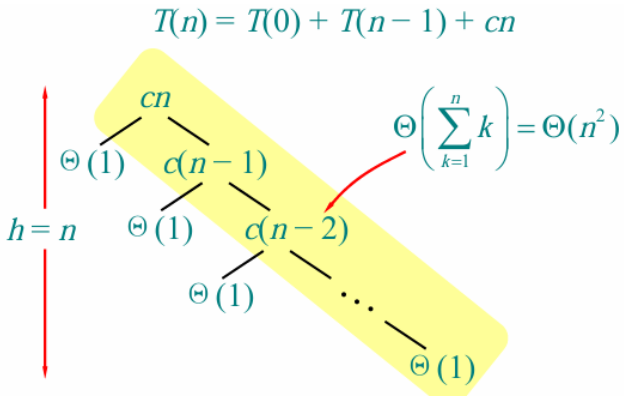
$$\Longrightarrow T(N) = \Theta(NlogN)$$

- Worst case:

We suppose we can get the worst case, in which the pivot divides the array into exactly one subarray with 0 element and the other with `n-1` element.

$$T(N) = T(0) + T(N-1) + \Theta(N)$$

$$T(N) = \Theta(1) + T(N-1) + \Theta(N) = \Theta(N^2)$$

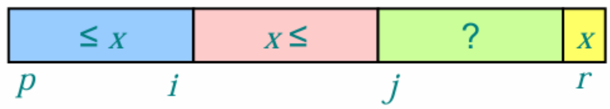


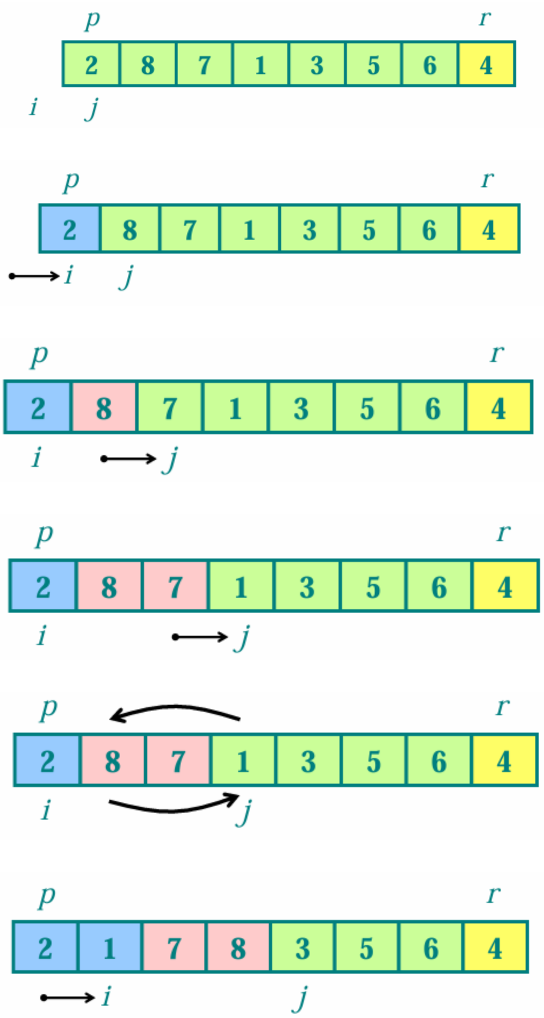
2. Implementation:

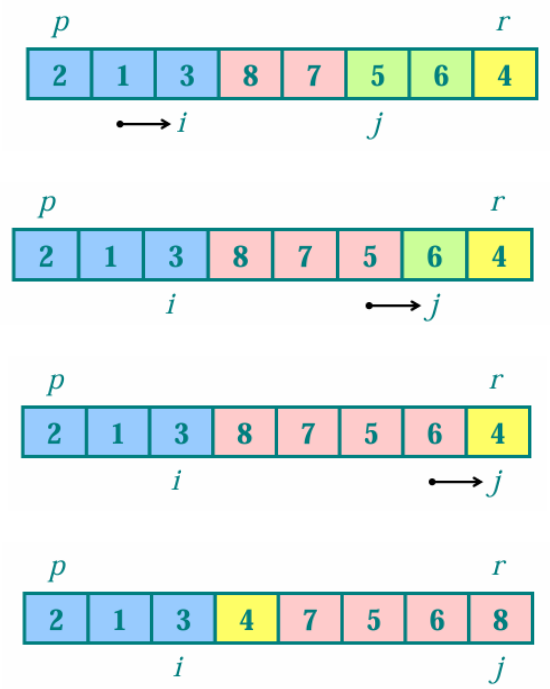
```
QUICKSORT(A, p, r)
if p < r
    then q ← PARTITION(A, p, r)
         QUICKSORT(A, p, q – 1)
         QUICKSORT(A, q + 1, r)
```

Initial call: QUICKSORT$(A, 1, n)$

```
PARTITION(A, p, r)    //A[p … r]
x ← A[r]    //pivot = A[p]
i ← p – 1
for j ← p to r – 1
    do if A[j] ≤ x
        then i ← i + 1
             exchange A[i] ↔ A[j]
exchange A[r] ↔ A[i + 1]
return i + 1
```







We can use the last element as pivot, implemented in the way which the pseudocode describes:

Quick Sort1

```
template<typename T>
void Quick_sort_1(std::vector<T>& arr){

    std::function<void(int, int)> helper = [&](int left, int right) -> void{
        // right is exclusive
        if(left >= right - 1){
            return;
        }

        // partition, choose arr[right - 1] to be the pivot
        T pivot = arr[right - 1];
        // i points to the last element we have explored that's smaller than the pivot
        int i = left - 1;
        // j points to the first element that we haven't explored.
        int j = left;

        while(j < right - 1){
            if(arr[j] > pivot){
                j++;
            }
            else{
                std::swap(arr[i + 1], arr[j]);
                i++;
                j++;
            }
        }
```

```
27
28          std::swap(arr[i + 1], arr[right - 1]);
29
30          helper(left, i + 1);
31          helper(i + 2, right);
32      };
33
34      helper(0, arr.size());
35  }
```

We can also use the first element as pivot, implemented in a slightly different way.

Quick Sort2

```
1   template<typename T>
2   void Quick_sort_2(std::vector<T>& arr){
3
4       std::function<void(int, int)> helper = [&](int left, int right) -> void{
5           // right is exclusive
6           if(left >= right - 1){
7               return;
8           }
9
10          // partition, choose arr[left] to be the pivot
11          T pivot = arr[left];
12
13          int small_point = left + 1;
14          int big_point = right - 1;
15          /**We can guarantee that:
16           * When index < small_point, elements must be smaller than the pivot
17           * When index > big_point, elements must be bigger than the pivot
18          */
19          while(small_point <= big_point){
20              if(arr[small_point] < pivot){
21                  small_point++;
22              }
23              else{
24                  std::swap(arr[small_point], arr[big_point]);
25                  big_point--;
26              }
27          }
28
29          std::swap(arr[left], arr[small_point - 1]);
30
31          helper(left, small_point - 1);
32          helper(small_point, right);
```

```
        };

        helper(0, arr.size());
    }
```

Average case: no human effect(not $nlgn$ exactly)

Expected case: with human effect