Lists, Stacks and Queues (Theory)

BASIC DATA STRUCTRES

PRAKASH HEGADE

SCHOOL OF CSE | KLE Tech

Stacks

If you are a fast food lover and the title reminded you of lay's stax potato chips, then you have almost understood the concept already. If you look out for the standard dictionary meaning for the word stack, you would end up reading 'a pile of objects, typically one that is neatly arranged'. Well, that is it! You comprehend the concept already. What remains is the mathematical perception and how a computer engineer sees it all.

Introduction

Let us consider an array with the name 'array' of size 'n'. You can access any element of the array by specifying the appropriate index. Visualize the picture mentally and this is how probably it should look like:

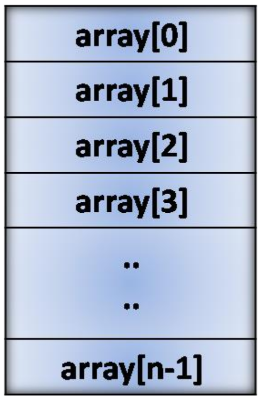


Fig: An array of size n

Let us now understand how the array becomes the stack. Look at the same array and apply the following conditions:

- Seal the three sides of the array by leaving the top or bottom end open
    - Say the top end is left open
    - To be more specific, we can only see what is at the top of the array
- Imagine an index which will allow us the access of top most element of stack
    - Say the index is named as 'top'

Visualize all the conditions on the above figure. It would be a lot better if you draw the new diagram by applying above conditions in your rough work and turn to the next page. I hope your thought process matches the figure.

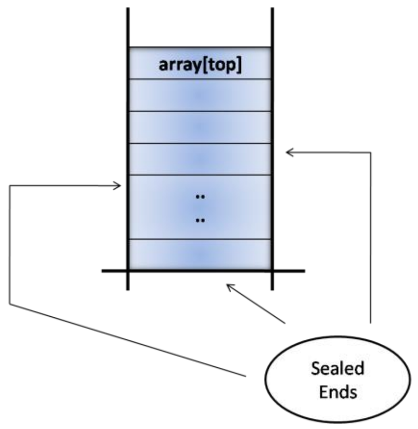


Fig: A Stack

With this understanding, let us note down the basic properties of stack:

- We can insert a new item only on top of the stack.
- We can delete an existing item only from top of the stack.
- We can only see what is at the top of the stack.

Stack represents a class of problems where the operations happen at one end. Let us try to identify some real world examples which exhibit the same behavior as stack. Following are a few:

- Parle Poppins
- Sequence of dreams in the movie Inception
- Bangles worn in hand
- Rings put inside the well
- Disks placed one above the other in disk stand

Can you think of more examples?

Definition

Before we formally study the definition, let us fix with the naming convention for the operations.

Si. No.	**Name**	**Description of the Operation**
1.	PUSH	Insert an item into the stack.
2.	POP	Delete an item from the stack.
3.	PEEK	Look out for the top most item from the stack.
4.	UNDERFLOW	Stack is empty. PEEK / POP are not allowed.
5.	OVERFLOW	Stack is full. PUSH is not allowed.

Table: Stack Operations

Implementation Details
Given the definition, we will now understand the design to carry out the implementation. The design will be done step wise slowly building up the functions one by one.

Problem
Implement a stack which can hold a maximum of 100 integer values. Use the following structure for the stack:

```
#define STACKSIZE 100
struct stack {
        int top;
        int items[STACKSIZE];
};
typedef struct stack STACK;
```

Support the implementation with the Push, Pop and Peek functions. Also write a Print function to print all the contents of the stack. The integer items, logically, have to be displayed from the top to zero indexes.

Give a user menu in the main() so that user can select the required operation to perform. Print meaningful messages after every operation.

Solution:
Let us first visualize the structure that is given to us with appropriate initializations.

Let us say for the given structure,

```
#define STACKSIZE 100
struct stack
{
  int top;
  int items[STACKSIZE];
};
typedef struct stack STACK;
```

We will create a variable of type STACK and initialize the top value.

```
STACK s;
s.top = -1
```

In memory, this would look something like following:

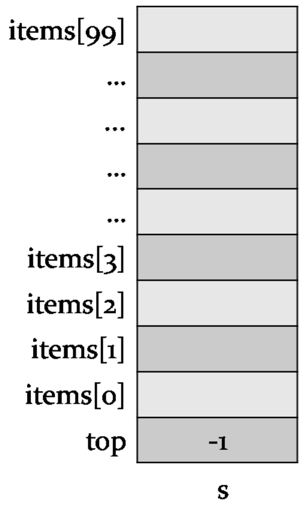


Fig: Memory layout for initialized stack

We have initialized the top to -1. Now what would be the stack memory layout for 'Underflow' condition? Can you visualize that in your mind? The state is in underflow when there are no more items left in stack to pop. The stack is empty with 'n' number of push and 'n' number of pop. Basically, this is same as figure above.

Let us now imagine the 'Overflow' state. It is overflow when there is no more room for push. The memory layout would look as following as shown in figure below.

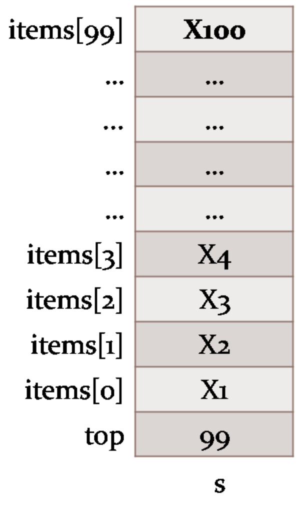


Fig: Memory layout for Overflow condition

Let us now understand the push operation. Top is initialized to -1, so first we need to increment the top and then push an item x into the stack. The steps are explained in the figure below.

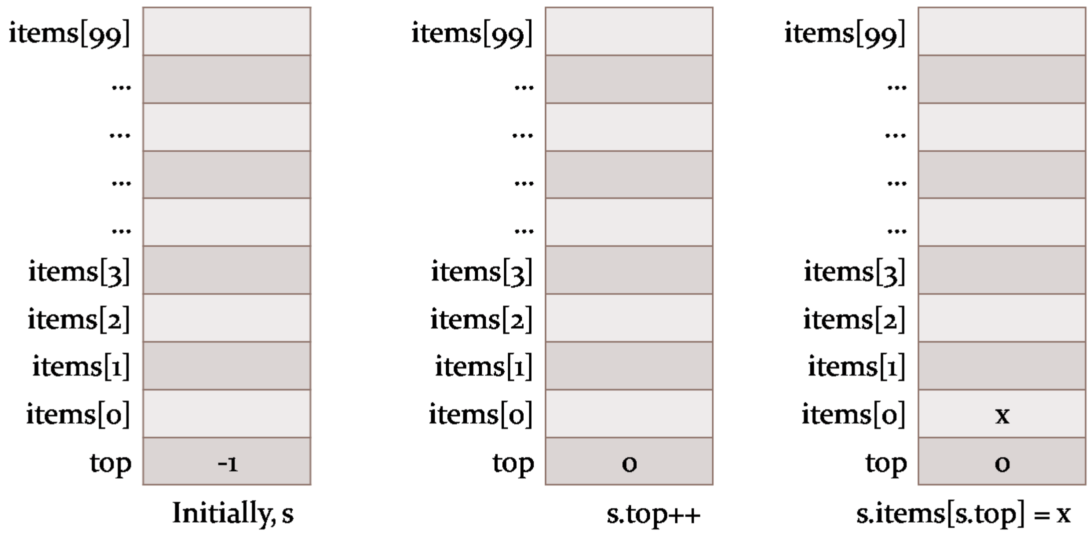


Fig: Memory layout for Push Operation

From the above figure we can note that push operation is made up of two simple steps.

PUSH

- Increment the top → s.top++
- Copy the item → s.items[s.top] = x

Similarly let us also understand the pop operation. In pop, we first copy the item and then decrement the top. Essentially, the operation is reverse of push operation. Figure below explains the steps.

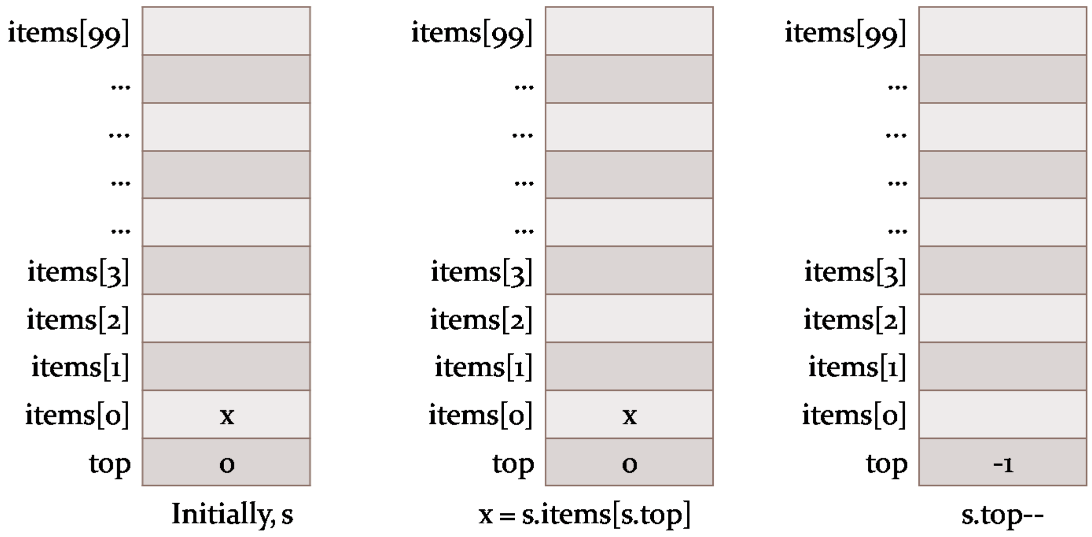


Fig: Memory layout for Pop Operation

The way we defined Push, we can define Pop operation in two steps as:

POP

- Copy the item to temporary variable x → x = s.items[s.top]
- Decrement the top → s.top--

Peek operation is accessing the top most items in the stack.

If 'x' is supposed to hold the result of PEEK operation, which of the following gets it right?

a. x = POP();
b. x = POP(); PUSH(x);
c. PUSH(x); x = POP();
d. Option a. and b. both

Answer:
Option b. is right. Peek operation does not remove the item from stack. It only returns it.

We now have sufficient knowledge to implement the stack.

Queues

Introduction
Real time examples for queues

- Standing in queue to book movie tickets
- Standing in queue to get the tickets/reservation from the ticket counter
- Standing in a queue to get the food in hostel
- Queue the movies/songs in a player

Properties

- Ordered collection of items in which items can be added at one end and deleted from another end
- General convention:
    - Deletion end – front
    - Insertion end – rear
- Order of the Queue is: **First in First out - FIFO (or LILO)**
- Primitive Operations:
    - Insert, remove, and empty

- Insert – operation can always be performed, since there is no limit on number of elements a queue may contain
- Remove – can be applied only when the queue is not empty
- The conventional names used are Enqueue, Dequeue and Display
- The result of an illegal attempt to remove an element from an empty queue is called **underflow**

Implementation Details
We will need three variables:
front and rear → two integer variables
items → array to hold the queue elements

Defining the structure considering above details we have,

```
#define MAXQUEUE  100
struct queue
{
        int front;
        int rear;
        int items[MAXQUEUE];
} q ;
```

Operations:
Initializations: [Done in main( )]

```
q.rear = -1;          // The Insertion end – increment and insert
q.front = 0;          // The deletion end – delete and increment
```

Empty:
whenever q.rear < q.front

Note: How did we arrive at this condition?

Si. No.	Queue is empty when	rear value	front vale
1	Initial condition	-1	0
2	Insert 2 elements and delete 2 elements	1	2
3	Insert 4 elements and delete 4 elements	3	4
4	Insert n elements and delete n elements	n-1	n

Hence we can conclude that queue will be empty whenever rear < front index

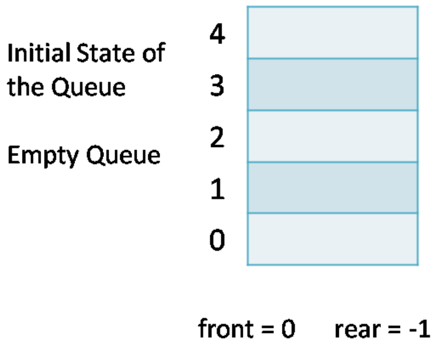
Full:
Queue will be full when there is no more space left for insertion. As the insertion happens at rear end, we can set the full condition as whenever the rear reaches its maximum index value. Hence queue will be full when
rear = MAXQUEUE-1

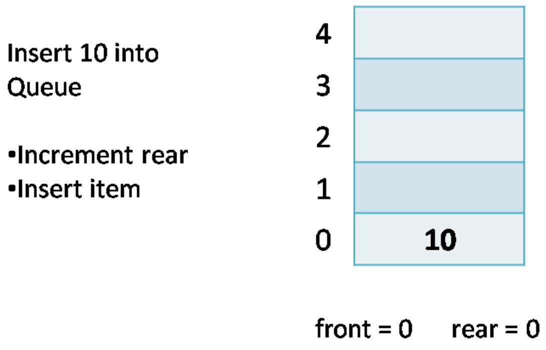
Insert Operation:
insert(q, x)→ Increment rear and insert
- (q.rear)++;
- q.items[q.rear] = x;

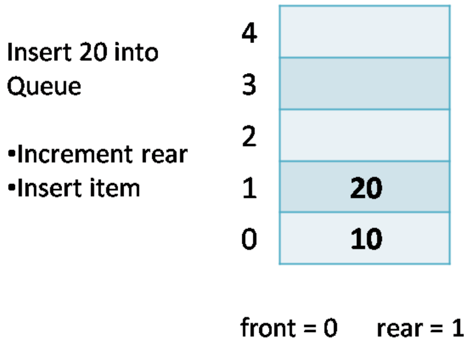
Remove Operation:
x = remove(q) → remove and increment front
- x= q.items[q.front];
- (q.front) ++;

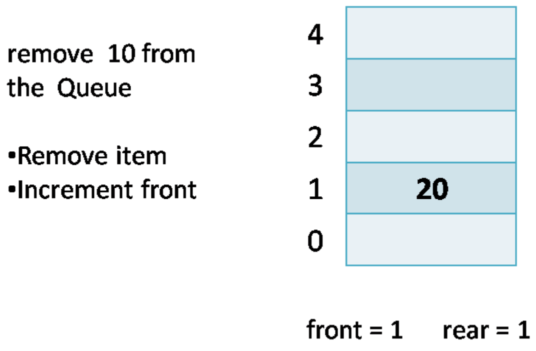
How do you find out the number of elements in a queue?
The number of elements in the queue at any given time is equal to the value of
Number_of_queue_elemets = q.rear – q.front + 1

Queue Sample Operations









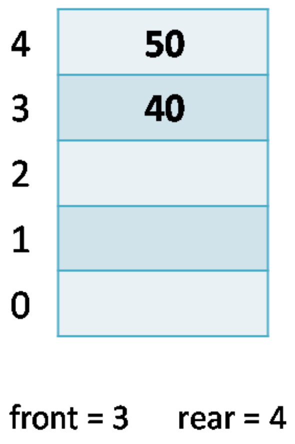
Circular Queue

Even though there is room for new elements in linear queue, at certain conditions we print the message queue is full.

Consider for example, When the Queue size is 5.

Insert 5 items into the Queue and delete 3 items.

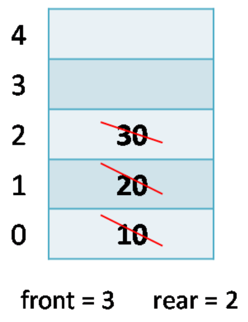
Even though there is space for 3 elements, as the rear has reached MAXQUEUE -1 we say Queue is full

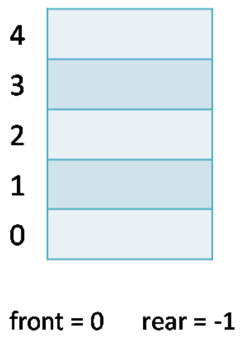


Improvement:

As an improvement for Linear Queue, whenever front index becomes greater than rear index we can reset back front and rear values to initial values.

```
if(q->front > q->rear)
{
        q->front = 0;
        q->rear = -1;
}
```





3 items are inserted and all the 3 are deleted. In such cases Queue is empty. Reset front and rear to initial values.

This will make some space for insertions.

What else can we also do?

We can shift all the values one down and make a room for new item which is coming in. However this becomes a costly operation when the size of the queue is very large.

Another possible alternate is to loop back to start of the Queue and continue with insertion again. This defines a **Circular Queue**.

In order to achieve this we make the following changes to the increment operations of rear and front.
cq->rear = (cq->rear+1) % MAXQUEUE;
cq->front = (cq->front+1) % MAXQUEUE;

Example:
If rear is MAXQUEUE-1, say 4 then,
rear = (4 + 1) % 5 = 5 % 5 = 0.
So the next insertion happens at position 0.

The logic now changes to
- increment rear and insert and
- increment front and delete

And we always want first insertion to happen at position 0. So we initialize rear and front of the queue with values:
- rear = MAXQUEUE – 1
- front = MAXQUEUE - 1

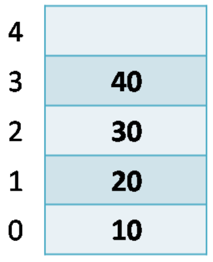
Now with the set initial condition, both empty and full conditions are going to be the same which is front == rear which is not valid.

We keep it as empty condition and design a new condition for full.

According to new design at any given point of time when we say Queue is full, one of the memory locations in items array is left unused.

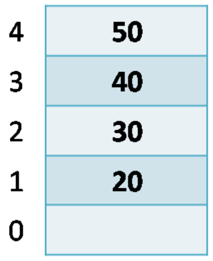
The design of the full condition can be seen in the page next.

front == rear + 1 % MAXQUEUE
4 = (3 + 1) % 5
= 4 % 5
= 4

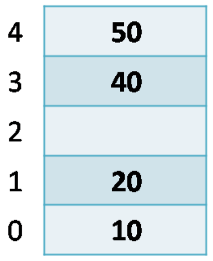


front = 4 rear = 3

front == rear + 1 % MAXQUEUE
0 = (4 + 1) % 5
= 5 % 5
= 0



front = 0 rear = 4

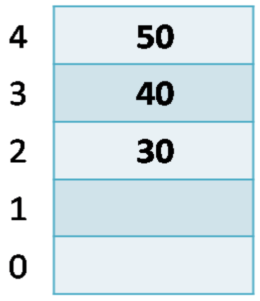


front = 2 rear = 1

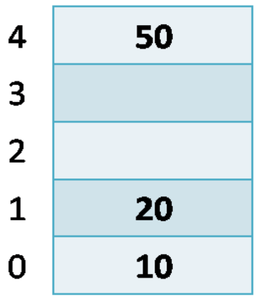
front == rear + 1 % MAXQUEUE
2 = (1 + 1) % 5
= 2 % 5
= 2

Above are examples of Queue full. The location pointed by front is always left unused. So when ever front == (rear + 1) % MAXQUEUE we say that Queue is full.

Analysis for Display Function of Circular Queue:



front = 1 rear = 4



front = 3 rear = 1

We have 2 conditions and shown above.
If rear > front,
Print all from front + 1 to rear

If front > rear,
Print all from front + 1 to MAXQUEUE-1 and
O index to rear

There is also another simple way. You must have already figured that out!

DECK (Double Ended Queue)
A Double ended Queue is one where insertions or deletions can happen over both the front as well as rear ends. We can have linear Queue as DECK as well as Circular Queue as DECK. (Refer class notes for more analysis)

Linear Queue as Double Ended Queue

Rear = -1	**Front = 0**
Increment and Insert	Remove and Increment
Remove and Decrement	Decrement and Insert

Circular Queue as DECK

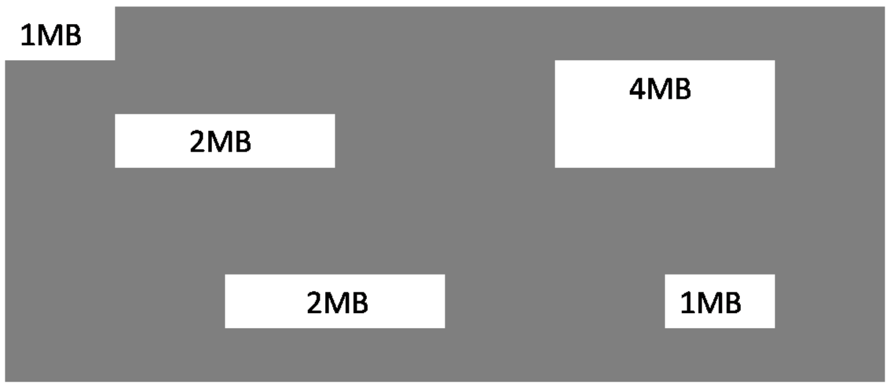
Rear = MAXQUEUE-1	**Front = MAXQUEUE-1**
Increment and Insert	Insert and Decrement
Remove and Decrement	Increment and Remove

Lists

Introduction
The data structures we have seen so far have

- Fixed amount of storage (we have used arrays and static memory allocation)
- Have implicit ordering
    - If x is current location then x+1 gives the next location

What we shall see next is rather an alternate representation of the data. Not all the times the system can guarantee the implicit ordering. Let us say we requested for 10MB of memory and following is the memory status:

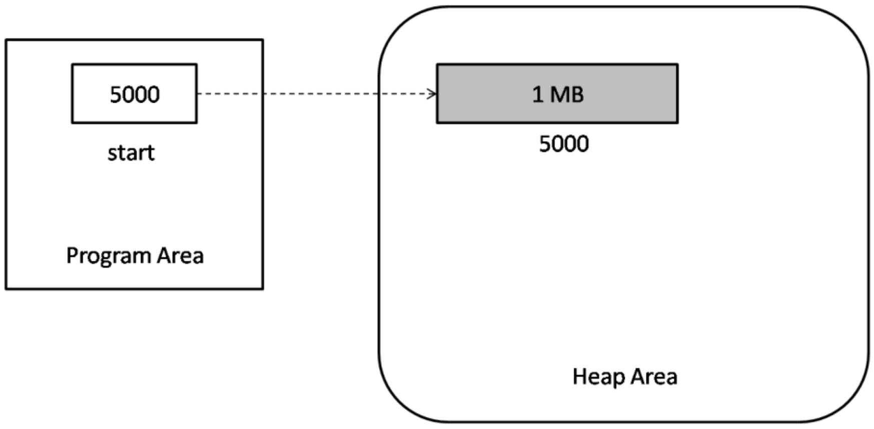


Memory Snapshot

As you see, we do have 10MB of memory. But they are not contagious. We don't want system to tell us that there is no memory available just because it is not contiguous. Instead we think of an alternative of how we can use this available space effectively.

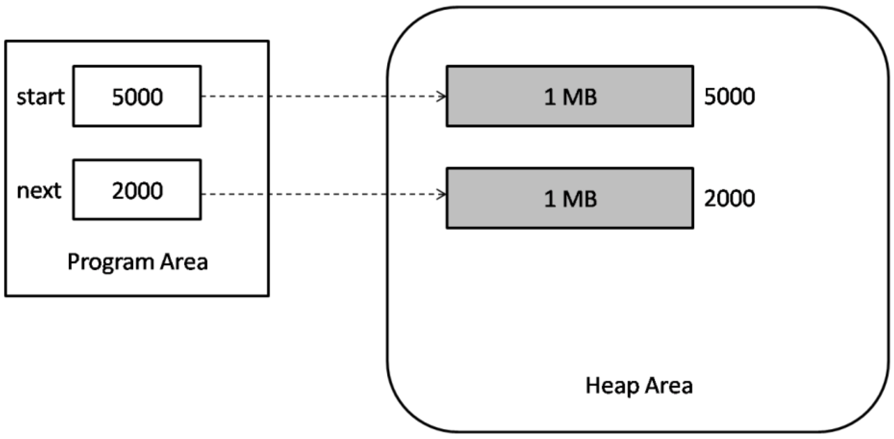
Now we know that we don't need all this 10MB space at once all at once. We might need 1MB at a time. That puts us with the question, rather than asking for 10MB at once, can I ask 1MB at once for 10 times? Yes. We can. We shall do that. Let us understand the process with a little more of technical detail. Let us first ask for 1MB of data. Remember, 1 MB is huge for the textual data we are dealing with. It is only an example to understand. Otherwise all our operations are going to end up with few bytes.

The program makes a request for 1MB of data. Memory is allocated from the heap. Heap returns the starting address of this 1MB of data so that the program can make access to it. Essentially, program should have a pointer to hold this address. Let us diagrammatically see how all this looks like.



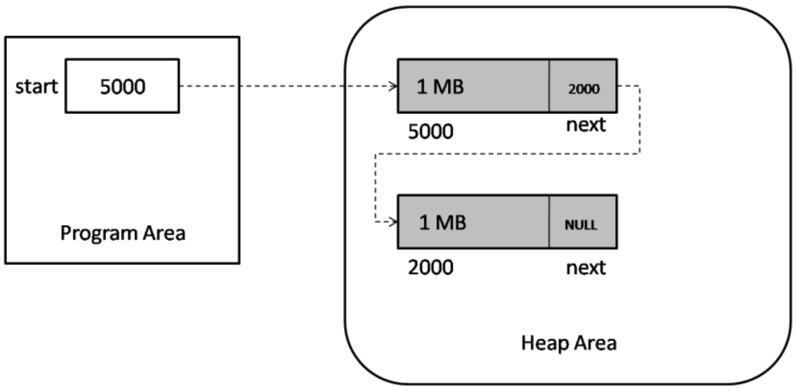
Memory Allocation of First Chunk

We needed 1MB of data and we have it in heap. The starting address of this 1MB is 5000 which we have captured through a pointer variable 'start'. We are all good and we go ahead with our program and suddenly at one fine time we run out of memory. We need 1MB more. We make a request again to heap and we get the memory allocated.



Memory Allocation of Second Chunk

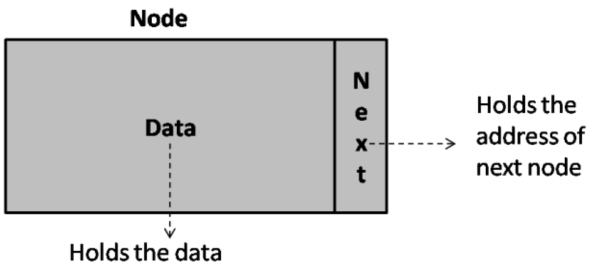
Okay, this looks fine. We now have one more pointer called 'next' to hold the starting address of the next allocated chunk. But wait, there is a suggestion. Instead of growing up the number of pointers in the program for each allocated chuck, we can do the following way:



Alternate Design of Memory Allocation

This looks more promising. Our first 1MB data holds the address of where the next 1MB is and this process will continue ahead for the further chunks of memory allocated. Every memory allocated chunk has a variable which holds the address of where the next chuck is.

Each chuck will be officially named as 'Node' as looks like the following:



Structure of a Node

What we are going to have is a chain of nodes all linked together. Thus what we have is a Linked list. Get ready for this exciting and challenging journey. We are going to get a little dirty here. We are going to get messed up with pointers, more errors and more system crashes and yes, more knowledge. This is a turn where you will have a lot of interview questions and lot of applications in industry as well.

Understanding a Node

In this section we shall understand the technical details of a node and creating a code.

Here is how the **node** goes. We have a structure which houses the data and the pointer to next node. It's a self referential structure and it has a reference of its own type. What we mean is the next field holds the address of a node whose data type is same as itself AKA self referential.

For now, let us assume we have integer data.

```
struct node
{
   int data;
   struct node *next;
};
typedef struct node NODE;
```

And the **memory allocation**:

```
NODE * newnode;
newnode = (NODE *) malloc( sizeof(NODE) );
```

Operations

We have a node with memory allocated. What kind of operations can we do?

- Add a new node
    - Where? Front? End? Or somewhere at required position?
- Delete a node
    - From where? Front? End? Or somewhere from required position?

Let us look at all the operations in detail one by one. Before we start with any, let us have few initial ground works done.

We have the structure:

```
struct node
{
   int data;
   struct node *next;
};
typedef struct node NODE;
```

Let us create a variable to hold the starting address of the list. Let us call it start and initialize it to NULL.

```
NODE *start = NULL;
```

Pop Quiz

Consider the structure given below (integer is 4 bytes):

```
struct node {
   int data;
   struct node *next;
};
struct node *newnode;
```

What is the memory allocated for the variable 'newnode'?

a. 8 bytes

b. 4 bytes

c. 0 bytes
d. 10 bytes
Answer: 4 bytes. 'newnode' is a pointer. Pointer stores address and requires integer number of bytes.

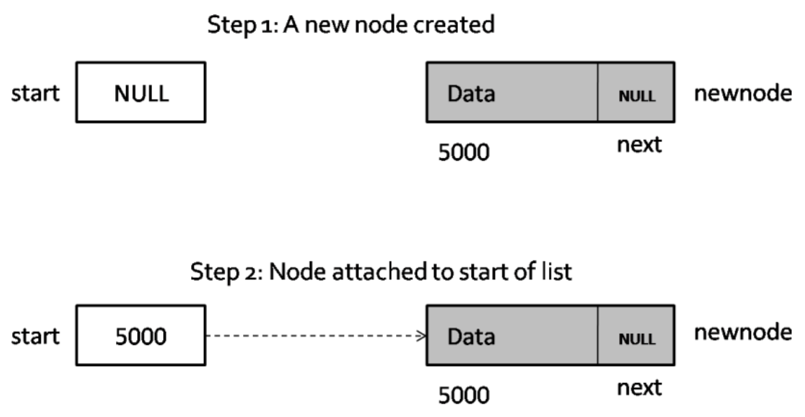
Adding a New Node

Like already said a new node can be added at following positions

- Front
- End
- Required position

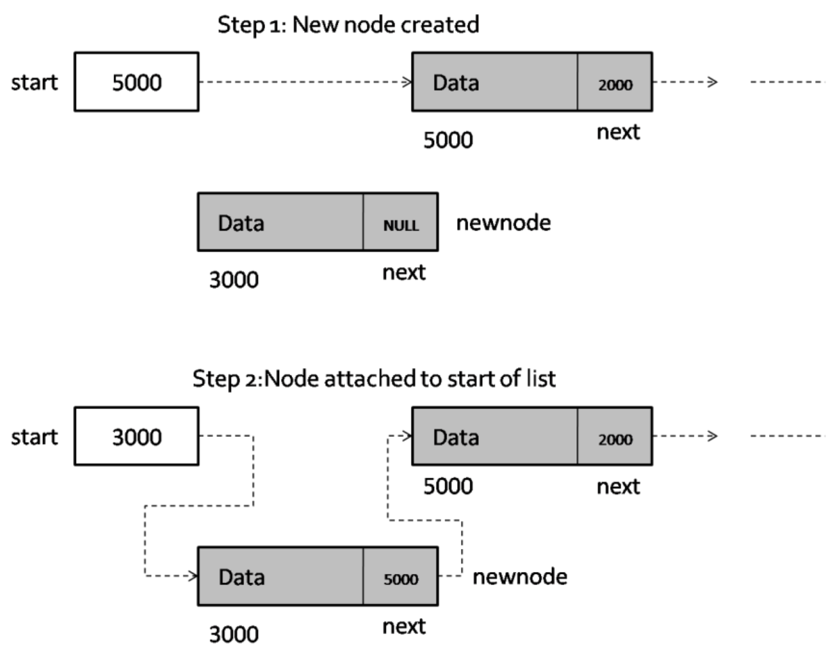
Adding a node at the front:

We have two cases. What if the list is empty? In case we assign the node address to the start pointer.


Step 1: A new node created
start
NULL
Data
NULL
newnode
5000
next
Step 2: Node attached to start of list
start
5000
Data
NULL
newnode
5000
next

Adding a Node at Front – Case 1

If the list is not empty, we do the following:

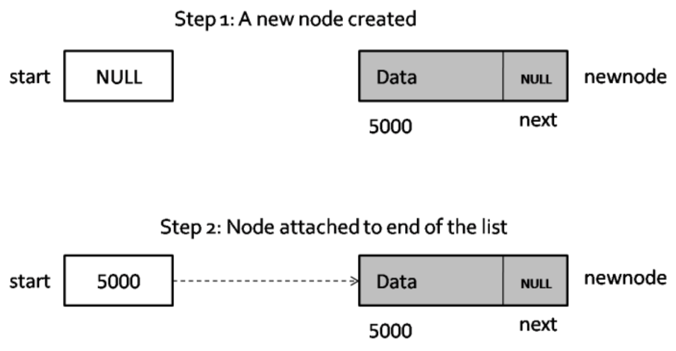


Adding a Node at Front – Case 2

Note how the pointer adjustments are done. We have two updates to make. One is the 'next' field of newly created node and other is the 'start' pointer.

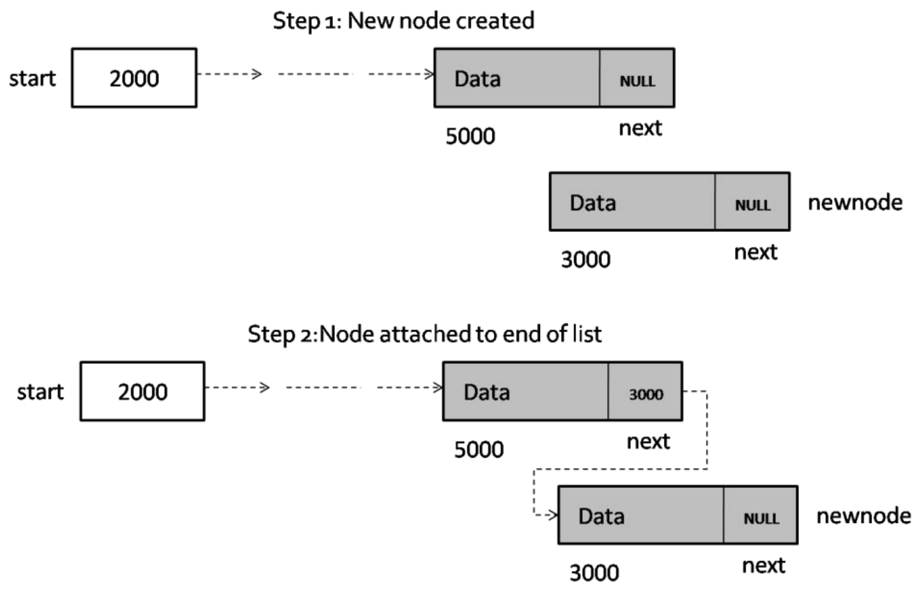
Adding a node at the end:

We have two cases. What if the list is empty? In case we assign the node address to the start pointer. This is same as the case of front when the list is empty. As the list is empty, the only node we insert is the front and is the end.


Step 1: A new node created
start
NULL
Data
NULL
newnode
5000
next
Step 2: Node attached to end of the list
start
5000
Data
NULL
newnode
5000
next

Adding a Node at End – Case 1

If the list is not empty, we do the following:

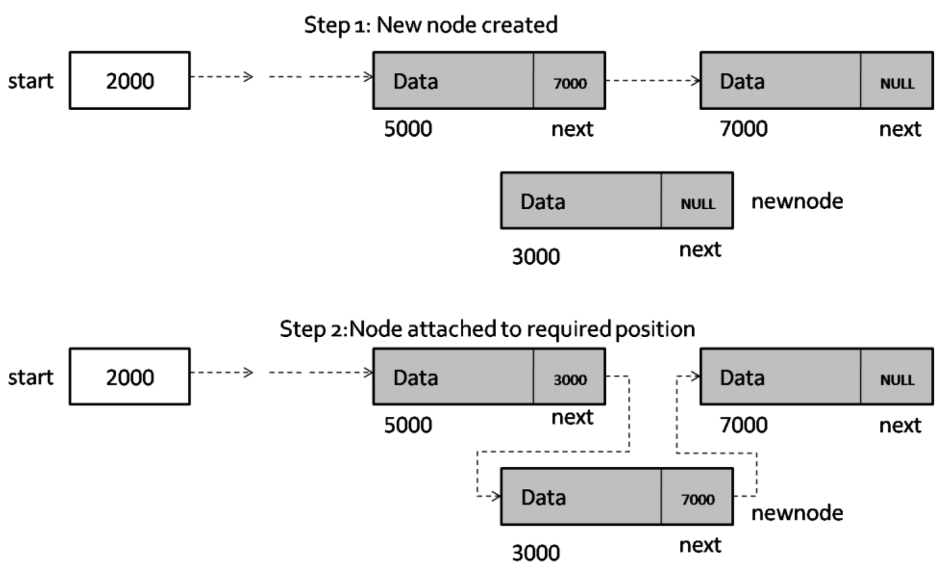


Adding a Node at End – Case 2

There are 'N' numbers of nodes and node with address 5000 is the last end. We attach the newly created node to end of this list. Look at how the 'next' field of node with address 5000 changes from 'NULL' to starting address '3000' which is of the newly created node.

Adding a node at the any position:

Any position can also be for first or last. We will skip those cases as we have already handled them. We shall the case of adding the node at nth position. Look at the figure below. The pointer adjustments are quite intuitive.



Adding a Node at position n

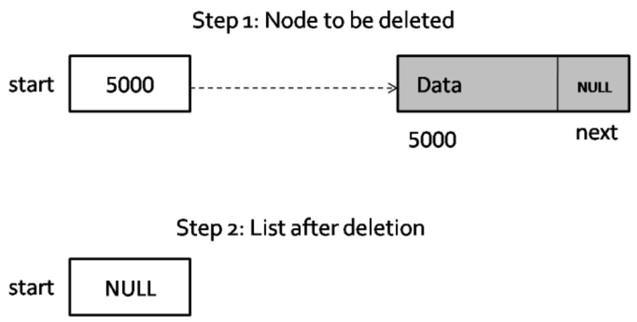
Deleting an Existing Node

Like already said an existing node can be deleted from following positions

- Front
- End
- Required position

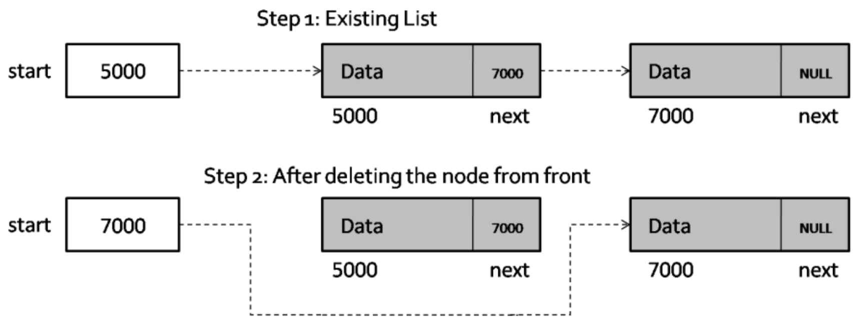
Deleting a node from front:

We have two cases. What if it is the last node to be deleted and what if there are many other nodes. We shall handle both the cases.



Deleting a Node from Front – Case 1

If the list has more than one node, we do the following pointer adjustments:

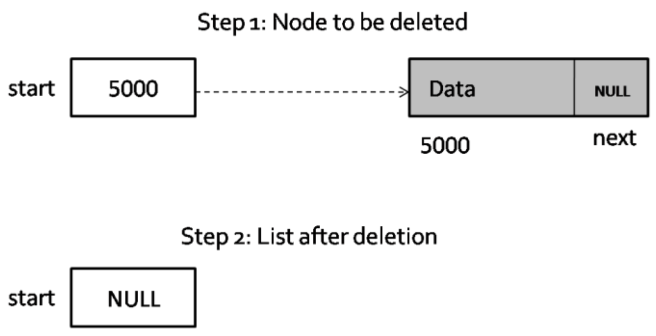


Deleting a Node from Front – Case 2

As you see, the start pointer is updated to the next node address. The next address can be obtained from the 'next' field of the first node.

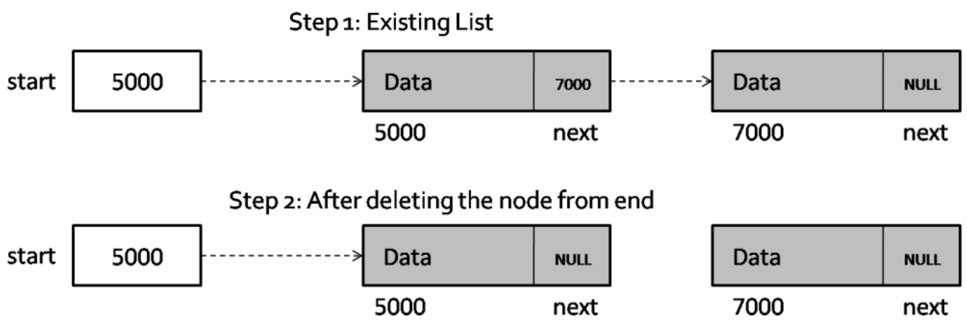
Deleting a node from end:

We have two cases. What if it is the last node to be deleted and what if there are many other nodes. We shall handle both the cases.



Deleting a Node from end – Case 1

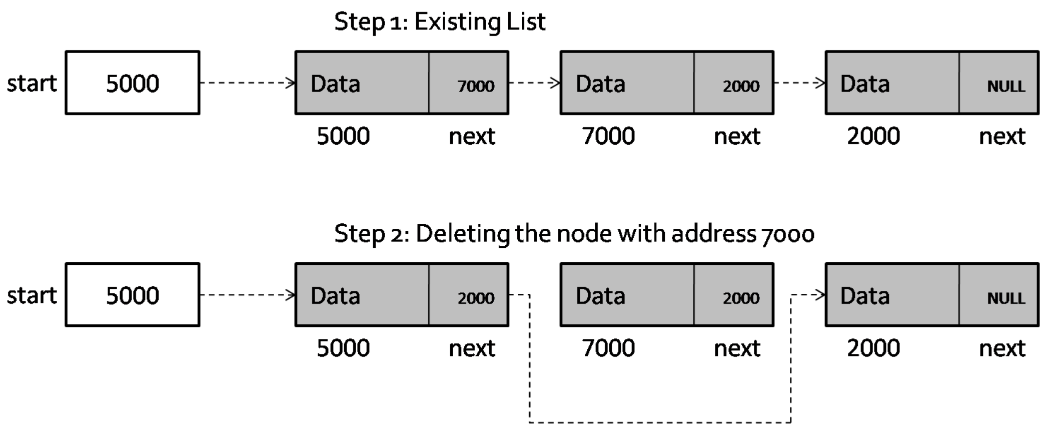
As you notice this case is same as the case for delete from front. As there is only one node front and end refer to same node. Let us look at the other case.



Deleting a Node from end – Case 2

Deleting a node from any position:

Any position can also be for first or last. We will skip those cases as we have already handled them. We shall look at the case of deleting a node from nth position. Look at the figure below. The pointer adjustments are quite intuitive.



Deleting a Node from any position

The pointer adjustments are evitable from the diagram. Our next idea is to convert all these diagrams to the ‘C’ code.

Linked Stack and Linked Queue

A linked stack can be implemented by providing following operations

a) Insert at front and
 Delete from front

b) Insert at end and
 Delete from end

A linked queue can be implemented by providing following operations

c) Insert at front and
 Delete from end

d) Insert at end and
 Delete from front

~*~*~*~*~*~*~