# Copyright

The content of this book comes from [geeksforgeeks.org](http://geeksforgeeks.org) and it's licensed under [Creative Commons Attribution-NonCommercial-NoDerivs 2.5 India](http://creativecommons.org/licenses/by-nc-nd/2.5/in/deed.en_US)

Made by [Jing](http://www.jing-zhou.me/). 2015.

Updated on September 30, 2015

Head over to [this github repo](https://github.com/gnijuohz/geeksforgeeks-as-books) to report issues or contribute.

# 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](http://en.wikipedia.org/wiki/Double-ended_queue), *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 [Aashish](http://www.geeksforgeeks.org/maximum-of-all-subarrays-of-size-k/#comment-10874)for 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/>

# Implement LRU Cache

How to implement LRU caching scheme? What data structures should be used?

We are given total possible page numbers that can be referred. We are also given cache (or memory) size (Number of page frames that cache can hold at a time). The LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache. Please see the Galvin book for more details (see the LRU page replacement slide [here](http://www.wiley.com/college/silberschatz6e/0471417432/slides/pdf2/mod10.2.pdf)).

We use two data structures to implement an LRU Cache.

**1.** A Queue which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size).  
 The most recently used pages will be near front end and least recently pages will be near rear end.

**2.** A Hash with page number as key and address of the corresponding queue node as value.

When a page is referenced, the required page may be in the memory. If it is in the memory, we need to detach the node of the list and bring it to the front of the queue.  
 If the required page is not in the memory, we bring that in memory. In simple words, we add a new node to the front of the queue and update the corresponding node address in the hash. If the queue is full, i.e. all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.

Note: Initially no page is in the memory.

Below is C implementation:

// A C program to show implementation of LRU cache  
#include <stdio.h>  
#include <stdlib.h>  
  
// A Queue Node (Queue is implemented using Doubly Linked List)  
typedef struct QNode  
{  
 struct QNode \*prev, \*next;  
 unsigned pageNumber; // the page number stored in this QNode  
} QNode;  
  
// A Queue (A FIFO collection of Queue Nodes)  
typedef struct Queue  
{  
 unsigned count; // Number of filled frames  
 unsigned numberOfFrames; // total number of frames  
 QNode \*front, \*rear;  
} Queue;  
  
// A hash (Collection of pointers to Queue Nodes)  
typedef struct Hash  
{  
 int capacity; // how many pages can be there  
 QNode\* \*array; // an array of queue nodes  
} Hash;  
  
// A utility function to create a new Queue Node. The queue Node  
// will store the given 'pageNumber'  
QNode\* newQNode( unsigned pageNumber )  
{  
 // Allocate memory and assign 'pageNumber'  
 QNode\* temp = (QNode \*)malloc( sizeof( QNode ) );  
 temp->pageNumber = pageNumber;  
  
 // Initialize prev and next as NULL  
 temp->prev = temp->next = NULL;  
  
 return temp;  
}  
  
// A utility function to create an empty Queue.  
// The queue can have at most 'numberOfFrames' nodes  
Queue\* createQueue( int numberOfFrames )  
{  
 Queue\* queue = (Queue \*)malloc( sizeof( Queue ) );  
  
 // The queue is empty  
 queue->count = 0;  
 queue->front = queue->rear = NULL;  
  
 // Number of frames that can be stored in memory  
 queue->numberOfFrames = numberOfFrames;  
  
 return queue;  
}  
  
// A utility function to create an empty Hash of given capacity  
Hash\* createHash( int capacity )  
{  
 // Allocate memory for hash  
 Hash\* hash = (Hash \*) malloc( sizeof( Hash ) );  
 hash->capacity = capacity;  
  
 // Create an array of pointers for refering queue nodes  
 hash->array = (QNode \*\*) malloc( hash->capacity \* sizeof( QNode\* ) );  
  
 // Initialize all hash entries as empty  
 int i;  
 for( i = 0; i < hash->capacity; ++i )  
 hash->array[i] = NULL;  
  
 return hash;  
}  
  
// A function to check if there is slot available in memory  
int AreAllFramesFull( Queue\* queue )  
{  
 return queue->count == queue->numberOfFrames;  
}  
  
// A utility function to check if queue is empty  
int isQueueEmpty( Queue\* queue )  
{  
 return queue->rear == NULL;  
}  
  
// A utility function to delete a frame from queue  
void deQueue( Queue\* queue )  
{  
 if( isQueueEmpty( queue ) )  
 return;  
  
 // If this is the only node in list, then change front  
 if (queue->front == queue->rear)  
 queue->front = NULL;  
  
 // Change rear and remove the previous rear  
 QNode\* temp = queue->rear;  
 queue->rear = queue->rear->prev;  
  
 if (queue->rear)  
 queue->rear->next = NULL;  
  
 free( temp );  
  
 // decrement the number of full frames by 1  
 queue->count--;  
}  
  
// A function to add a page with given 'pageNumber' to both queue  
// and hash  
void Enqueue( Queue\* queue, Hash\* hash, unsigned pageNumber )  
{  
 // If all frames are full, remove the page at the rear  
 if ( AreAllFramesFull ( queue ) )  
 {  
 // remove page from hash  
 hash->array[ queue->rear->pageNumber ] = NULL;  
 deQueue( queue );  
 }  
  
 // Create a new node with given page number,  
 // And add the new node to the front of queue  
 QNode\* temp = newQNode( pageNumber );  
 temp->next = queue->front;  
  
 // If queue is empty, change both front and rear pointers  
 if ( isQueueEmpty( queue ) )  
 queue->rear = queue->front = temp;  
 else // Else change the front  
 {  
 queue->front->prev = temp;  
 queue->front = temp;  
 }  
  
 // Add page entry to hash also  
 hash->array[ pageNumber ] = temp;  
  
 // increment number of full frames  
 queue->count++;  
}  
  
// This function is called when a page with given 'pageNumber' is referenced  
// from cache (or memory). There are two cases:  
// 1. Frame is not there in memory, we bring it in memory and add to the front  
// of queue  
// 2. Frame is there in memory, we move the frame to front of queue  
void ReferencePage( Queue\* queue, Hash\* hash, unsigned pageNumber )  
{  
 QNode\* reqPage = hash->array[ pageNumber ];  
  
 // the page is not in cache, bring it  
 if ( reqPage == NULL )  
 Enqueue( queue, hash, pageNumber );  
  
 // page is there and not at front, change pointer  
 else if (reqPage != queue->front)  
 {  
 // Unlink rquested page from its current location  
 // in queue.  
 reqPage->prev->next = reqPage->next;  
 if (reqPage->next)  
 reqPage->next->prev = reqPage->prev;  
  
 // If the requested page is rear, then change rear  
 // as this node will be moved to front  
 if (reqPage == queue->rear)  
 {  
 queue->rear = reqPage->prev;  
 queue->rear->next = NULL;  
 }  
  
 // Put the requested page before current front  
 reqPage->next = queue->front;  
 reqPage->prev = NULL;  
  
 // Change prev of current front  
 reqPage->next->prev = reqPage;  
  
 // Change front to the requested page  
 queue->front = reqPage;  
 }  
}  
  
// Driver program to test above functions  
int main()  
{  
 // Let cache can hold 4 pages  
 Queue\* q = createQueue( 4 );  
  
 // Let 10 different pages can be requested (pages to be  
 // referenced are numbered from 0 to 9  
 Hash\* hash = createHash( 10 );  
  
 // Let us refer pages 1, 2, 3, 1, 4, 5  
 ReferencePage( q, hash, 1);  
 ReferencePage( q, hash, 2);  
 ReferencePage( q, hash, 3);  
 ReferencePage( q, hash, 1);  
 ReferencePage( q, hash, 4);  
 ReferencePage( q, hash, 5);  
  
 // Let us print cache frames after the above referenced pages  
 printf ("%d ", q->front->pageNumber);  
 printf ("%d ", q->front->next->pageNumber);  
 printf ("%d ", q->front->next->next->pageNumber);  
 printf ("%d ", q->front->next->next->next->pageNumber);  
  
 return 0;  
}

Output:

5 4 1 3

This article is compiled by [Aashish Barnwal](https://www.facebook.com/barnwal.aashish)and reviewed by GeeksforGeeks team. 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/implement-lru-cache/>

Category: [Linked Lists](http://www.geeksforgeeks.org/category/linked-list/) Tags: [Advance Data Structures](http://www.geeksforgeeks.org/tag/advance-data-structures/), [Advanced Data Structures](http://www.geeksforgeeks.org/tag/advanced-data-structures/), [Queue](http://www.geeksforgeeks.org/tag/queue/)

Post navigation

[← Microsoft Interview | Set 7](http://www.geeksforgeeks.org/microsoft-interview-set-7-3/) [Median of two sorted arrays of different sizes →](http://www.geeksforgeeks.org/median-of-two-sorted-arrays-of-different-sizes/)

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org/), generate link and share the link here.

# Implement Stack using Queues

The problem is opposite of [this](http://www.geeksforgeeks.org/archives/5009)post. We are given a Queue data structure that supports standard operations like enqueue() and dequeue(). We need to implement a Stack data structure using only instances of Queue.

A stack can be implemented using two queues. Let stack to be implemented be ‘s’ and queues used to implement be ‘q1′ and ‘q2′. Stack ‘s’ can be implemented in two ways:

**Method 1 (By making push operation costly)**  
 This method makes sure that newly entered element is always at the front of ‘q1′, so that pop operation just dequeues from ‘q1′. ‘q2′ is used to put every new element at front of ‘q1′.

push(s, x) // x is the element to be pushed and s is stack  
 1) Enqueue x to q2  
 2) One by one dequeue everything from q1 and enqueue to q2.  
 3) Swap the names of q1 and q2   
// Swapping of names is done to avoid one more movement of all elements   
// from q2 to q1.   
  
pop(s)  
 1) Dequeue an item from q1 and return it.

**Method 2 (By making pop operation costly)**  
 In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.

push(s, x)  
 1) Enqueue x to q1 (assuming size of q1 is unlimited).  
  
pop(s)   
 1) One by one dequeue everything except the last element from q1 and enqueue to q2.  
 2) Dequeue the last item of q1, the dequeued item is result, store it.  
 3) Swap the names of q1 and q2  
 4) Return the item stored in step 2.  
// Swapping of names is done to avoid one more movement of all elements   
// from q2 to q1.

**References:**  
 [Implement Stack using Two Queues](http://stackoverflow.com/questions/688276/implement-stack-using-two-queues)

This article is compiled by **Sumit Jain** and reviewed by GeeksforGeeks team. 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/implement-stack-using-queue/>

Category: [Misc](http://www.geeksforgeeks.org/category/c-programs/) Tags: [Queue](http://www.geeksforgeeks.org/tag/queue/), [stack](http://www.geeksforgeeks.org/tag/stack/)

# 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

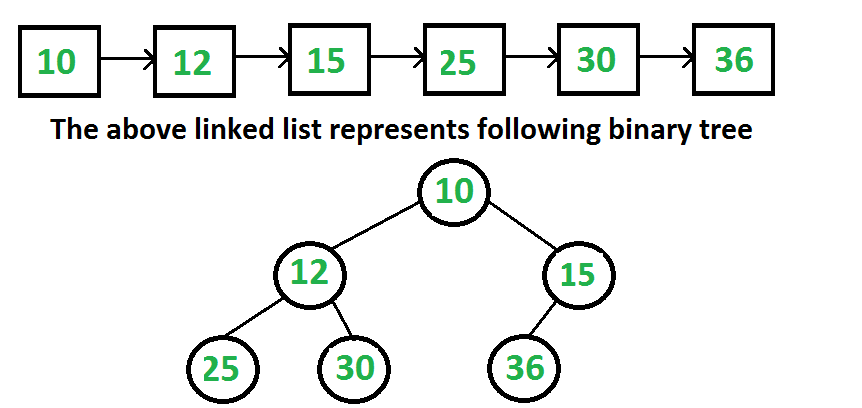
### Source

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

# Construct Complete Binary Tree from its Linked List Representation

Given Linked List Representation of Complete Binary Tree, construct the Binary tree. A complete binary tree can be represented in an array in the following approach.

If root node is stored at index i, its left, and right children are stored at indices 2\*i+1, 2\*i+2 respectively.  
 Suppose tree is represented by a linked list in same way, how do we convert this into normal linked representation of binary tree where every node has data, left and right pointers? In the linked list representation, we cannot directly access the children of the current node unless we traverse the list.

[](http://d2hc1qfcrygj4j.cloudfront.net//wp-content/uploads/LinkedListToBST.png)

We are mainly given level order traversal in sequential access form. We know head of linked list is always is root of the tree. We take the first node as root and we also know that the next two nodes are left and right children of root. So we know partial Binary Tree. The idea is to do Level order traversal of the partially built Binary Tree using queue and traverse the linked list at the same time. At every step, we take the parent node from queue, make next two nodes of linked list as children of the parent node, and enqueue the next two nodes to queue.

**1.** Create an empty queue.  
 **2.** Make the first node of the list as root, and enqueue it to the queue.  
 **3.** Until we reach the end of the list, do the following.  
 ………**a.** Dequeue one node from the queue. This is the current parent.  
 ………**b.** Traverse two nodes in the list, add them as children of the current parent.  
 ………**c.** Enqueue the two nodes into the queue.

Below is the code which implements the same in C++.

// C++ program to create a Complete Binary tree from its Linked List  
// Representation  
#include <iostream>  
#include <string>  
#include <queue>  
using namespace std;  
  
// Linked list node  
struct ListNode  
{  
 int data;  
 ListNode\* next;  
};  
  
// Binary tree node structure  
struct BinaryTreeNode  
{  
 int data;  
 BinaryTreeNode \*left, \*right;  
};  
  
// Function to insert a node at the beginning of the Linked List  
void push(struct ListNode\*\* head\_ref, int new\_data)  
{  
 // allocate node and assign data  
 struct ListNode\* new\_node = new ListNode;  
 new\_node->data = new\_data;  
  
 // link the old list off the new node  
 new\_node->next = (\*head\_ref);  
  
 // move the head to point to the new node  
 (\*head\_ref) = new\_node;  
}  
  
// method to create a new binary tree node from the given data  
BinaryTreeNode\* newBinaryTreeNode(int data)  
{  
 BinaryTreeNode \*temp = new BinaryTreeNode;  
 temp->data = data;  
 temp->left = temp->right = NULL;  
 return temp;  
}  
  
// converts a given linked list representing a complete binary tree into the  
// linked representation of binary tree.  
void convertList2Binary(ListNode \*head, BinaryTreeNode\* &root)  
{  
 // queue to store the parent nodes  
 queue<BinaryTreeNode \*> q;  
  
 // Base Case  
 if (head == NULL)  
 {  
 root = NULL; // Note that root is passed by reference  
 return;  
 }  
  
 // 1.) The first node is always the root node, and add it to the queue  
 root = newBinaryTreeNode(head->data);  
 q.push(root);  
  
 // advance the pointer to the next node  
 head = head->next;  
  
 // until the end of linked list is reached, do the following steps  
 while (head)  
 {  
 // 2.a) take the parent node from the q and remove it from q  
 BinaryTreeNode\* parent = q.front();  
 q.pop();  
  
 // 2.c) take next two nodes from the linked list. We will add  
 // them as children of the current parent node in step 2.b. Push them   
 // into the queue so that they will be parents to the future nodes  
 BinaryTreeNode \*leftChild = NULL, \*rightChild = NULL;  
 leftChild = newBinaryTreeNode(head->data);  
 q.push(leftChild);  
 head = head->next;  
 if (head)  
 {  
 rightChild = newBinaryTreeNode(head->data);  
 q.push(rightChild);  
 head = head->next;  
 }  
  
 // 2.b) assign the left and right children of parent  
 parent->left = leftChild;  
 parent->right = rightChild;  
 }  
}  
  
// Utility function to traverse the binary tree after conversion  
void inorderTraversal(BinaryTreeNode\* root)  
{  
 if (root)  
 {  
 inorderTraversal( root->left );  
 cout << root->data << " ";  
 inorderTraversal( root->right );  
 }  
}  
  
// Driver program to test above functions  
int main()  
{  
 // create a linked list shown in above diagram  
 struct ListNode\* head = NULL;  
 push(&head, 36); /\* Last node of Linked List \*/  
 push(&head, 30);  
 push(&head, 25);  
 push(&head, 15);  
 push(&head, 12);  
 push(&head, 10); /\* First node of Linked List \*/  
  
 BinaryTreeNode \*root;  
 convertList2Binary(head, root);  
  
 cout << "Inorder Traversal of the constructed Binary Tree is: \n";  
 inorderTraversal(root);  
 return 0;  
}

Output:

Inorder Traversal of the constructed Binary Tree is:  
25 12 30 10 36 15

**Time Complexity:** Time complexity of the above solution is O(n) where n is the number of nodes.

This article is compiled by **Ravi Chandra Enaganti**. 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/given-linked-list-representation-of-complete-tree-convert-it-to-linked-representation/>

Category: [Trees](http://www.geeksforgeeks.org/category/tree/) Tags: [Queue](http://www.geeksforgeeks.org/tag/queue/)

# How to efficiently implement k Queues in a single array?

We have discussed [efficient implementation of k stack in an array](http://www.geeksforgeeks.org/efficiently-implement-k-stacks-single-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](http://www.geeksforgeeks.org/efficiently-implement-k-stacks-single-array/), 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/>