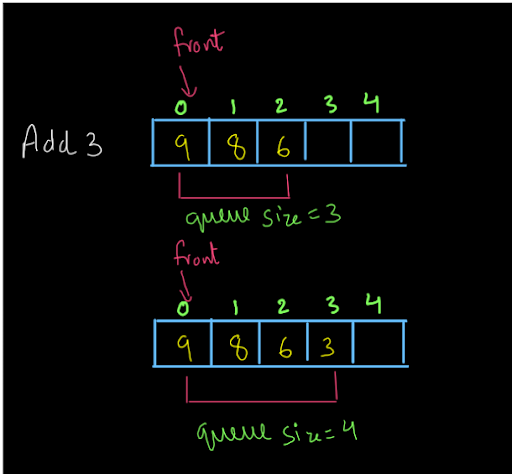
**1. Problem Discussion :**

We are given a linked list and we have to adapt it to make a queue. The previous problem was a LINKED LIST TO STACK ADAPTER. There, we used a linked list to make a stack and here we have to use it to make a queue. Note that the adapter does not mean that we are actually making the queue. It means that we are using the linked list in such a way that it functions like a queue. You may refer to the question video if you have any doubts regarding the question.

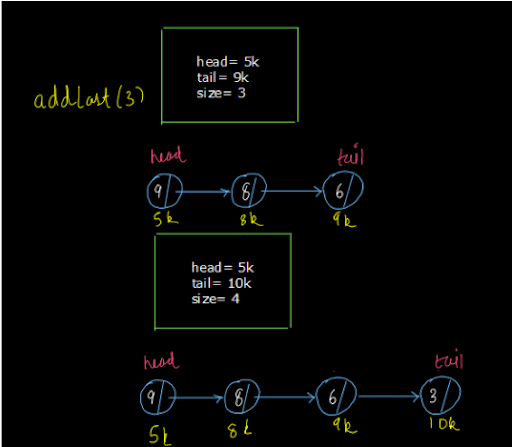
**2. Approach :**

Let us discuss all the functions that we have to write and see how we can adapt the queue from a linked list.

● Add: We know that queue is a data structure which is based on FIFO principle i.e. first-in-first-out. Whenever we add an element to a queue it is always added to the rear of the queue. (Have a look at fig-1)

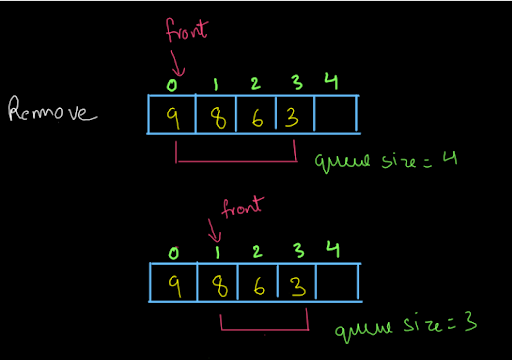
****

● Now, we want to add the data in a similar way into the linked list. So, adding the data to the end (tail) of the linked list is one option that looks exactly the same as adding the data to a queue. (Have a look at fig-2)

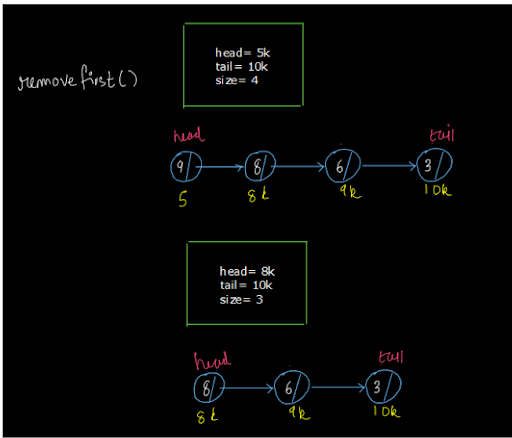
****

● As shown in the above diagram, adding a node to the end of the linked list looks like the same procedure as adding the element to a queue. So, we will use the addLast(val) function of the linked list data structure to add an element in the queue.

● Remove: Let us have a look at the remove method of a queue. Since the queue follows the FIFO principle, the element that came first will be removed first. The element at the front of the queue is the element that came first. So, the removal always happens from the front of the queue as shown in fig-3.

****

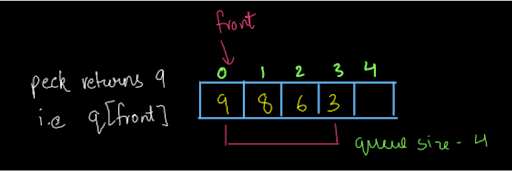
This is again, similar to removing the first element from the linked list i.e. removing the element at the head of the linked list. (see fig-4)

****

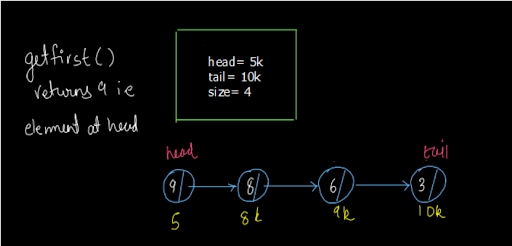
So, to remove an element from the queue, we will remove the first element from the linked list.

● Size: Think carefully. Instead of adding the elements to a queue, we are adding them to a linked list. Now, does it matter how we are adding the elements? No!! Whether we add the elements (nodes) to the linked list at the beginning, somewhere in the middle or in the end, since we are adding the elements to the linked list instead of adding them to a queue, the number of elements i.e. the count of the elements will not change. So, the size of the queue will be equal to the size of the linked list.

● Peek: The peek method in the queue returns the element at the front index of the queue.

****

The element at the front of the queue is the element at the head of the linked list. So, we can apply the getFirst() function of the linked list to get the element at the head of the linked list.

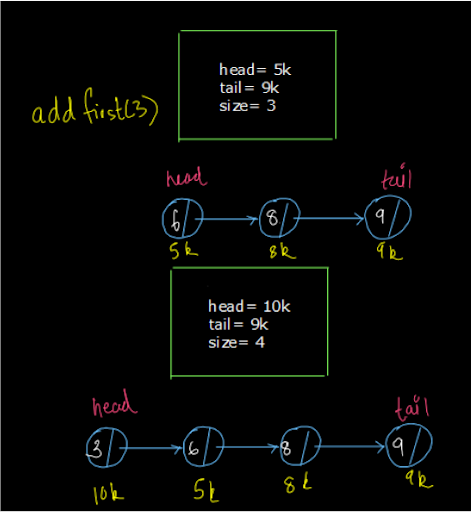
****

So, to apply the peek() method, we will use the getFirst() method of the linked list.

The Other Way Around:

Dear reader, we hope that you have understood the above procedure completely and have understood the analogy of a linked list and a queue. Now, it is time to rethink what we have done and find whether or not there is another way of doing what we have done above.

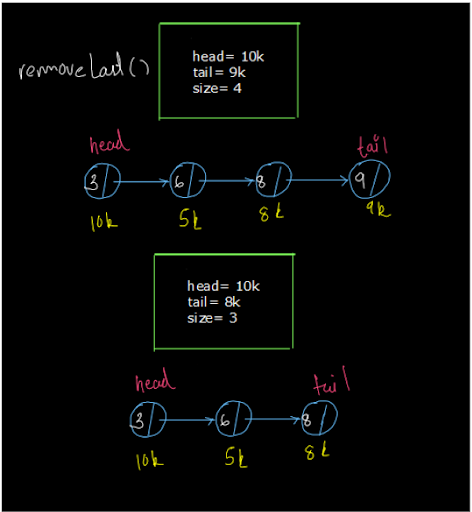
● Add: We said that adding a new element is analogous to adding a new node at the end of the linked list. Yes, it is, but is this the only way? If you think carefully, my friend, you will understand that there is no harm in inserting the node at the head of the linked list also. See, we just want to insert the element in a constant time as we used to do it in a queue and we want to follow the FIFO principle. It does not however, matter at all whether we choose the head of the linked list as the front of the queue or we choose it as the rear. Let us say we wanted to insert an element in the queue (see fig-1).

****

So, we can insert it at the head of the linked list. Note that every element prior to this is also inserted at the head of the linked list only. Inserting at head will also take constant time as inserting at tail in the linked list and as add(value) function of the queue.

● Remove: Okay, you know what is coming next, right? You are absolutely right dear reader, if we can reverse the add procedure we will also have to reverse the remove procedure. So, the remove() function always removes the data from the front of the queue and the tail of our linked list represents the front of the queue (since we added the elements to the head instead of tail). Therefore, we will have to remove the element from the tail of the linked list i.e. we will use the removeLast() function of the linked list to achieve this.

● Have a look at fig-3. We want to remove an element so that our remaining queue is now [8,6,3]. So, let us try to remove the node at the tail of the linked list. (see fig-9)

****

So, if we look at our linked list now from tail to head, it is the exact replica of our queue. Also, the deletion procedure that we perform will also be performed in a constant time. So, we have seen another method to remove an element from the queue.

Note: We said that the remove() procedure will take O(1) time. But, this is only for the Linked List available in Java not for the linked list that we make. As the linked list we make will take O(n) time for deletion from the tail. This is because, though we can access the tail node directly, we still have to traverse the entire list to modify the tail of the linked list after deletion. So, if we are not using the Java Linked

List then, it is preferred to make the adapter as we studied in the previous method i.e. head of the Linked List is the front of the queue and its tail is its rear.

● Size: This procedure will remain the same and we have discussed the reason behind it in the previous method only.

● Peek: The peek method returns the element at the front of the linked list. In this procedure, the tail of the linked list is the front of the queue. So, we will use the getLast() function of the linked list to peek into the queue.

So, dear reader, we have now understood both the methods as well as the limitation of the second method when not using the Java Linked List. Now it is the time for us to code the solution. You may refer to the solution video if you have any doubts regarding the procedure explained above. So, we will code the first method as it is the better one:

**3. Pseudo Code:**

1• Add: The add method adds an element to the end of the queue which is analogous to adding a node at the tail of the linked list. So, we'll do the same. 2• Remove: The remove function removes an element from the front of the linked list which is analogous to deleting at head in a linked list. So, to remove an element from the queue, we will remove the element from the head of the linked list. 3• Size: The size function returns the number of elements in the queue analogous to the size function of the linked list. So, we'll use the linked list size function. 4• Peek: It returns the data at the front of the linked list. To do that, we will return the data at the head of the linked list.

**4. CODE**

ConsoleCpp

#include <iostream>

using namespace std;

class node{

public:

int val;

node\* next;

};

class LinkedList {

public:

node\* head=nullptr;

node\* tail=nullptr;

int size\_=0;

void insert\_at\_tail(int val){

if(head==NULL){

node \*newnode = new node;

newnode->val=val;

newnode->next=NULL;

head=newnode;

}

else{

node \*newnode = new node;

newnode->val=val;

newnode->next=NULL;

node \*temp = head;

while(temp->next!=NULL){

temp=temp->next;

}

temp->next = newnode;

}

}

void insertion\_at\_head(int val){

node \*newnode = new node;

newnode->val=val;

newnode->next=NULL;

if(head==NULL){

head=newnode;

}

else{

newnode ->next =head;

head = newnode;

}

}

void print (){

node \*temp =head;

if(head==NULL){

cout << "0" << endl;

return;

}

while(temp!=NULL){

cout<<temp->val<<" ";

temp=temp->next;

}

cout << endl;

}

void deletion\_at\_head(){

if(head==NULL) return;

node \*temp=head;

head=head->next;

delete temp;

}

void deletion\_at\_tail(){

if(head==NULL) return;

node\* previous=NULL;

node\* temp=head;

while(temp->next!=NULL){

previous=temp;

temp=temp->next;

}

previous->next = NULL;

delete temp;

}

void last(){

node\* temp=head;

while(temp->next!=NULL){

temp=temp->next;

}

cout << temp->val << endl;

}

int size(){

int cnt=0;

node\* temp=head;

while(temp!=NULL){

temp=temp->next;

cnt++;

}

return cnt;

}

int first(){

return head->val ;

}

node\* getAt(int p){

int cnt=0;

node\* temp=head;

while(cnt < p){

cnt++;

temp=temp->next;

}

cout << temp->val << endl;

return temp;

}

void addAt(int pos,int data){

if(pos==0){

insertion\_at\_head(data);

return;

}

node\* newnode=new node;

newnode->val=data;

int cnt=0;

node\* temp=head;

while(cnt<pos-1){

cnt++;

temp=temp->next;

}

newnode->next=temp->next;

temp->next=newnode;

}

void removeAt(int pos){

if(pos==0){

deletion\_at\_head();

return;

}

int cnt=0;

node\* temp=head;

while(cnt<pos-1){

cnt++;

temp=temp->next;

}

temp->next = temp->next->next;

}

void reverse\_di(){

int left = 0;

int right = size() - 1;

while(left < right){

node\* templ = getAt( left);

node\* tempr = getAt( right);

int temp = templ->val;

templ->val = tempr->val;

tempr->val = temp;

left++;

right--;

}

}

void reverse\_pi(){

if(size()<=1){

return;

}

node\* t=head;

while(t->next!=NULL){

t=t->next;

}

tail=t;

node\* prev=nullptr;

node\* curr=head;

while(curr !=nullptr){

node\* next= curr->next;

curr->next=prev;

prev=curr;

curr=next;

}

node\* temp=head;

head =tail;

tail=temp;

}

};

class LLToStackAdapter{

public:

LinkedList l1;

int size1(){

// write your code here

}

void push(int val){

// write your code here

}

int pop(){

// write your code here

}

int top(){

// write your code here

}

};

int main() {

LLToStackAdapter l1;

string s;

cin >> s;

while(s!="quit"){

if(s == "push"){

int val;

cin >> val;

l1.push(val);

}

else if(s == "pop"){

int val= l1.pop();

if (val != -1){

cout << val << endl;

}

}

else if(s== "top"){

int val= l1.top();

if(val != -1){

cout << val << endl;

}

}else if(s== "size"){

cout << l1.size1() << endl;

}

cin >> s;

}

}

If you have any doubts regarding the code written above, you may refer to the solution video to clear all your doubts. Now, let us analyze the time and space complexity of each of these methods.

**5. Analysis:**

Time Complexity:

1• Add: The time complexity of this method will be O(1) whether we follow the approach of adding an element at the tail or at the head as we are not traversing the linked list in any case. 2• Remove: The time complexity of remove is also O(1) as removing an element from head or from tail takes constant time and there is no traversal of linked lists in both of these cases as well. But, if we don't use the Java Linked List, then the method of removing an element from the tail will take O(n) time. That is why the second approach that we discussed is not preferred. 3• Size: The time complexity of this method is O(1) as we just have to return the size of the linked list and we don't have to traverse it. 4• Peek: The time complexity of this method is also O(1) as we don't have to traverse the linked and directly return the data of the head (in the first approach) or the data of the tail (in the second approach).

Space Complexity:

The space complexity for all the above methods is O(1) as we have not used any extra memory to implement these methods.

So dear reader, we hope that you have understood the entire solution along with code and the time and space complexity analysis. If you still have any doubts in your mind, you may refer to the complete solution video to clear them.

**6. Suggestions:**

Here are some suggestions from our side that you do not want to miss:

1• We suggest you refer to the STACK TO QUEUE ADAPTER-ADD EFFICIENT and the STACK TO QUEUE ADAPTER-REMOVE EFFICIENT problems. In these problems we were only able to optimize one method of the queue, either add or remove but we were not able to optimize both at the same time. This is something we have achieved using the linked list data structure. 2• We also suggest you refer to the previous question, LINKED LIST TO STACK ADAPTER as well and analyze then visit QUEUE TO STACK ADAPTER-PUSH EFFICIENT and QUEUE TO STACK ADAPTER-POP EFFICIENT. Again, you can see that we were not able to make both the push and pop function optimized at the same time. 3• Now, let us give you a hint and we suggest you meditate on this fact. We were not able to make both the methods optimized when we made the stack to queue adapter or the queue to stack adapter because stack is LIFO and queue is FIFO and it is difficult to achieve either one of them from the other. But, a linked list has both LIFO and FIFO properties i.e. it is not that specific. So, it was easier to make the adapters efficient using the linked list. Think about this!!!! Until then, Happy Coding!!!!!