Contents

Chapter 1

Applications of Queue Data Structure

Queue is used when things don't have to be processed immediatly, but have to be processed in **F**irst **I**n **F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

See thisfor more detailed applications of Queue and Stack.

References:
http://introcs.cs.princeton.edu/43stack/

Source

http://www.geeksforgeeks.org/applications-of-queue-data-structure/

Category: Misc

Chapter 2

Implement Queue using Stacks

A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1 (By making enQueue operation costly)
This method makes sure that newly entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

```
enQueue(q, x)
  1) While stack1 is not empty, push everything from satck1 to stack2.
  2) Push x to stack1 (assuming size of stacks is unlimited).
  3) Push everything back to stack1.

dnQueue(q)
  1) If stack1 is empty then error
  2) Pop an item from stack1 and return it
```

Method 2 (By making deQueue operation costly)
In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

```
enQueue(q,  x)
  1) Push x to stack1 (assuming size of stacks is unlimited).

deQueue(q)
  1) If both stacks are empty then error.
  2) If stack2 is empty
       While stack1 is not empty, push everything from satck1 to stack2.
  3) Pop the element from stack2 and return it.
```

Method 2 is definitely better than method 1. Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty.

Implementation of method 2:

```
/* Program to implement a queue using two stacks */
#include<stdio.h>
#include<stdlib.h>

/* structure of a stack node */
struct sNode
{
   int data;
   struct sNode *next;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* structure of queue having two stacks */
struct queue
{
   struct sNode *stack1;
   struct sNode *stack2;
};

/* Function to enqueue an item to queue */
void enQueue(struct queue *q, int x)
{
   push(&q->stack1, x);
}

/* Function to dequeue an item from queue */
int deQueue(struct queue *q)
{
   int x;

   /* If both stacks are empty then error */
   if(q->stack1 == NULL && q->stack2 == NULL)
   {
      printf("Q is empty");
      getchar();
      exit(0);
   }

   /* Move elements from satck1 to stack 2 only if
       stack2 is empty */
   if(q->stack2 == NULL)
   {
     while(q->stack1 != NULL)
     {
        x = pop(&q->stack1);
        push(&q->stack2, x);
     }
   }
```

```
   x = pop(&q->stack2);
   return x;
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
  /* allocate node */
  struct sNode* new_node =
            (struct sNode*) malloc(sizeof(struct sNode));

  if(new_node == NULL)
  {
     printf("Stack overflow \n");
     getchar();
     exit(0);
  }

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*top_ref);

  /* move the head to point to the new node */
  (*top_ref)    = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
  int res;
  struct sNode *top;

  /*If stack is empty then error */
  if(*top_ref == NULL)
  {
     printf("Stack overflow \n");
     getchar();
     exit(0);
  }
  else
  {
     top = *top_ref;
     res = top->data;
     *top_ref = top->next;
     free(top);
     return res;
  }
}

/* Driver function to test anove functions */
int main()
```

```
{
   /* Create a queue with items 1 2 3*/
   struct queue *q = (struct queue*)malloc(sizeof(struct queue));
   q->stack1 = NULL;
   q->stack2 = NULL;
   enQueue(q, 1);
   enQueue(q, 2);
   enQueue(q, 3);

   /* Dequeue items */
   printf("%d  ", deQueue(q));
   printf("%d  ", deQueue(q));
   printf("%d  ", deQueue(q));

   getchar();
}
```

Queue can also be implemented using one user stack and one Function Call Stack.
Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

```
enQueue(x)
  1) Push x to stack1.

deQueue:
  1) If stack1 is empty then error.
  2) If stack1 has only one element then return it.
  3) Recursively pop everything from the stack1, store the popped item
     in a variable res,  push the res back to stack1 and return res
```

The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in *stack1* (step 2), we get the last element of *stack1* in dequeue() and all other items are pushed back in step 3.

Implementation of method 2 using Function Call Stack:

```
/* Program to implement a queue using one user defined stack and one Function Call Stack */
#include<stdio.h>
#include<stdlib.h>

/* structure of a stack node */
struct sNode
{
   int data;
   struct sNode *next;
};

/* structure of queue having two stacks */
struct queue
{
```

```
  struct sNode *stack1;
};

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data);

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref);

/* Function to enqueue an item to queue */
void enQueue(struct queue *q, int x)
{
  push(&q->stack1, x);
}

/* Function to dequeue an item from queue */
int deQueue(struct queue *q)
{
   int x, res;

   /* If both stacks are empty then error */
   if(q->stack1 == NULL)
   {
     printf("Q is empty");
     getchar();
     exit(0);
   }
   else if(q->stack1->next == NULL)
   {
      return pop(&q->stack1);
   }
   else
   {
      /* pop an item from the stack1 */
      x = pop(&q->stack1);

      /* store the last dequeued item */
      res = deQueue(q);

      /* push everything back to stack1 */
      push(&q->stack1, x);

      return res;
   }
}

/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
  /* allocate node */
  struct sNode* new_node =
            (struct sNode*) malloc(sizeof(struct sNode));

  if(new_node == NULL)
```

```
  {
     printf("Stack overflow \n");
     getchar();
     exit(0);
  }

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*top_ref);

  /* move the head to point to the new node */
  (*top_ref)    = new_node;
}

/* Function to pop an item from stack*/
int pop(struct sNode** top_ref)
{
  int res;
  struct sNode *top;

  /*If stack is empty then error */
  if(*top_ref == NULL)
  {
     printf("Stack overflow \n");
     getchar();
     exit(0);
  }
  else
  {
     top = *top_ref;
     res = top->data;
     *top_ref = top->next;
     free(top);
     return res;
  }
}

/* Driver function to test above functions */
int main()
{
  /* Create a queue with items 1 2 3*/
  struct queue *q = (struct queue*)malloc(sizeof(struct queue));
  q->stack1 = NULL;

  enQueue(q, 1);
  enQueue(q, 2);
  enQueue(q, 3);

  /* Dequeue items */
  printf("%d  ", deQueue(q));
  printf("%d  ", deQueue(q));
  printf("%d  ", deQueue(q));
```

```
  getchar();
}
```

Please write comments if you find any of the above codes/algorithms incorrect, or find better ways to solve the same problem.

Source

http://www.geeksforgeeks.org/queue-using-stacks/

Category: Misc

Post navigation

← Result of sizeof operator gets() is risky to use! →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 3

Check whether a given Binary Tree is Complete or not | Set 1 (Iterative Solution)

Given a Binary Tree, write a function to check whether the given Binary Tree is Complete Binary Tree or not.

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. See following examples.

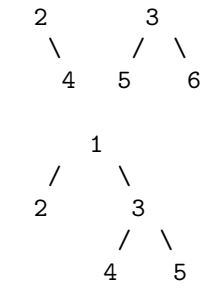
```
The following trees are examples of Complete Binary Trees
    1
  /   \
 2     3

       1
    /    \
   2       3
  /
 4

       1
    /    \
   2      3
  /  \    /
 4    5  6
```

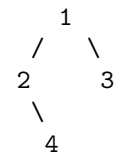
```
The following trees are examples of Non-Complete Binary Trees
    1
      \
       3

       1
    /    \
```



Source: Write an algorithm to check if a tree is complete binary tree or not

The method 2 of level order traversal post can be easily modified to check whether a tree is Complete or not. To understand the approach, let us first define a term 'Full Node'. A node is 'Full Node' if both left and right children are not empty (or not NULL).
The approach is to do a level order traversal starting from root. In the traversal, once a node is found which is NOT a Full Node, all the following nodes must be leaf nodes.
Also, one more thing needs to be checked to handle the below case: If a node has empty left child, then the right child must be empty.



Thanks to Guddu Sharma for suggesting this simple and efficient approach.

```
// A program to check if a given binary tree is complete or not
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_Q_SIZE 500

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* frunction prototypes for functions needed for Queue data
   structure. A queue is needed for level order tarversal */
struct node** createQueue(int *, int *);
void enQueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);
bool isQueueEmpty(int *front, int *rear);

/* Given a binary tree, return true if the tree is complete
```

```
   else false */
bool isCompleteBT(struct node* root)
{
  // Base Case: An empty tree is complete Binary Tree
  if (root == NULL)
    return true;

  // Create an empty queue
  int rear, front;
  struct node **queue = createQueue(&front, &rear);

  // Create a flag variable which will be set true
  // when a non full node is seen
  bool flag = false;

  // Do level order traversal using queue.
  enQueue(queue, &rear, root);
  while(!isQueueEmpty(&front, &rear))
  {
    struct node *temp_node = deQueue(queue, &front);

    /* Ceck if left child is present*/
    if(temp_node->left)
    {
       // If we have seen a non full node, and we see a node
       // with non-empty left child, then the given tree is not
       // a complete Binary Tree
       if (flag == true)
         return false;

       enQueue(queue, &rear, temp_node->left);  // Enqueue Left Child
    }
    else // If this a non-full node, set the flag as true
       flag = true;

    /* Ceck if right child is present*/
    if(temp_node->right)
    {
       // If we have seen a non full node, and we see a node
       // with non-empty left child, then the given tree is not
       // a complete Binary Tree
       if(flag == true)
         return false;

       enQueue(queue, &rear, temp_node->right);  // Enqueue Right Child
    }
    else // If this a non-full node, set the flag as true
       flag = true;
  }

  // If we reach here, then the tree is complete Bianry Tree
  return true;
}
```

```
/*UTILITY FUNCTIONS*/
struct node** createQueue(int *front, int *rear)
{
  struct node **queue =
   (struct node **)malloc(sizeof(struct node*)*MAX_Q_SIZE);

  *front = *rear = 0;
  return queue;
}

void enQueue(struct node **queue, int *rear, struct node *new_node)
{
  queue[*rear] = new_node;
  (*rear)++;
}

struct node *deQueue(struct node **queue, int *front)
{
  (*front)++;
  return queue[*front - 1];
}

bool isQueueEmpty(int *front, int *rear)
{
   return (*rear == *front);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
int main()
{
   /* Let us construct the following Binary Tree which
      is not a complete Binary Tree
            1
          /   \
         2     3
        / \     \
       4   5     6
    */

  struct node *root  = newNode(1);
```

```
  root->left         = newNode(2);
  root->right        = newNode(3);
  root->left->left   = newNode(4);
  root->left->right  = newNode(5);
  root->right->right = newNode(6);

  if ( isCompleteBT(root) == true )
      printf ("Complete Binary Tree");
  else
      printf ("NOT Complete Binary Tree");

  return 0;
}
```

Output:

```
NOT Complete Binary Tree
```

Time Complexity: O(n) where n is the number of nodes in given Binary Tree

Auxiliary Space: O(n) for queue.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-complete-tree-or-not/

Category: Trees

Post navigation

← Microsoft Interview | Set 3 Amazon Interview | Set 4 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 4

Find the largest multiple of 3

Given an array of non-negative integers. Find the largest multiple of 3 that can be formed from array elements.

For example, if the input array is {8, 1, 9}, the output should be "9 8 1 , and if the input array is {8, 1, 7, 6, 0}, output should be "8 7 6 0 .

Method 1 (Brute Force)
The simple & straight forward approach is to generate all the combinations of the elements and keep track of the largest number formed which is divisible by 3.

Time Complexity: O(n x 2^n). There will be 2^n combinations of array elements. To compare each combination with the largest number so far may take O(n) time.
Auxiliary Space: O(n) // to avoid integer overflow, the largest number is assumed to be stored in the form of array.

Method 2 (Tricky)
This problem can be solved efficiently with the help of O(n) extra space. This method is based on the following facts about numbers which are multiple of 3.

1) A number is multiple of 3 if and only if the sum of digits of number is multiple of 3. For example, let us consider 8760, it is a multiple of 3 because sum of digits is 8 + 7+ 6+ 0 = 21, which is a multiple of 3.

2) If a number is multiple of 3, then all permutations of it are also multiple of 3. For example, since 6078 is a multiple of 3, the numbers 8760, 7608, 7068, ... are also multiples of 3.

3) We get the same remainder when we divide the number and sum of digits of the number. For example, if divide number 151 and sum of it digits 7, by 3, we get the same remainder 1.

What is the idea behind above facts?
The value of 10%3 and 100%3 is 1. The same is true for all the higher powers of 10, because 3 divides 9, 99, 999, … etc.
Let us consider a 3 digit number n to prove above facts. Let the first, second and third digits of n be 'a', 'b' and 'c' respectively. n can be written as

```
n = 100.a + 10.b + c
```

Since (10^x)%3 is 1 for any x, the above expression gives the same remainder as following expression

```
1.a + 1.b + c
```

So the remainder obtained by sum of digits and 'n' is same.

Following is a solution based on the above observation.

1. Sort the array in non-decreasing order.

2. Take three queues. One for storing elements which on dividing by 3 gives remainder as 0.The second queue stores digits which on dividing by 3 gives remainder as 1. The third queue stores digits which on dividing by 3 gives remainder as 2. Call them as queue0, queue1 and queue2

3. Find the sum of all the digits.

4. Three cases arise:
.....**4.1** The sum of digits is divisible by 3. Dequeue all the digits from the three queues. Sort them in non-increasing order. Output the array.

.....**4.2** The sum of digits produces remainder 1 when divided by 3.
Remove one item from queue1. If queue1 is empty, remove two items from queue2. If queue2 contains less than two items, the number is not possible.

.....**4.3** The sum of digits produces remainder 2 when divided by 3.
Remove one item from queue2. If queue2 is empty, remove two items from queue1. If queue1 contains less than two items, the number is not possible.

5. Finally empty all the queues into an auxiliary array. Sort the auxiliary array in non-increasing order. Output the auxiliary array.

Based on the above, below is C implementation:

The below code works only if the input arrays has numbers from 0 to 9. It can be easily extended for any positive integer array. We just have to modify the part where we sort the array in decreasing order, at the end of code.

```
/* A program to find the largest multiple of 3 from an array of elements */
#include <stdio.h>
#include <stdlib.h>

// A queue node
typedef struct Queue
{
    int front;
    int rear;
    int capacity;
    int* array;
} Queue;

// A utility function to create a queue with given capacity
Queue* createQueue( int capacity )
{
    Queue* queue = (Queue *) malloc (sizeof(Queue));
    queue->capacity = capacity;
    queue->front = queue->rear = -1;
    queue->array = (int *) malloc (queue->capacity * sizeof(int));
    return queue;
}

// A utility function to check if queue is empty
int isEmpty (Queue* queue)
```

```
{
    return queue->front == -1;
}

// A function to add an item to queue
void Enqueue (Queue* queue, int item)
{
    queue->array[ ++queue->rear ] = item;
    if ( isEmpty(queue) )
        ++queue->front;
}

// A function to remove an item from queue
int Dequeue (Queue* queue)
{
    int item = queue->array[ queue->front ];
    if( queue->front == queue->rear )
        queue->front = queue->rear = -1;
    else
        queue->front++;

    return item;
}

// A utility function to print array contents
void printArr (int* arr, int size)
{
    int i;
    for (i = 0; i< size; ++i)
        printf ("%d ", arr[i]);
}

/* Following two functions are needed for library function qsort().
   Refer following link for help of qsort()
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compareAsc( const void* a, const void* b )
{
    return *(int*)a > *(int*)b;
}
int compareDesc( const void* a, const void* b )
{
    return *(int*)a < *(int*)b;
}

// This function puts all elements of 3 queues in the auxiliary array
void populateAux (int* aux, Queue* queue0, Queue* queue1,
                            Queue* queue2, int* top )
{
    // Put all items of first queue in aux[]
    while ( !isEmpty(queue0) )
        aux[ (*top)++ ] = Dequeue( queue0 );

    // Put all items of second queue in aux[]
    while ( !isEmpty(queue1) )
```

```
        aux[ (*top)++ ] = Dequeue( queue1 );

    // Put all items of third queue in aux[]
    while ( !isEmpty(queue2) )
        aux[ (*top)++ ] = Dequeue( queue2 );
}

// The main function that finds the largest possible multiple of
// 3 that can be formed by arr[] elements
int findMaxMultupleOf3( int* arr, int size )
{
    // Step 1: sort the array in non-decreasing order
    qsort( arr, size, sizeof( int ), compareAsc );

    // Create 3 queues to store numbers with remainder 0, 1
    // and 2 respectively
    Queue* queue0 = createQueue( size );
    Queue* queue1 = createQueue( size );
    Queue* queue2 = createQueue( size );

    // Step 2 and 3 get the sum of numbers and place them in
    // corresponding queues
    int i, sum;
    for ( i = 0, sum = 0; i < size; ++i )
    {
        sum += arr[i];
        if ( (arr[i] % 3) == 0 )
            Enqueue( queue0, arr[i] );
        else if ( (arr[i] % 3) == 1 )
            Enqueue( queue1, arr[i] );
        else
            Enqueue( queue2, arr[i] );
    }

    // Step 4.2: The sum produces remainder 1
    if ( (sum % 3) == 1 )
    {
        // either remove one item from queue1
        if ( !isEmpty( queue1 ) )
            Dequeue( queue1 );

        // or remove two items from queue2
        else
        {
            if ( !isEmpty( queue2 ) )
                Dequeue( queue2 );
            else
                return 0;

            if ( !isEmpty( queue2 ) )
                Dequeue( queue2 );
            else
                return 0;
        }
```

```
    }

    // Step 4.3: The sum produces remainder 2
    else if ((sum % 3) == 2)
    {
        // either remove one item from queue2
        if ( !isEmpty( queue2 ) )
            Dequeue( queue2 );

        // or remove two items from queue1
        else
        {
            if ( !isEmpty( queue1 ) )
                Dequeue( queue1 );
            else
                return 0;

            if ( !isEmpty( queue1 ) )
                Dequeue( queue1 );
            else
                return 0;
        }
    }

    int aux[size], top = 0;

    // Empty all the queues into an auxiliary array.
    populateAux (aux, queue0, queue1, queue2, &top);

    // sort the array in non-increasing order
    qsort (aux, top, sizeof( int ), compareDesc);

    // print the result
    printArr (aux, top);

    return top;
}

// Driver program to test above functions
int main()
{
    int arr[] = {8, 1, 7, 6, 0};
    int size = sizeof(arr)/sizeof(arr[0]);

    if (findMaxMultupleOf3( arr, size ) == 0)
        printf( "Not Possible" );

    return 0;
}
```

The above method can be optimized in following ways.
1) We can use Heap Sort or Merge Sort to make the time complexity O(nLogn).

2) We can avoid extra space for queues. We know at most two items will be removed from the input array.

So we can keep track of two items in two variables.

3) At the end, instead of sorting the array again in descending order, we can print the ascending sorted array in reverse order. While printing in reverse order, we can skip the two elements to be removed.

Time Complexity: O(nLogn), assuming a O(nLogn) algorithm is used for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-the-largest-number-multiple-of-3/

Chapter 5

Find the first circular tour that visits all petrol pumps

Suppose there is a circle. There are n petrol pumps on that circle. You are given two sets of data.

1. The amount of petrol that every petrol pump has.
2. Distance from that petrol pump to the next petrol pump.

Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). Expected time complexity is O(n). Assume for 1 litre petrol, the truck can go 1 unit of distance.

For example, let there be 4 petrol pumps with amount of petrol and distance to next petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from where truck can make a circular tour is 2nd petrol pump. Output should be "start = 1 (index of 2nd petrol pump).

A **Simple Solution** is to consider every petrol pumps as starting point and see if there is a possible tour. If we find a starting point with feasible solution, we return that starting point. The worst case time complexity of this solution is O(n^2).

We can **use a Queue** to store the current tour. We first enqueue first petrol pump to the queue, we keep enqueueing petrol pumps till we either complete the tour, or current amount of petrol becomes negative. If the amount becomes negative, then we keep dequeueing petrol pumps till the current amount becomes positive or queue becomes empty.

Instead of creating a separate queue, we use the given array itself as queue. We maintain two index variables start and end that represent rear and front of queue.

```
// C program to find circular tour for a truck
#include <stdio.h>

// A petrol pump has petrol and distance to next petrol pump
struct petrolPump
{
  int petrol;
  int distance;
};

// The function returns starting point if there is a possible solution,
// otherwise returns -1
```

```
int printTour(struct petrolPump arr[], int n)
{
    // Consider first petrol pump as a starting point
    int start = 0;
    int end =  1;

    int curr_petrol = arr[start].petrol - arr[start].distance;

    /* Run a loop while all petrol pumps are not visited.
      And we have reached first petrol pump again with 0 or more petrol */
    while (end != start || curr_petrol < 0)
    {
        // If curremt amount of petrol in truck becomes less than 0, then
        // remove the starting petrol pump from tour
        while (curr_petrol < 0 && start != end)
        {
            // Remove starting petrol pump. Change start
            curr_petrol -= arr[start].petrol - arr[start].distance;
            start = (start + 1)%n;

            // If 0 is being considered as start again, then there is no
            // possible solution
            if (start == 0)
               return -1;
        }

        // Add a petrol pump to current tour
        curr_petrol += arr[end].petrol - arr[end].distance;

        end = (end + 1)%n;
    }

    // Return starting point
    return start;
}

// Driver program to test above functions
int main()
{
    struct petrolPump arr[] = {{6, 4}, {3, 6}, {7, 3}};

    int n = sizeof(arr)/sizeof(arr[0]);
    int start = printTour(arr, n);

    (start == -1)? printf("No solution"): printf("Start = %d", start);

    return 0;
}
```

Output:

```
 start = 2
```

Time Complexity: Seems to be more than linear at first look. If we consider the items between start and end as part of a circular queue, we can observe that every item is enqueued at most two times to the queue. The total number of operations is proportional to total number of enqueue operations. Therefore the time complexity is O(n).

Auxiliary Space: O(1)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-a-tour-that-visits-all-stations/

Chapter 6

Maximum of all subarrays of size k (Added a O(n) method)

Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.

Examples:

Input :
arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}
k = 3
Output :
3 3 4 5 5 5 6

Input :
arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}
k = 4
Output :
10 10 10 15 15 90 90

Method 1 (Simple)
Run two loops. In the outer loop, take all subarrays of size k. In the inner loop, get the maximum of the current subarray.

```
#include<stdio.h>

void printKMax(int arr[], int n, int k)
{
    int j, max;

    for (int i = 0; i <= n-k; i++)
    {
        max = arr[i];

        for (j = 1; j < k; j++)
        {
            if (arr[i+j] > max)
               max = arr[i+j];
        }
        printf("%d ", max);
```

```
    }
}


int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Time Complexity: The outer loop runs n-k+1 times and the inner loop runs k times for every iteration of outer loop. So time complexity is O((n-k+1)*k) which can also be written as O(nk).

Method 2 (Use Self-Balancing BST)
1) Pick first k elements and create a Self-Balancing Binary Search Tree (BST) of size k.
2) Run a loop for i = 0 to n – k
…..a) Get the maximum element from the BST, and print it.
…..b) Search for arr[i] in the BST and delete it from the BST.
…..c) Insert arr[i+k] into the BST.

Time Complexity: Time Complexity of step 1 is O(kLogk). Time Complexity of steps 2(a), 2(b) and 2(c) is O(Logk). Since steps 2(a), 2(b) and 2(c) are in a loop that runs n-k+1 times, time complexity of the complete algorithm is O(kLogk + (n-k+1)*Logk) which can also be written as O(nLogk).

Method 3 (A O(n) method: use Dequeue)
We create a Dequeue, *Qi* of capacity k, that stores only useful elements of current window of k elements. An element is useful if it is in current window and is greater than all other elements on left side of it in current window. We process all array elements one by one and maintain *Qi* to contain useful elements of current window and these useful elements are maintained in sorted order. The element at front of the *Qi* is the largest and element at rear of *Qi* is the smallest of current window. Thanks to Aashishfor suggesting this method.

Following is C++ implementation of this method.

```
#include <iostream>
#include <deque>

using namespace std;

// A Dequeue (Double ended queue) based method for printing maixmum element of
// all subarrays of size k
void printKMax(int arr[], int n, int k)
{
    // Create a Double Ended Queue, Qi that will store indexes of array elements
    // The queue will store indexes of useful elements in every window and it will
    // maintain decreasing order of values from front to rear in Qi, i.e.,
    // arr[Qi.front[]] to arr[Qi.rear()] are sorted in decreasing order
    std::deque<int>  Qi(k);

    /* Process first k (or first window) elements of array */
    int i;
    for (i = 0; i < k; ++i)
```

```
    {
        // For very element, the previous smaller elements are useless so
        // remove them from Qi
        while ( (!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();  // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i);
    }

    // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for ( ; i < n; ++i)
    {
        // The element at the front of the queue is the largest element of
        // previous window, so print it
        cout << arr[Qi.front()] << " ";

        // Remove the elements which are out of this window
        while ( (!Qi.empty()) && Qi.front() <= i - k)
            Qi.pop_front();  // Remove from front of queue

        // Remove all elements smaller than the currently
        // being added element (remove useless elements)
        while ( (!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();

         // Add current element at the rear of Qi
        Qi.push_back(i);
    }

    // Print the maximum element of last window
    cout << arr[Qi.front()];
}

// Driver program to test above functions
int main()
{
    int arr[] = {12, 1, 78, 90, 57, 89, 56};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Output:

```
 78 90 90 90 89
```

Time Complexity: O(n). It seems more than O(n) at first look. If we take a closer look, we can observe that every element of array is added and removed at most once. So there are total 2n operations.
Auxiliary Space: O(k)

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

Source

http://www.geeksforgeeks.org/maximum-of-all-subarrays-of-size-k/

Category: Arrays Tags: Queue, Stack-Queue, StackAndQueue

Post navigation

← XOR Linked List – A Memory Efficient Doubly Linked List | Set 1 Time Complexity of building a heap →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 7

An Interesting Method to Generate Binary Numbers from 1 to n

Given a number n, write a function that generates and prints all binary numbers with decimal values from 1 to n.

Examples:

```
Input: n = 2
Output: 1, 10

Input: n = 5
Output: 1, 10, 11, 100, 101
```

A simple method is to run a loop from 1 to n, call decimal to binary inside the loop.

Following is an interesting method that uses queue data structureto print binary numbers. Thanks to Vivekfor suggesting this approach.
1) Create an empty queue of strings
2) Enqueue the first binary number "1" to queue.
3) Now run a loop for generating and printing n binary numbers.
……a) Dequeue and Print the front of queue.
……b) Append "0" at the end of front item and enqueue it.
……c) Append "1" at the end of front item and enqueue it.

Following is C++ implementation of above algorithm.

```
// C++ program to generate binary numbers from 1 to n
#include <iostream>
#include <queue>
using namespace std;

// This function uses queue data structure to print binary numbers
void generatePrintBinary(int n)
{
    // Create an empty queue of strings
```

```
    queue<string> q;

    // Enqueue the first binary number
    q.push("1");

    // This loops is like BFS of a tree with 1 as root
    // 0 as left child and 1 as right child and so on
    while (n--)
    {
        // print the front of queue
        string s1 = q.front();
        q.pop();
        cout << s1 << "\n";

        string s2 = s1;  // Store s1 before changing it

        // Append "0" to s1 and enqueue it
        q.push(s1.append("0"));

        // Append "1" to s2 and enqueue it. Note that s2 contains
        // the previous front
        q.push(s2.append("1"));
    }
}

// Driver program to test above function
int main()
{
    int n = 10;
    generatePrintBinary(n);
    return 0;
}
```

Output:

```
 1
10
11
100
101
110
111
1000
1001
1010
```

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/interesting-method-generate-binary-numbers-1-n/

Category: Misc

Chapter 8

How to efficiently implement k Queues in a single array?

We have discussed efficient implementation of k stack in an array. In this post, same for queue is discussed. Following is the detailed problem statement.

Create a data structure kQueues that represents k queues. Implementation of kQueues should use only one array, i.e., k queues should use the same array for storing elements. Following functions must be supported by kQueues.

enqueue(int x, int qn) –> adds x to queue number 'qn' where qn is from 0 to k-1
dequeue(int qn) –> deletes an element from queue number 'qn' where qn is from 0 to k-1

Method 1 (Divide the array in slots of size n/k)
A simple way to implement k queues is to divide the array in k slots of size n/k each, and fix the slots for different queues, i.e., use arr[0] to arr[n/k-1] for first queue, and arr[n/k] to arr[2n/k-1] for queue2 where arr[] is the array to be used to implement two queues and size of array be n.

The problem with this method is inefficient use of array space. An enqueue operation may result in overflow even if there is space available in arr[]. For example, consider k as 2 and array size n as 6. Let we enqueue 3 elements to first and do not enqueue anything to second second queue. When we enqueue 4th element to first queue, there will be overflow even if we have space for 3 more elements in array.

Method 2 (A space efficient implementation)
The idea is similar to the stack post, here we need to use three extra arrays. In stack post, we needed to extra arrays, one more array is required because in queues, enqueue() and dequeue() operations are done at different ends.

Following are the three extra arrays are used:
1) **front**[]: This is of size k and stores indexes of front elements in all queues.
2) **rear**[]: This is of size k and stores indexes of rear elements in all queues.
2) **next**[]: This is of size n and stores indexes of next item for all items in array arr[].

Here arr[] is actual array that stores k stacks.

Together with k queues, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable 'free'.

All entries in front[] are initialized as -1 to indicate that all queues are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, 'free' is initialized as 0.

Following is C++ implementation of the above idea.

```
// A C++ program to demonstrate implementation of k queues in a single
// array in time and space efficient way
#include<iostream>
#include<climits>
using namespace std;

// A C++ class to represent k queues in a single array of size n
class kQueues
{
    int *arr;   // Array of size n to store actual content to be stored in queue
    int *front;   // Array of size k to store indexes of front elements of queue
    int *rear;   // Array of size k to store indexes of rear elements of queue
    int *next;  // Array of size n to store next entry in all queues
                // and free list
    int n, k;
    int free; // To store beginning index of free list
public:
    //constructor to create k queue in an array of size n
    kQueues(int k, int n);

    // A utility function to check if there is space available
    bool isFull()   {  return (free == -1);  }

    // To enqueue an item in queue number 'qn' where qn is from 0 to k-1
    void enqueue(int item, int qn);

    // To dequeue an from queue number 'qn' where qn is from 0 to k-1
    int dequeue(int qn);

    // To check whether queue number 'qn' is empty or not
    bool isEmpty(int qn)  {  return (front[qn] == -1); }
};

// Constructor to create k queues in an array of size n
kQueues::kQueues(int k1, int n1)
{
    // Initialize n and k, and allocate memory for all arrays
    k = k1, n = n1;
    arr = new int[n];
    front = new int[k];
    rear = new int[k];
    next = new int[n];

    // Initialize all queues as empty
    for (int i = 0; i < k; i++)
        front[i] = -1;

    // Initialize all spaces as free
    free = 0;
    for (int i=0; i<n-1; i++)
        next[i] = i+1;
    next[n-1] = -1;  // -1 is used to indicate end of free list
}
```

```
// To enqueue an item in queue number 'qn' where qn is from 0 to k-1
void kQueues::enqueue(int item, int qn)
{
    // Overflow check
    if (isFull())
    {
        cout << "\nQueue Overflow\n";
        return;
    }

    int i = free;      // Store index of first free slot

    // Update index of free slot to index of next slot in free list
    free = next[i];

    if (isEmpty(qn))
        front[qn] = i;
    else
        next[rear[qn]] = i;

    next[i] = -1;

    // Update next of rear and then rear for queue number 'qn'
    rear[qn] = i;

    // Put the item in array
    arr[i] = item;
}

// To dequeue an from queue number 'qn' where qn is from 0 to k-1
int kQueues::dequeue(int qn)
{
    // Underflow checkSAS
    if (isEmpty(qn))
    {
         cout << "\nQueue Underflow\n";
         return INT_MAX;
    }

    // Find index of front item in queue number 'qn'
    int i = front[qn];

    front[qn] = next[i];  // Change top to store next of previous top

    // Attach the previous front to the beginning of free list
    next[i] = free;
    free = i;

    // Return the previous front item
    return arr[i];
}

/* Driver program to test kStacks class */
```

```
int main()
{
    // Let us create 3 queue in an array of size 10
    int k = 3, n = 10;
    kQueues ks(k, n);

    // Let us put some items in queue number 2
    ks.enqueue(15, 2);
    ks.enqueue(45, 2);

    // Let us put some items in queue number 1
    ks.enqueue(17, 1);
    ks.enqueue(49, 1);
    ks.enqueue(39, 1);

    // Let us put some items in queue number 0
    ks.enqueue(11, 0);
    ks.enqueue(9, 0);
    ks.enqueue(7, 0);

    cout << "Dequeued element from queue 2 is " << ks.dequeue(2) << endl;
    cout << "Dequeued element from queue 1 is " << ks.dequeue(1) << endl;
    cout << "Dequeued element from queue 0 is " << ks.dequeue(0) << endl;

    return 0;
}
```

Output:

```
 Dequeued element from queue 2 is 15
Dequeued element from queue 1 is 17
Dequeued element from queue 0 is 11
```

Time complexities of enqueue() and dequeue() is O(1).

The best part of above implementation is, if there is a slot available in queue, then an item can be enqueued in any of the queues, i.e., no wastage of space. This method requires some extra space. Space may not be an issue because queue items are typically large, for example queues of employees, students, etc where every item is of hundreds of bytes. For such large queues, the extra space used is comparatively very less as we use three integer arrays as extra space.

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/efficiently-implement-k-queues-single-array/