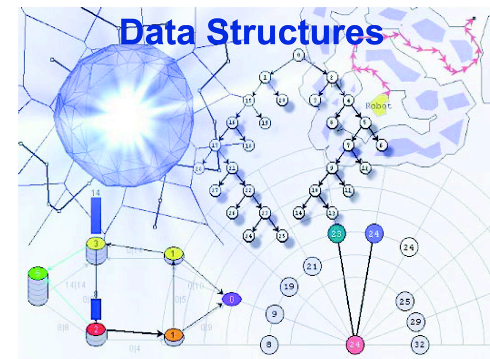
BBM 201
DATA STRUCTURES

Lecture 6:
Stacks and Queues

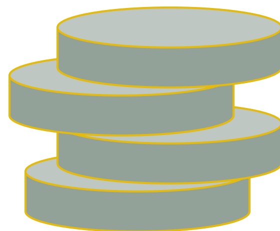
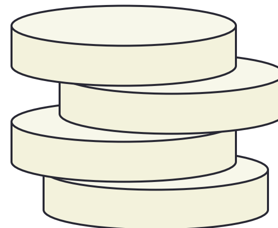
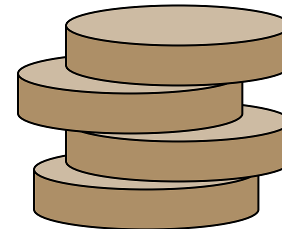


BUS STOP
6 9
13 15 60
96 291 294 297

Stacks

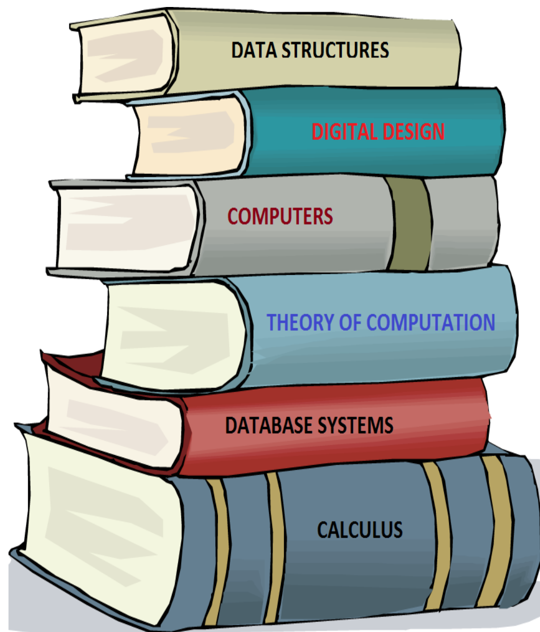
- A list on which insertion and deletion can be performed.
  - **Based on Last-in-First-out (LIFO)**
- Stacks are used for a number of applications:
  - Converting a decimal number into binary
  - Program execution
  - Parsing
  - Evaluating postfix expressions
  - Towers of Hanoi

  …

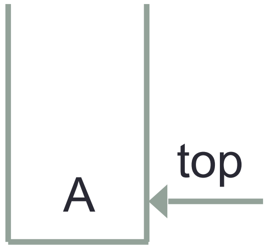




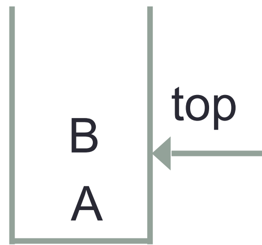
Stacks

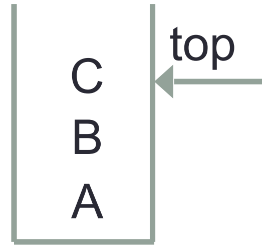
A stack is an ordered lists in which insertions and deletions are made at one end called the ***top***.

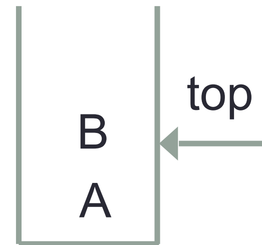


Stacks

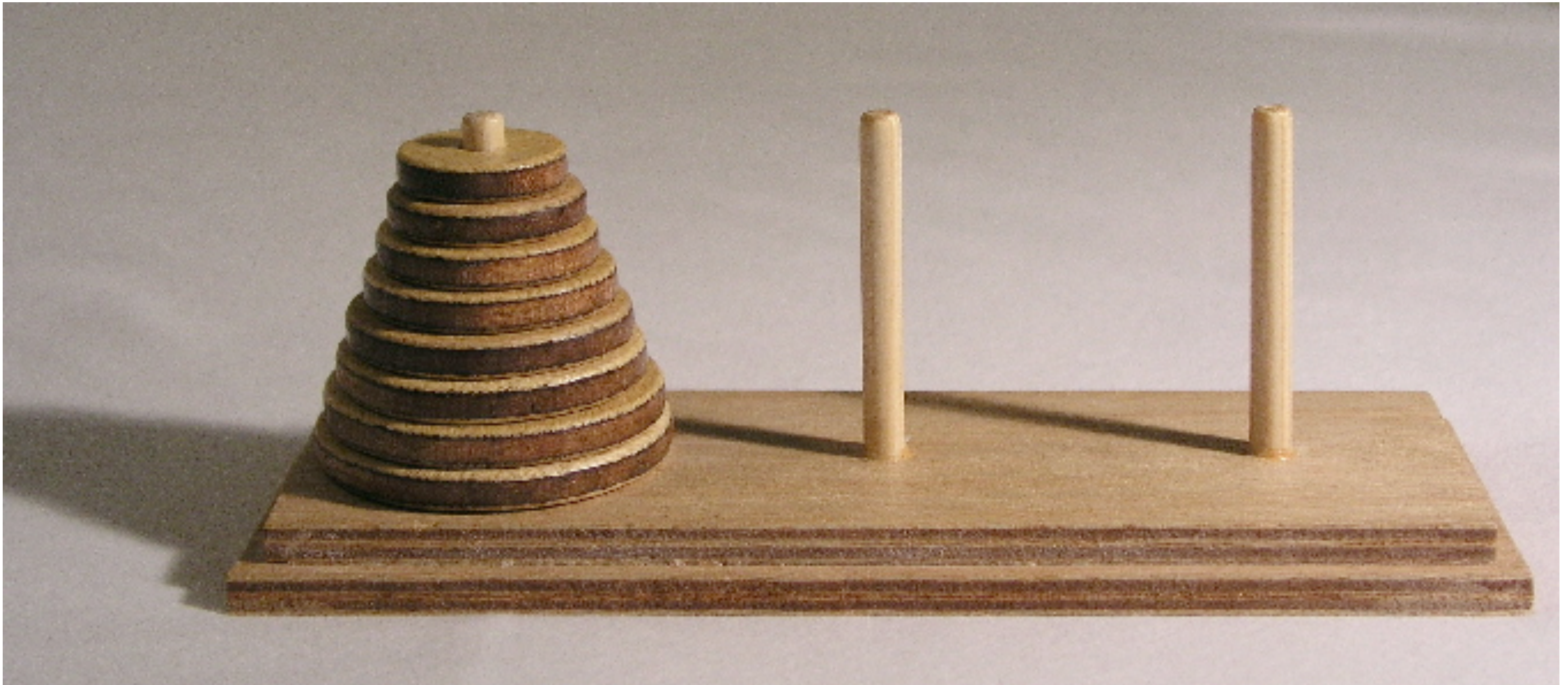






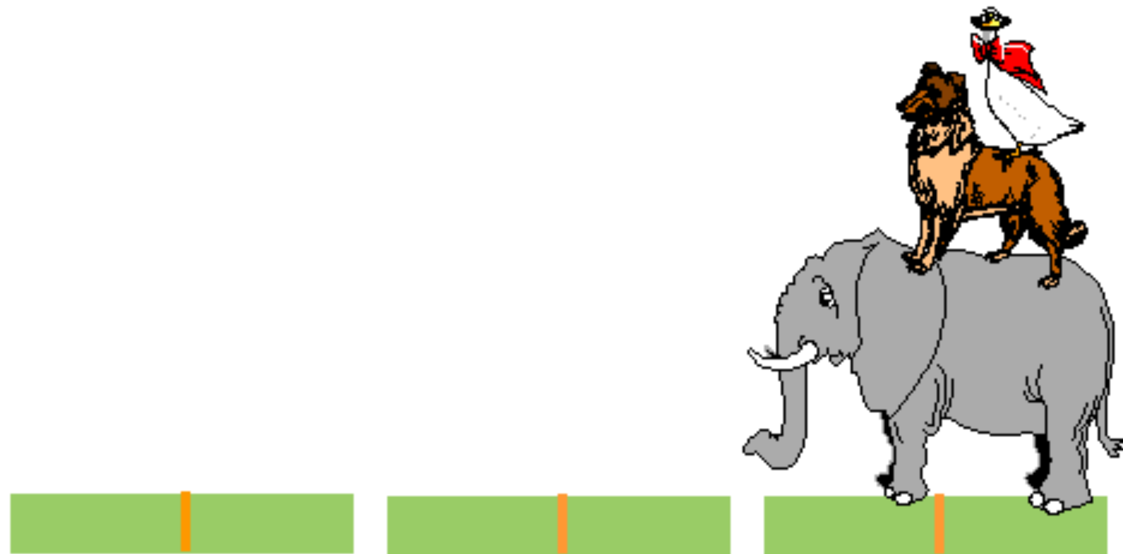


Towers of Hanoi

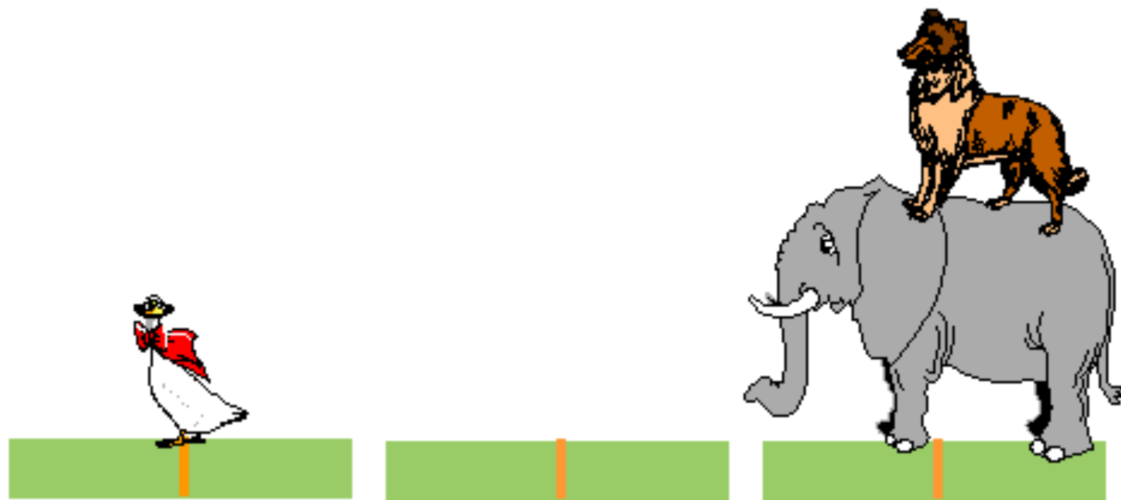


Object of the game is to move all the disks (animals) over to Tower 3.
But you cannot place a larger disk onto a smaller disk.

Towers of Hanoi



Towers of Hanoi

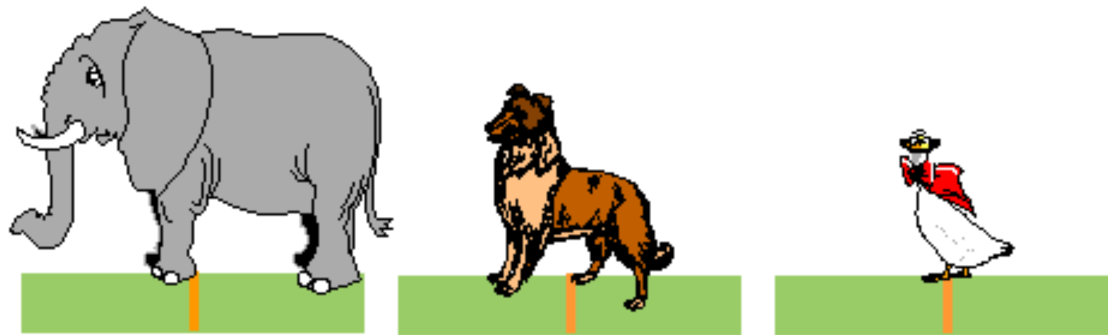


Towers of Hanoi

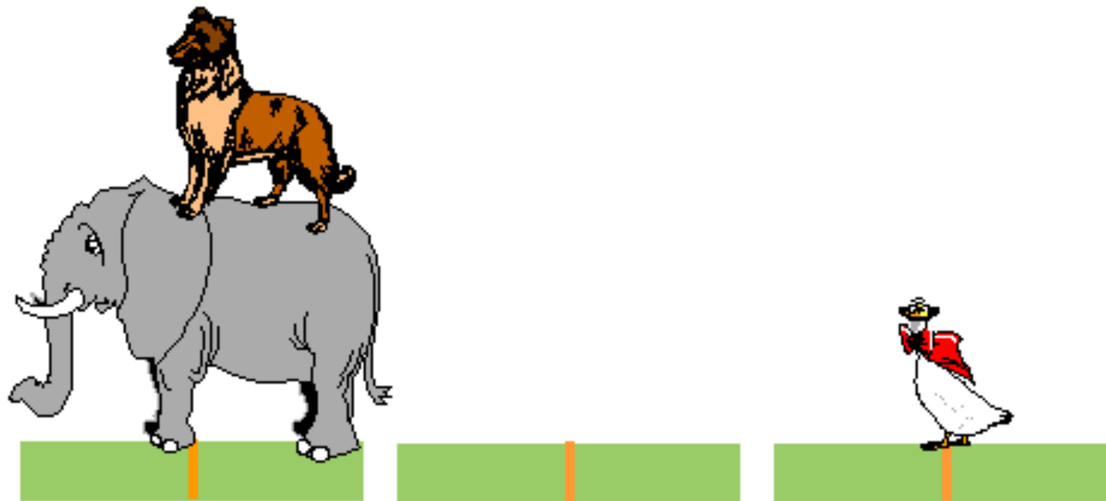
Towers of Hanoi

Towers of Hanoi

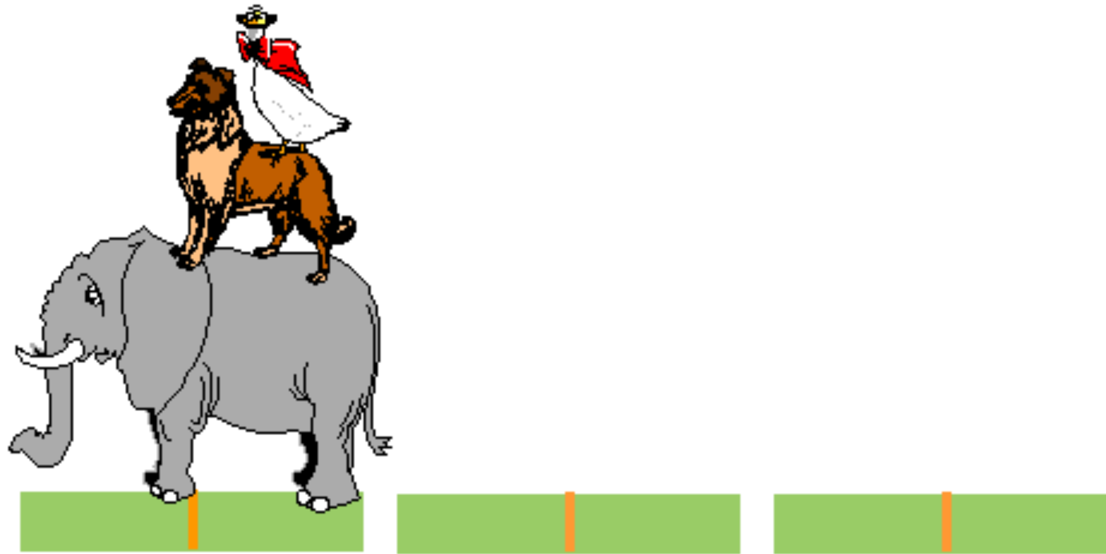
Towers of Hanoi



Towers of Hanoi



Towers of Hanoi



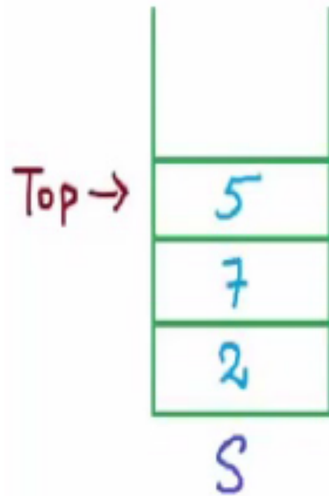
Stack Operations

1. Pop()
2. Push(x)
3. Top()
4. IsEmpty()

- An insertion (of, say x) is called push operation and removing the most recent element from stack is called pop operation.
- Top returns the element at the top of the stack.
- IsEmpty returns true if the stack is empty, otherwise returns false.

All of these take constant time - O(1)

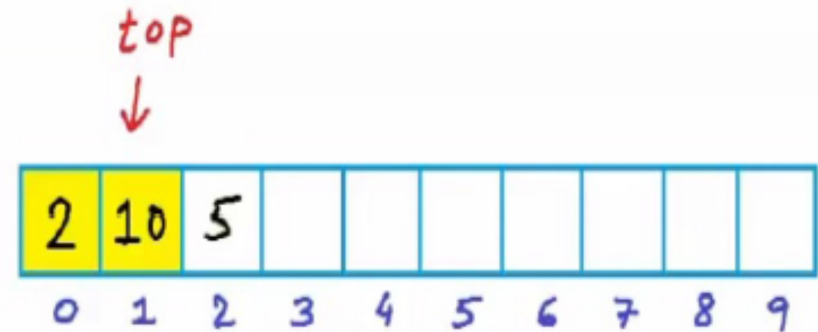
Example



- Push(2)
- Push(10)
- Pop()
- Push(7)
- Push(5)
- Top(): 5
- IsEmpty(): False

Array implementation of stack (pseudocode)

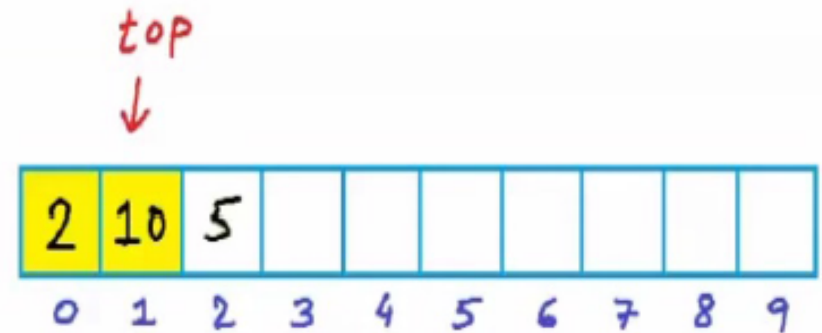
```
int A[10]
top = -1 //empty stack
Push(x)
{
    top = top + 1
    A[top]= x
}
Pop()
{
    top = top – 1
}
```



For an empty stack, top is set to -1.
In push function, we increment top.
In pop, we decrement top by 1.

Array implementation of stack (pseudocode)

```
Top()
{
    return A[top]
}
IsEmpty()
{
    if(top == -1)
        return true
    else
        return false
}
```



Stack
Data Structure

```
#define MAX_STACK_SIZE 100

int stack[MAX_STACK_SIZE];
int top=-1;
```

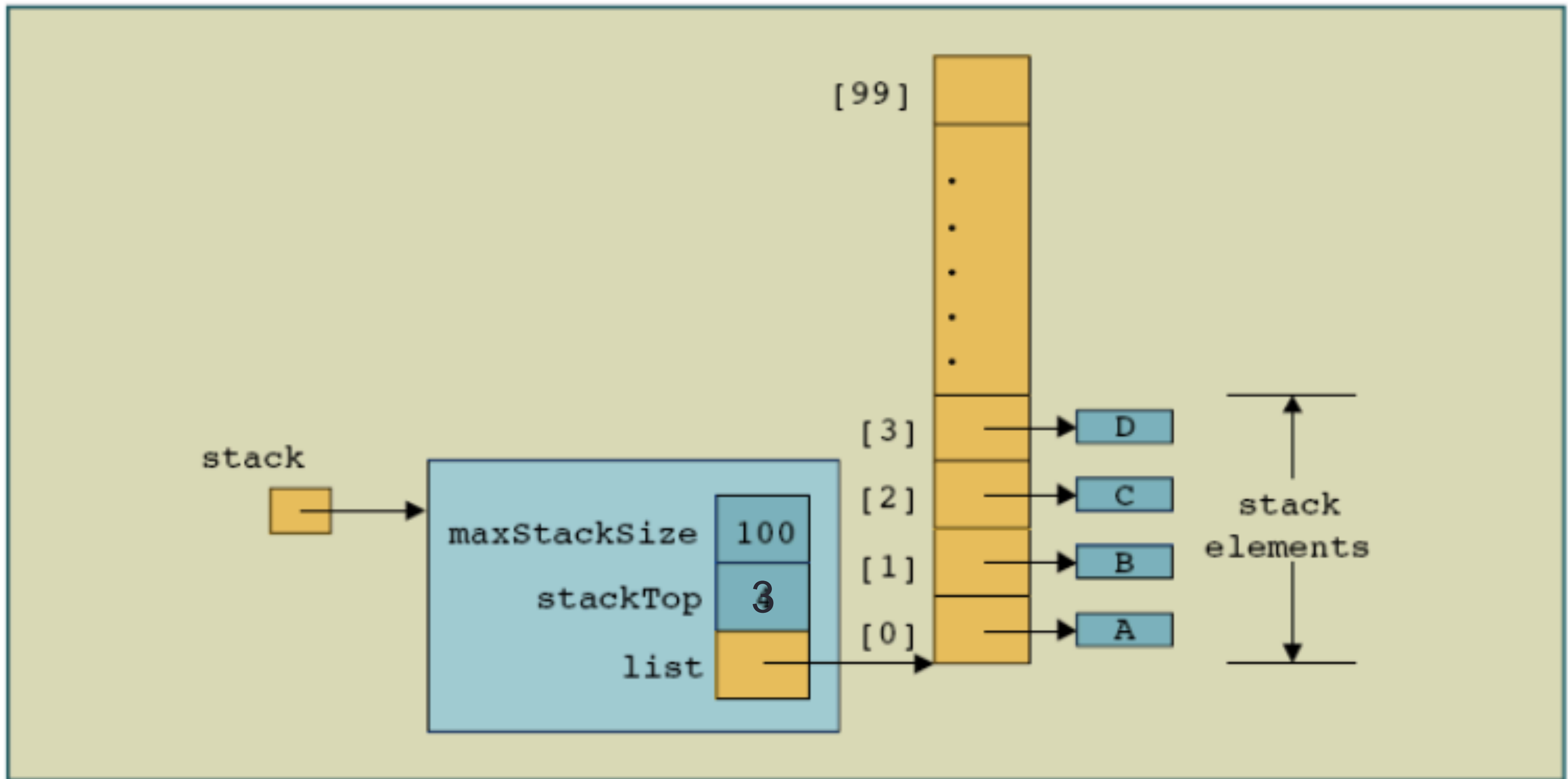
Push Stack

```
// adds the given item to stack if not full
void push (int item)
{
    if(top >= MAX_STACK_SIZE-1){ //isFull
            return;
    }
    stack[++top]=item;
}
```

Pop Stack

```
// removes and returns the item
// some pop implementations only removes
// see further slide
int pop()
{
   if(top==-1)
      return -1; //empty
   return stack[top--];
}
```

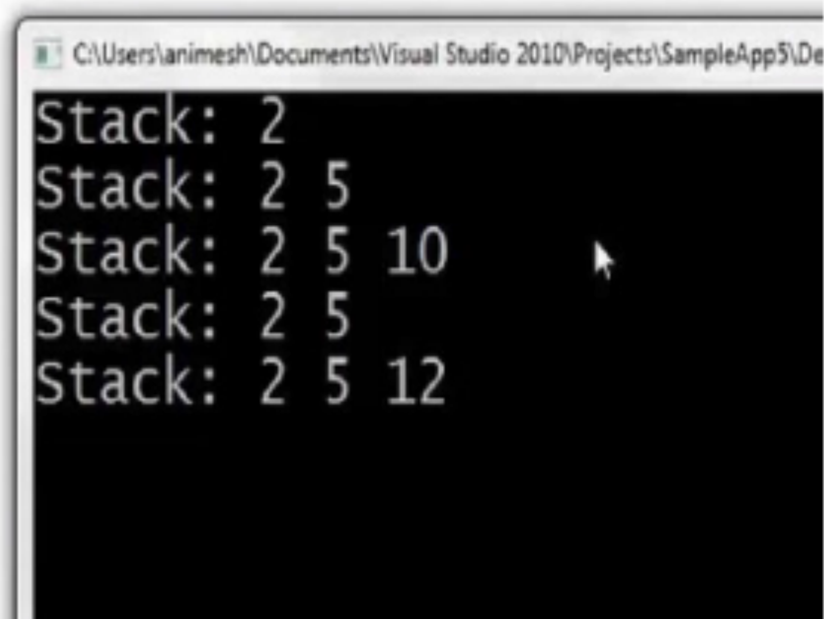
Implementation of Stacks Using Arrays



More array implementation

```c
// Stack - Array based implementation.
#include<stdio.h>
#define MAX_SIZE 101
int A[MAX_SIZE];
int top = -1;
void Push(int x) {
    if(top == MAX_SIZE -1) {
        printf("Error: stack overflow\n");
        return;
    }
    A[++top] = x;
}
void Pop() {
    if(top == -1) {
        printf("Error: No element to pop\n");
        return;
    }
    top--;
}
int Top() {
    return A[top];
}
int main() {
}
```

```c
void Print() {
    int i;
    printf("Stack: ");
    for(i = 0;i<=top;i++)
        printf("%d ",A[i]);
    printf("\n");
}
int main() {
    Push(2);Print();
    Push(5);Print();
    Push(10);Print();
    Pop();Print();
    Push(12);Print();
}
```



Check For Balanced Parentheses using Stack

Expression	Balanced?
(A+B)	
{(A+B)+(C+D)}	
{(x+y)*(z)	
[2*3]+(A)]	
{a+z)	

Check For Balanced Parantheses using Stack

Expression	Balanced?
( )	Yes
{ ( ) ( ) }	Yes
{ ( ) ( )	No
[ ] ( )]	No
{)	No

The count of opening should be equal to the count of closings.
AND
Any parenthesis opened last should be closed first.

Idea: Create an empty list

- Scan from left to right
  - If opening symbol, add it to the list
    - Push it into the stack
  - If closing symbol, remove last symbol using Pop from the stack
    - it should be the same type with opening symbol
  - Should end with an empty list

Check For Balanced Parantheses: Pseudocode

```
CheckBalancedParanthesis(exp){
    n←  length(exp)
    Create a stack: S
    for i←  0 to n-1{
        if exp[i] is '(' or '{' or '['
            Push(exp[i])
        else if exp[i] is ')' or '}' or ']'{
     if (S is empty or
                top does not pair with exp[i])
              return false
            else
              pop()
    }
     }
  return S is empty?
}
```

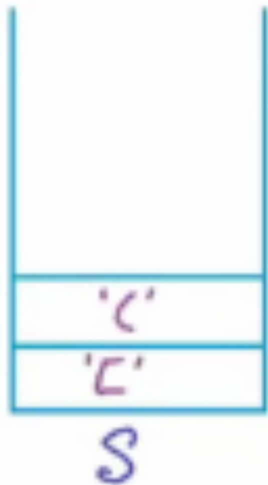
Create a stack of characters and scan this string
by using push if the character is an opening parenthesis and
by using pop if the character is a closing parenthesis. (See next slide)

Examples

exp = [( ])

↑

i = 2

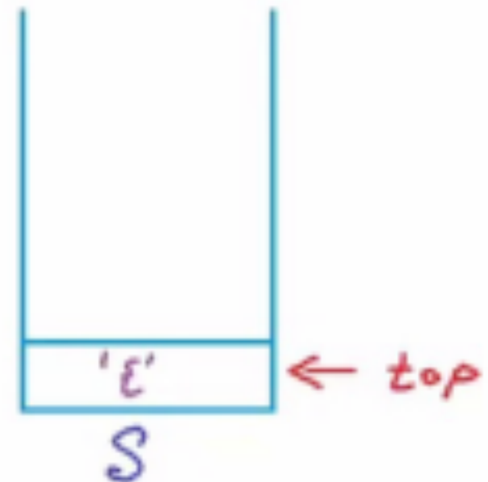


The pseudo code will return false.

exp = { ( ) ( ) }

↑

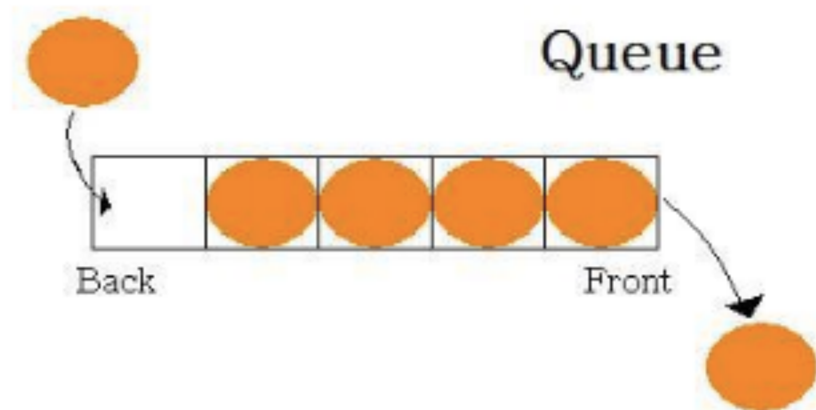
i = 5



The pseudo code will return true.

Queues

- A queue is an ordered list on which
  - all insertions take place at one end called the **rear/back** and
  - all deletions take place at the opposite end called the **front**.
  - Based on **First-in-First-out (FIFO)**



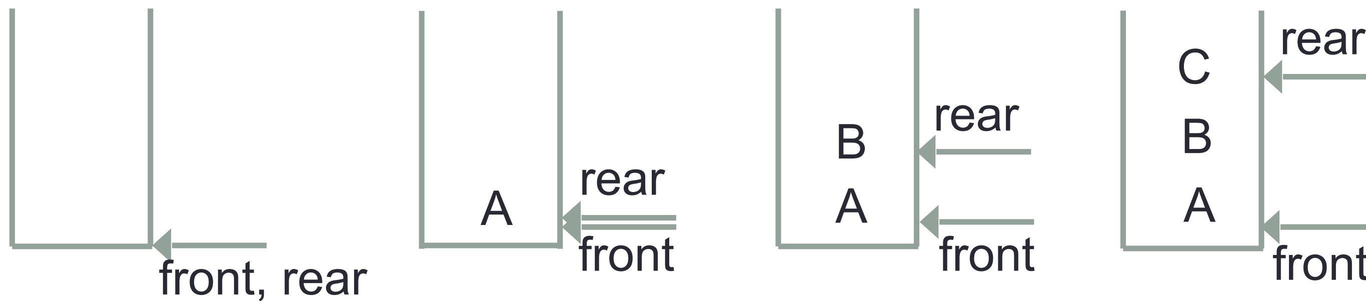
Comparison of Queue and Stack

Queue ADT

Queue - First-In-First-Out (FIFO)

Stack - Last-In-First-Out (LIFO)

Queues



Queue is a list with the restriction that insertion can be made at one end (rear)
And deletion can be made at other end (front).

Built-in Operations for Queue

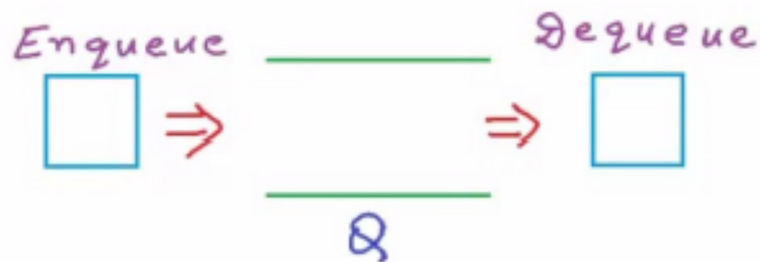
Enqueue(x) or Push(x)

Dequeue() or Pop()

Front(): Returns the element in the front without removing it.

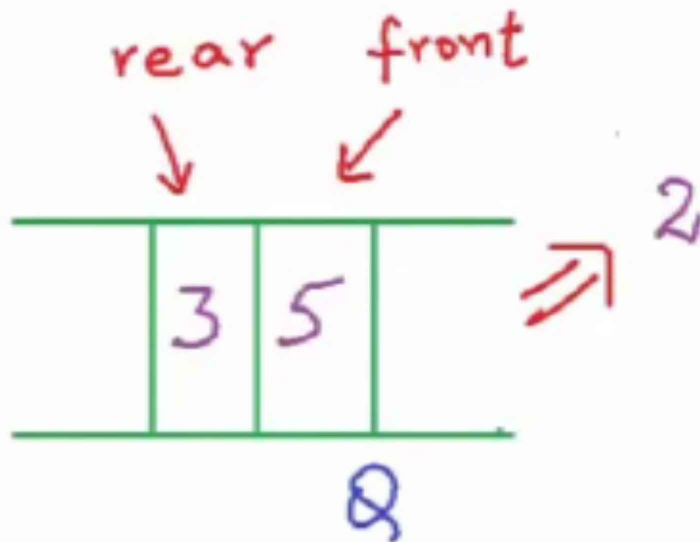
IsEmpty(): Returns true or false as an answer.

IsFull()



Each operation takes constant time, therefore has O(1) time complexity.

Example



Enqueue(2)

Enqueue(5)

Enqueue(3)

Dequeue()→ 2

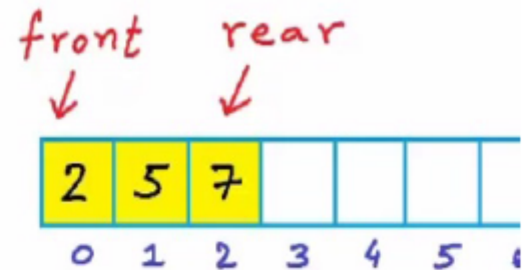
Front()→ 5

IsEmpty()→ False

Applications:
- Printer queue
- Process scheduling

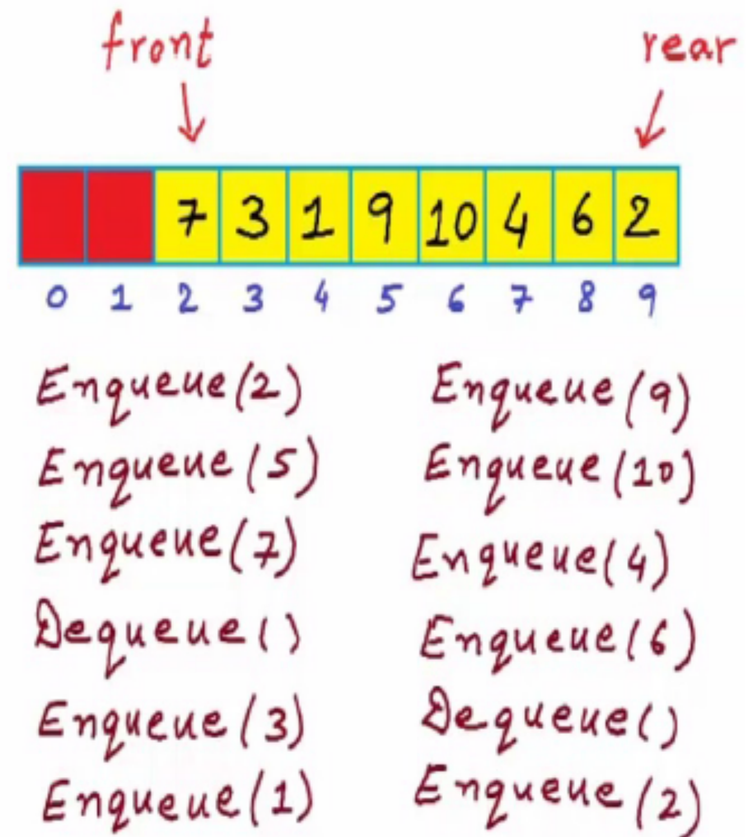
Array implementation of queue (Pseudocode)

```
int A[10]
front ← -1
rear ← -1
IsEmpty(){
    if (front == -1 && rear == -1)
        return true
    else
        return false}
Enqueue(x){
    if IsFull()
        return
    else if IsEmpty()
        front ← rear ← 0
    else
        rear ← rear+1
    A[rear]← x
}
```



Array implementation of queue (Pseudocode)

```
Deque(){
    if IsEmpty(){
        return
    else if (front == rear)
        front ← rear ← -1
    else{
        front ← front+1
}
```



At this stage, we cannot Enqueue an element anymore.

Queue
Data Structure

```
#define MAX_QUEUE_SIZE 100

int queue[MAX_QUEUE_SIZE];
int front=-1;
int rear=-1;
```

Add Queue

```
void enqueue( int item)
{
   if(rear == MAX_QUEUE_SIZE-1){
         return;
   }
   if(front==-1)
     front++;
   queue[++rear]=item;
}
```

Delete Queue

```
int dequeue()
{
   if(front == -1)
      return -1; //empty
   if(front == rear){
      int temp = queue[front];
      front = rear = -1;
      return temp;
   }
   return queue[front++];
}
```

Circular Queue

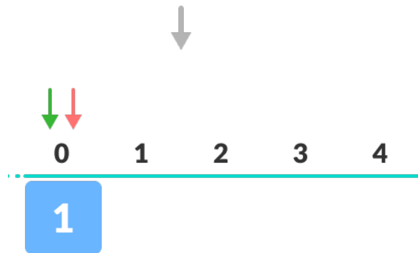
- More efficient queue representation

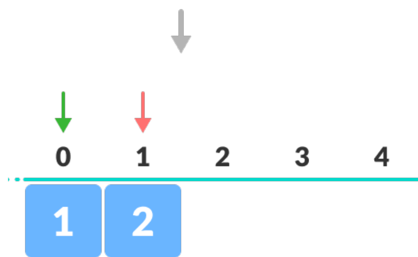
- When the queue is full
  (the rear index equals to MAX_QUEUE_SIZE)
  - We should move the entire queue to the left
  - Recalculate the rear

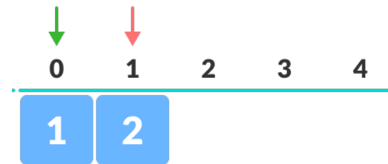
Shifting an array is time-consuming!

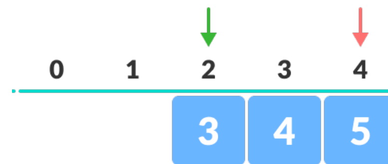
- O(MAX_QUEUE_SIZE)

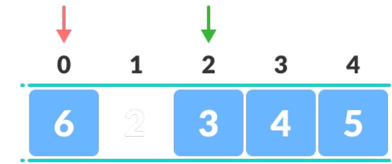
Full Circular Queue

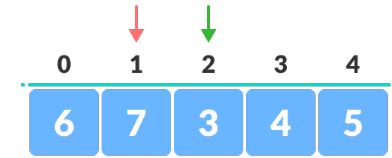










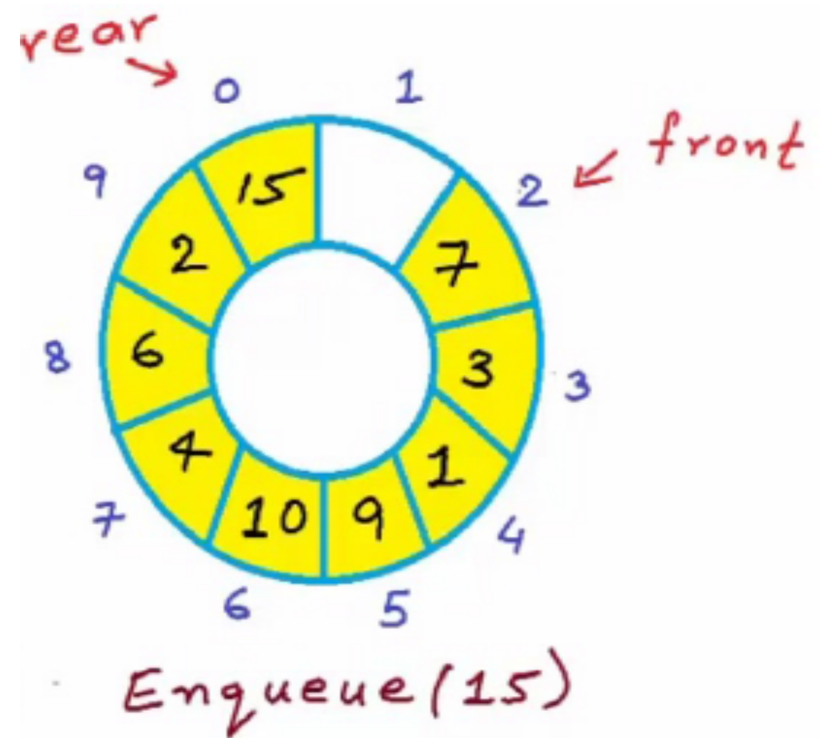


x 2

Enqueue for circular array (Pseudocode)

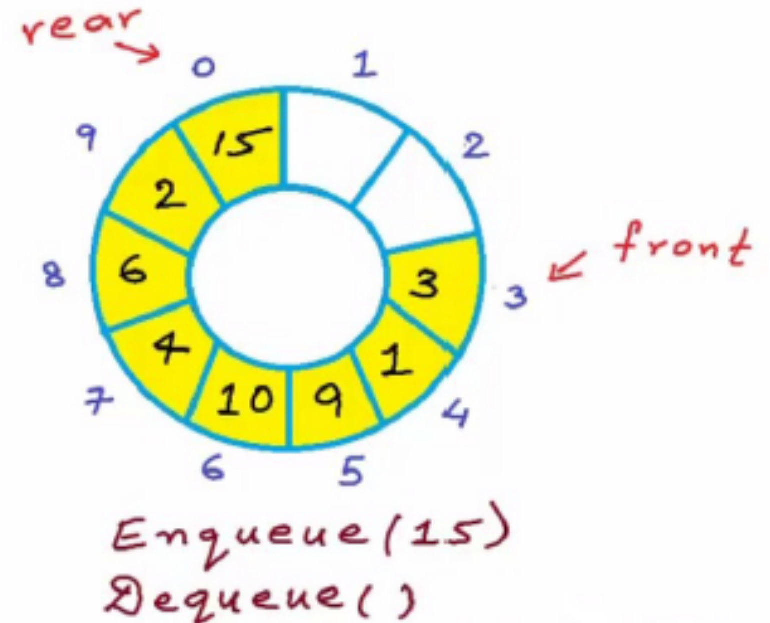
Current position = i
Next position = (i+1)% N
previous position = (i+N-1)%N

```
Enqueue(x){
    if (rear+1)%N == front
        return
    else if IsEmpty()
        front ←  rear ←  0
    else
        rear ←  (rear+1)%N
    A[rear]←  x
}
```



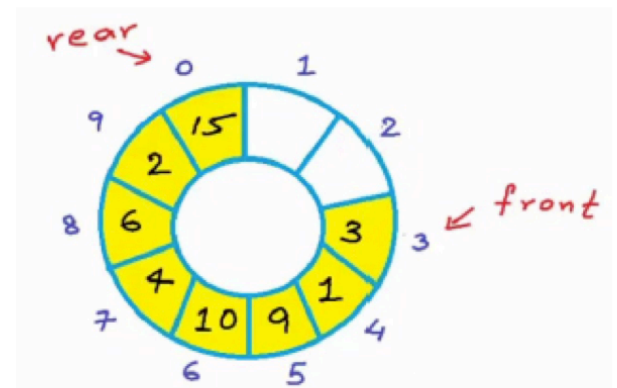
Dequeue for circular array (Pseudocode)

```
Dequeue(){
    if IsEmpty()
        return
    else if(front == rear)
        front ← rear ← -1
    else
        x ← A[front]
        front ← (front+1)%N
  return x
}
```



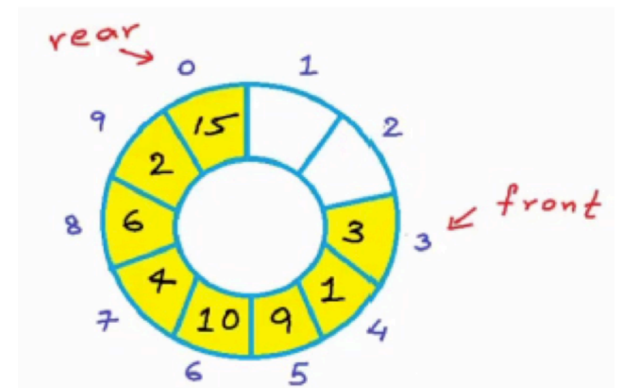
Add Circular Queue

```
void enqueue( int item)
{
    if(front == ((rear+1)% MAX_QUEUE_SIZE){
      return; //queue full
    }
    rear=(rear+1)% MAX_QUEUE_SIZE;
    if(front==-1)
      front++;
    queue[rear]=item;
```



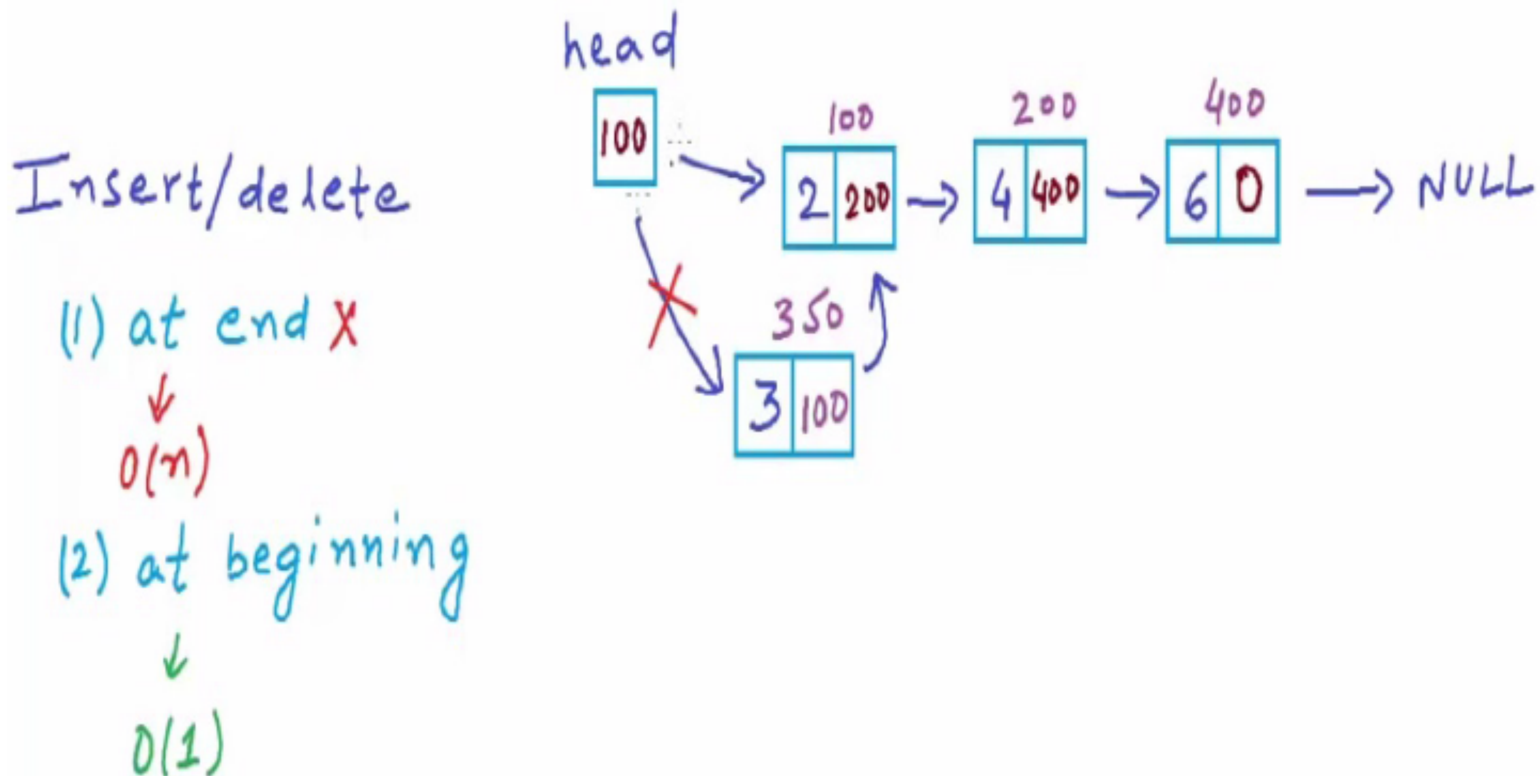
Delete Circular Queue

```
int dequeue()
{
    if(front == -1)
        return -1; //empty
    if(front == rear){
        int temp = queue[front];
        front = rear = -1;
        return temp;
    }
    front=(front+1)% MAX_QUEUE_SIZE;
    return queue[front];
}
```



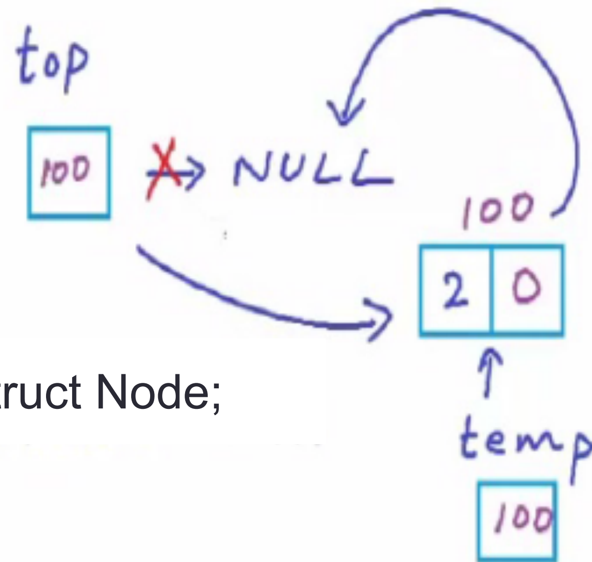
Linked list implementation of stacks

The cost of insert and delete at the beginning of a linked list is constant, that is O(1).
So, in the linked list implementation of stack, we insert and delete a node at the beginning so that time complexity of Push and Pop is O(1).



Instead of head, we use variable name top. When top is NULL, the stack is empty.
With dereferencing temp->data = x and temp->link = top, we fill the fields of the new node, called temp. Finally, with top=temp, top points to the new node.

```
struct Node {
    int data;
    struct Node* link;
};
struct Node* top = NULL;
void Push(int x) {
    struct Node* temp = new struct Node;
    temp->data = x;
    temp->link = top;
    top = temp;
}
```

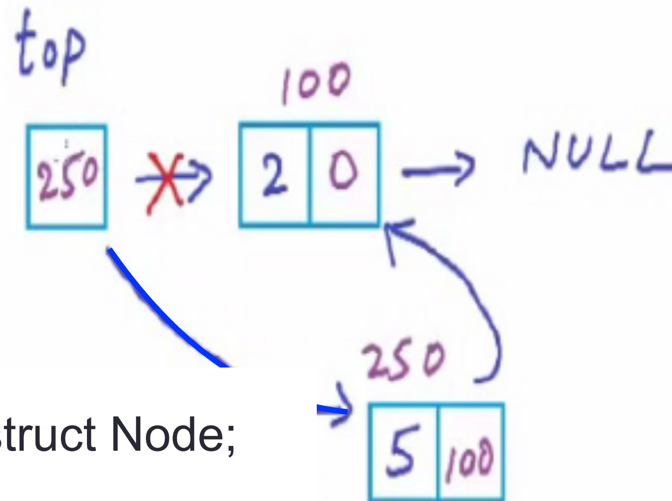


Push(2)

}

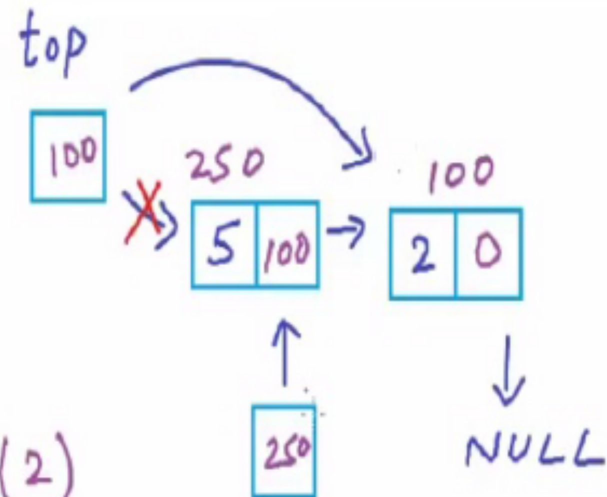
Stack data structure uses the memory efficiently by using push when something is needed and pop when not needed in an array or linked list.

```
struct Node {
    int data;
    struct Node* link;
};
struct Node* top = NULL;
void Push(int x) {
    struct Node* temp = new struct Node;
    temp->data = x;
    temp->link = top;
    top = temp;
}
```



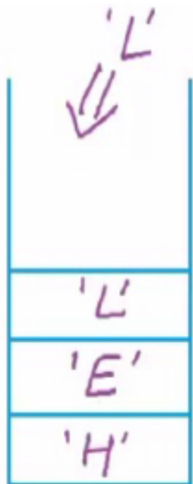
Push(2)
Push(5)

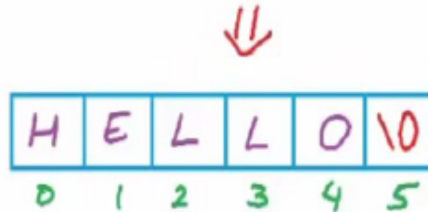
```
struct Node* top = NULL;
void Push(int x) {
    struct Node* temp = new struct Node;
    temp->data = x;
    temp->link = top;
    top = temp;
}
void Pop() {
    struct Node *temp;
    if(top == NULL) return;
    temp = top;
    top = top->link;
    delete temp;
}
```

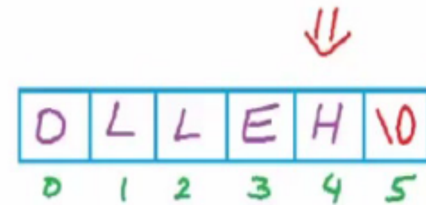


Reverse a string using stack

We can create a stack of characters by traversing the string from left to right and using the push operation.
Then, we can pop the elements from the stack and overwrite all the positions in the string.







We pass the string and the length of the string to the Reverse function.

```
void Reverse(char C[],int n)
{

}
int main() {
    char C[51];
    printf("Enter a string: ");
    gets(C);
    Reverse(C,strlen(C));
    printf("Output = %s",C);
}
```

In C++, we can use these operations available from the standard temporary library called stack.

```
class Stack
{
private:
    char A[101];
    int top;
public:
    void Push(int x);
    void Pop();
    int Top();
    bool IsEmpty();
};
```

As we traverse the string from left to right, we use the push function. In the second loop, we use the top and pop functions for each character in the stack to reverse the string C. See next slide for output.

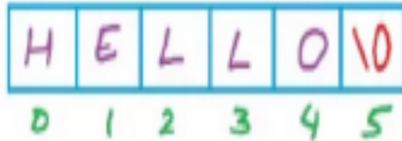
```
#include<iostream>
#include<stack> // stack from standard template library (STL)
using namespace std;
void Reverse(char *C,int n)
{
    stack<char> S;
    //loop for push
    for(int i=0;i<n;i++){
        S.push(C[i]);
    }
    //loop for pop
    for(int i =0;i<n;i++){
        C[i] = S.top(); //overwrite the character at index i.
        S.pop();  // perform pop.
    }
}
```

```
#include<stack> // stack from standard template library (STL)
using namespace std;
void Reverse(char *C,int n)
{
    stack<char> S;
    //loop for push
    for(int i=0;i<n;i++){
        S.push(C[i]);
    }
    //loop for pop
    for(int i =0;i<n;i++){
        C[i] = S.top(); //overwrite the character at index i.
        S.pop();  // perform pop.
    }
}
int main() {
    char C[51];
    printf("Enter a string: ");
    gets(C);
    Reverse(C,strlen(C));
    printf("Output = %s",C);
}
```

C:\Users\animesh\Documents\Visual Studio 2010\Projects\SampleApp2\Debug\S

```
Enter a string: HELLO
Output = OLLEH_
```

Problem: Reverse a string

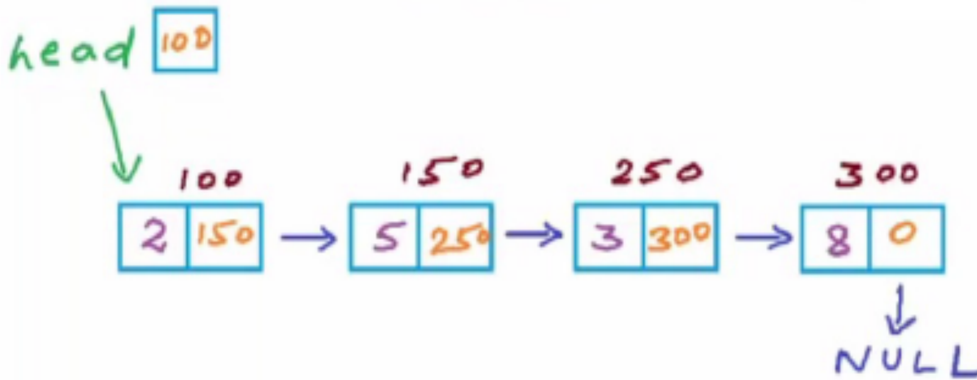



```
void Reverse(char *C,int n)
{
    stack<char> S;
    //loop for push
    for(int i=0;i<n;i++){      // O(n)
        S.push(C[i]);
    }
    //loop for pop
    for(int i =0;i<n;i++){     // O(n)
        C[i] = S.top();
        S.pop();
    }
}
```

- All operations on stack take constant time, O(1).
- Thus, for each loop, time cost is O(n).
- Space complexity is also O(n).

Reverse a linked list using stack

Time complexity of reversing a linked list



Iterative Solution

Time - O(n)

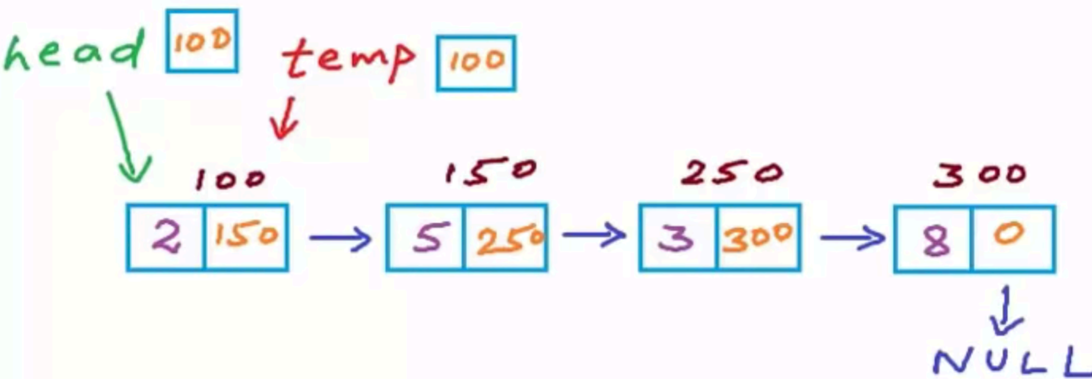
Space - O(1)

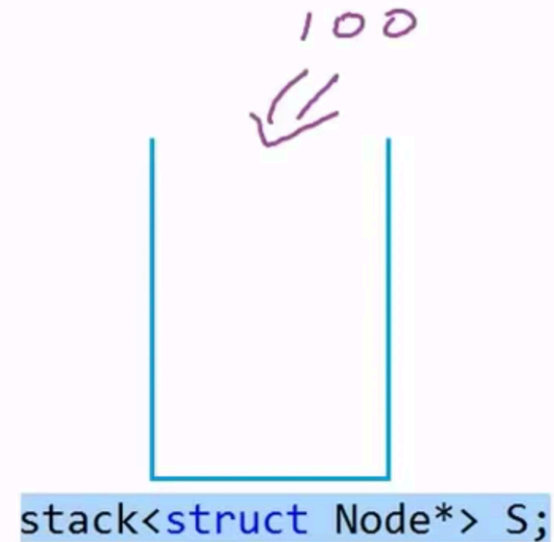
Recursive Solution
(Implicit Stack)

Time - O(n)

Space - O(n)

Time complexity of reversing a linked list



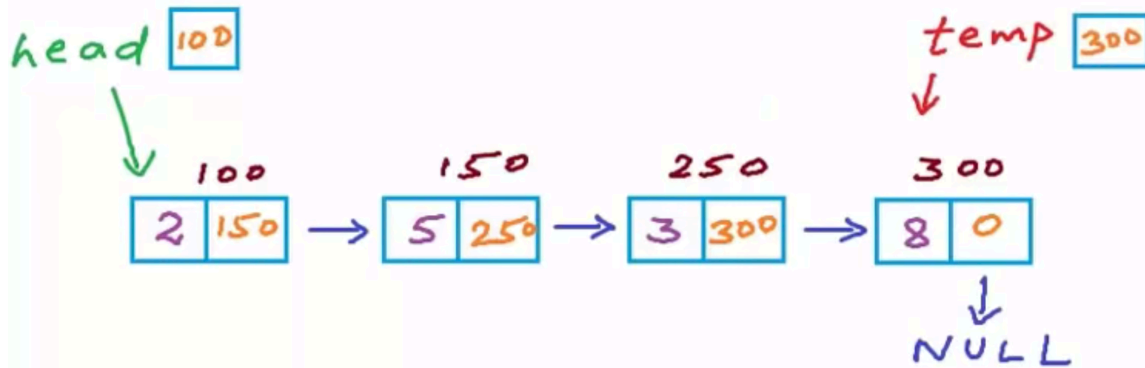



```
struct Node {
    int data;
    Node* next;
};
```

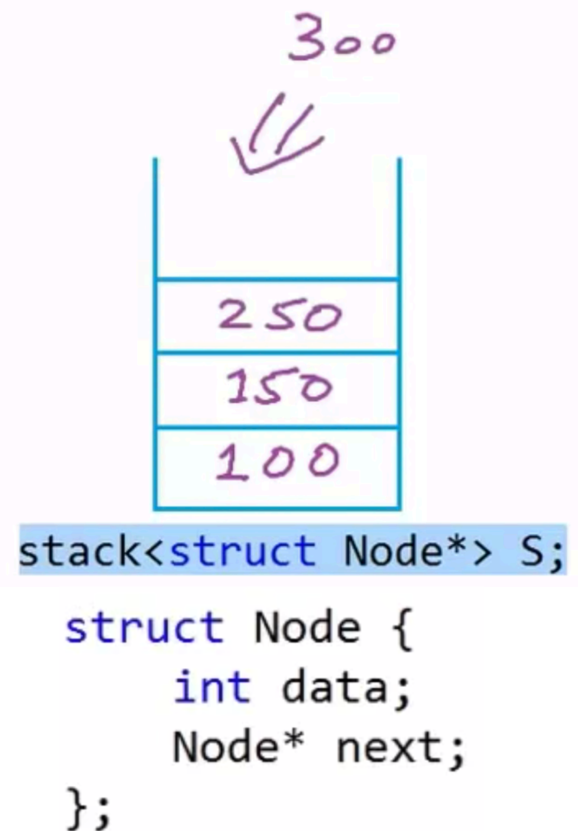
```
Node* temp = head;
while(temp != NULL) {
    S.push(temp);
    temp= temp->next;
}
```

The values we are pushing into the stack are the pointers (references) to the node.

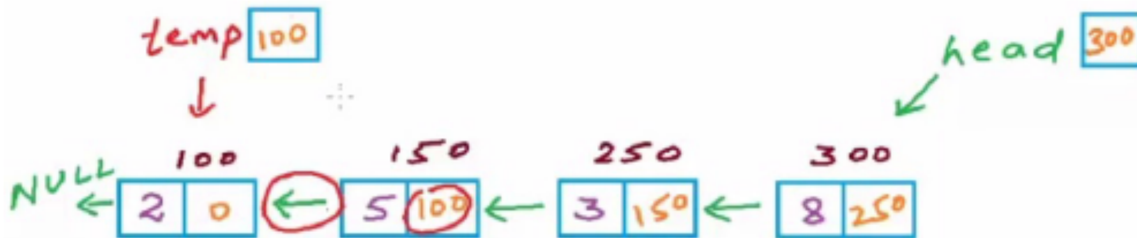
Time complexity of reversing a linked list




```
Node* temp = head;
while(temp != NULL) {
    S.push(temp);
    temp= temp->next;
}
```



Reverse a linked list using stack



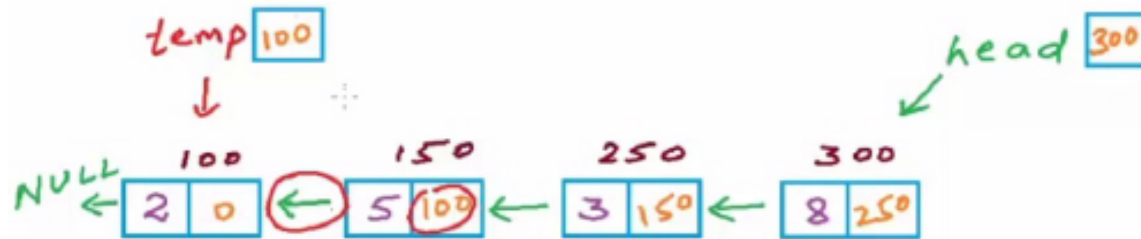

```
Node *temp = S.top();
head = temp;
S.pop();
while(!S.empty()){
    temp->next = S.top();
    S.pop();
    temp = temp->next;
}
temp->next = NULL;
```

300
250
150
100

```
stack<struct Node*> S;

struct Node {
    int data;
    Node* next;
};
```

Once all the references are pushed onto the stack, we can start popping them. As we pop them, we will get references to nodes in reverse order. While traversing the list in reverse order, we can build reversed links.

Reverse a linked list using stack

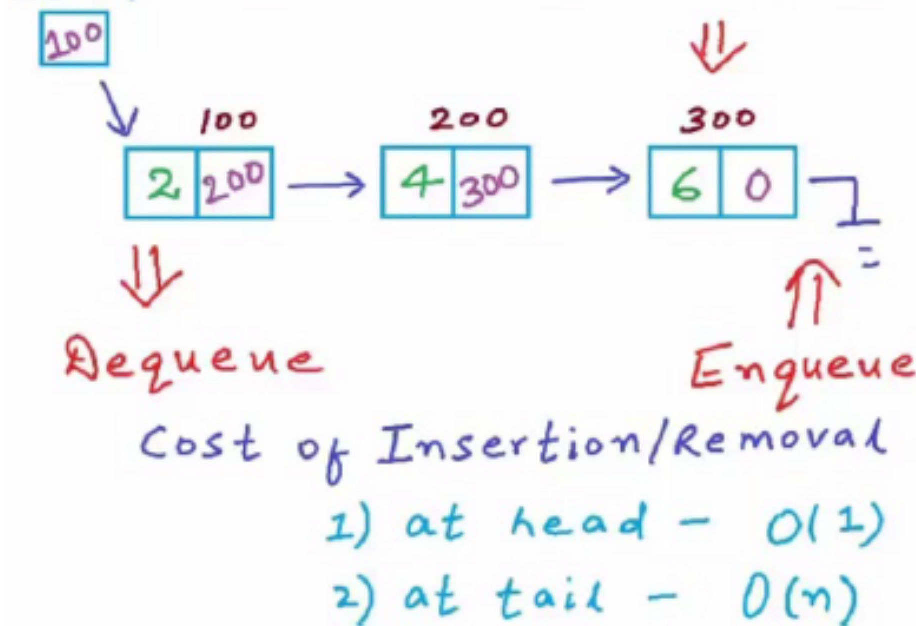



```
Node *temp = S.top();
head = temp;
S.pop();
while(!S.empty()){
    temp->next = S.top();
    S.pop();
    temp = temp->next;
}
temp->next = NULL;
```

```
stack<struct Node*> S;

struct Node {
    int data;
    Node* next;
};
```

Initially, S.top() returns 300. We store head as this address, so head stores 300.
In the loop, while stack is not empty, set temp->next to the address at the top of the stack, which is 250 at first.
Finally, temp points to the node at address 100.
The last line is needed so that the end of the reversed linked list points to NULL and not to 150.

Linked list implementation of queue

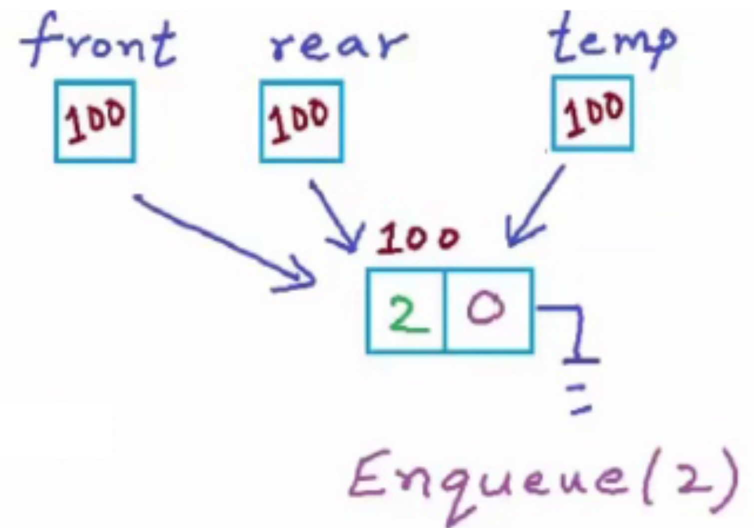


In this case above, Dequeue will take constant time (O(1)), but Enqueue will take O(n) time.

As before, the traversal of the linked list to learn the address of the last node takes O(n) time.

Solution: Keep a pointer called rear that stores the address of the node at the rear position.

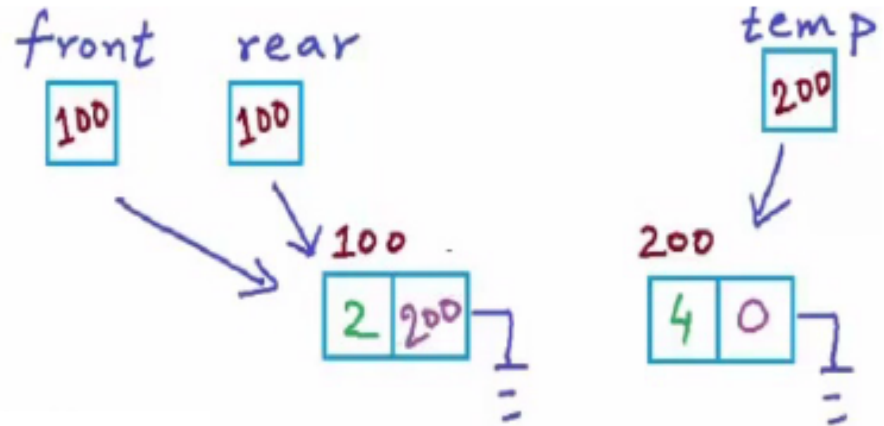
```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp = new struct Node;

    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



Instead of declaring a pointer variable pointing to head, we declare two pointer variables one pointing to the front node and another pointing to the rear node.

For the Enqueue, we again use a pointer called temp to point to the new node created dynamically.
If the list is empty, both front and rear point to the new node.

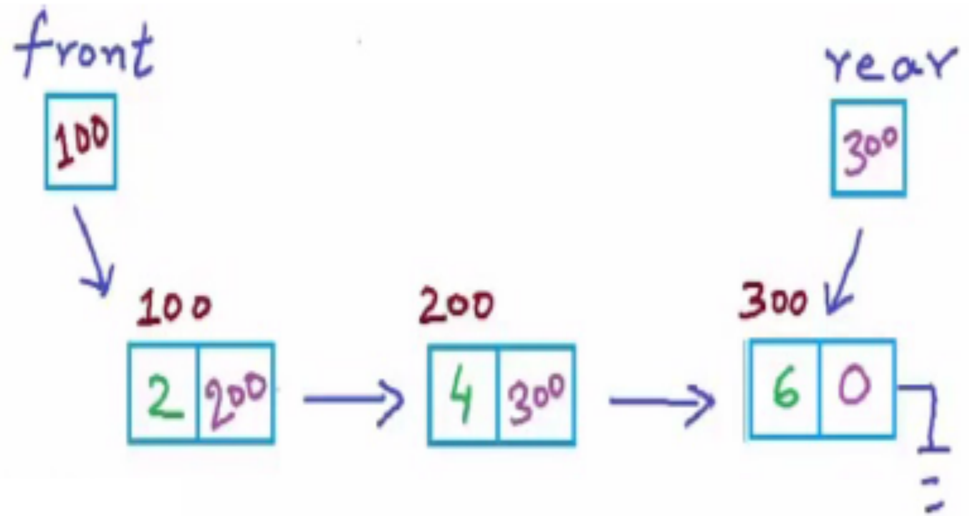
```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =    new struct Node;

    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



If the list is not empty, first, set the address part of the current rear node to the address of the new node.

Then let the rear pointer point to the new node.

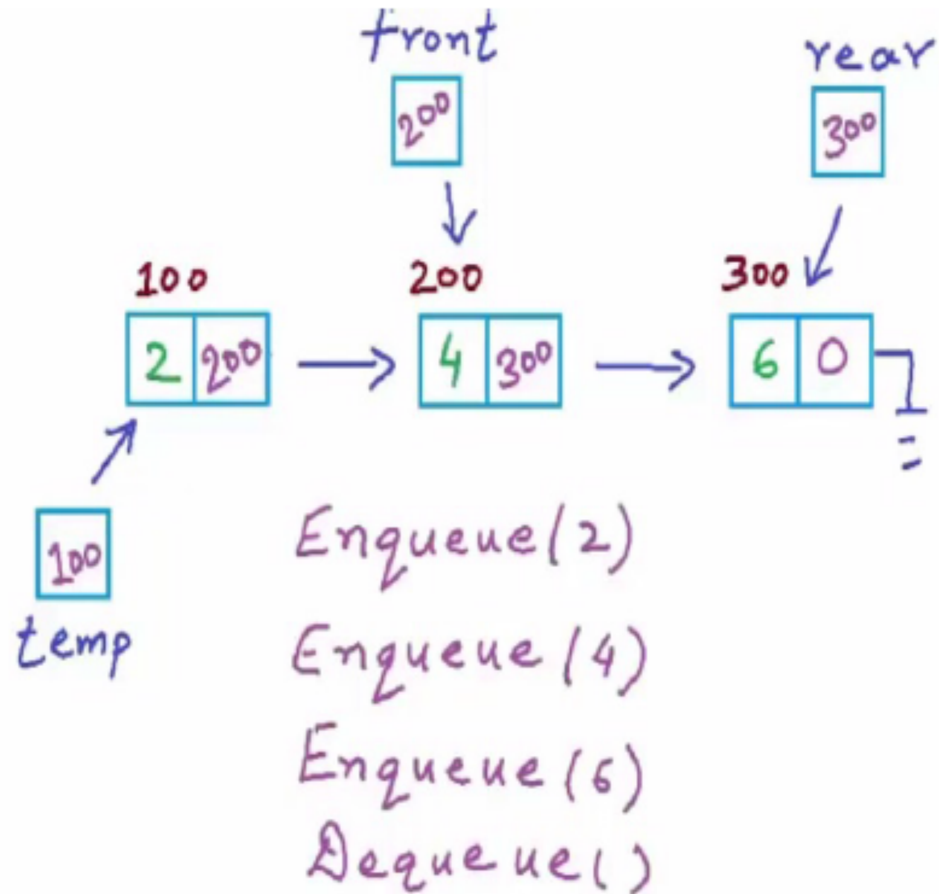
```c
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =   new struct Node;

    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



Enqueue(2)
Enqueue(4)
Enqueue(6)

Implementation of Dequeue

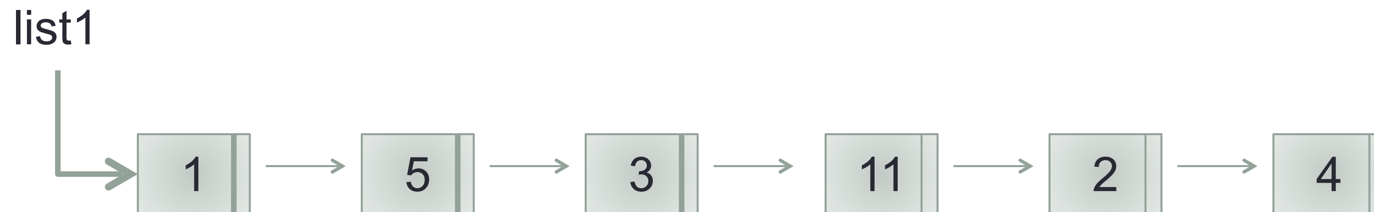
```
void Dequeue() {
    struct Node* temp = front;
    if(front == NULL) return;
    if(front == rear) {
        front = rear = NULL;
    }
    else {
        front = front->next;
    }
⇒ delete temp;
}
```



Multiple Stack Implementation Using Linked List

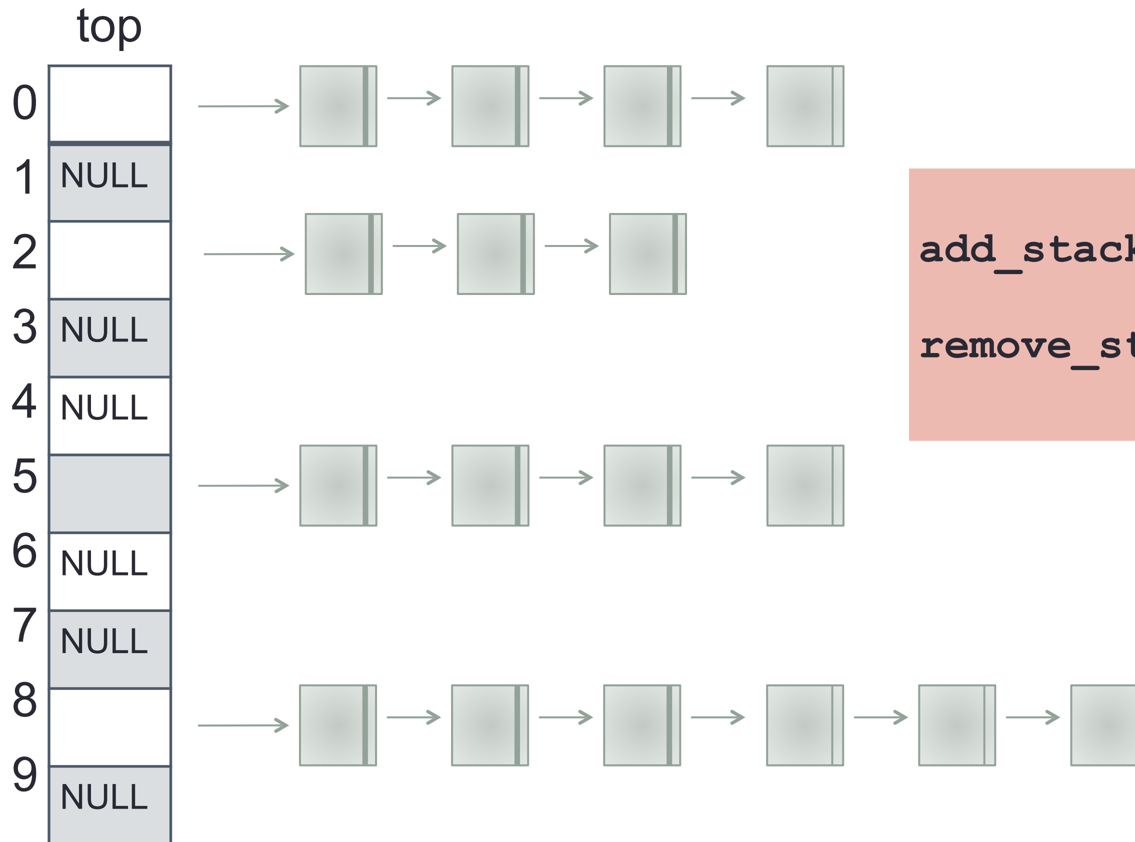
Multiple Stack Implementation Using Linked List

A single stack:



Multiple Stack Implementation Using Linked List

Multiple stacks:




```
add_stack(&top[stack_i], x);

remove_stack(&top[stack_i]);
```

Linked List

```
#include <iostream>
#define MAX_STACK 10 // number of max stacks
using namespace std;

typedef struct{
        int  value;
        //other fields …
}item;

struct stack{
        item x;
        stack * next;
};

typedef struct stack *STACKPTR;

STACKPTR top[MAX_STACK];
```

Linked List

```
#define ISFULL(p) (!p)
#define ISEMPTY(p) (!p)

int main()
{
    for(int i=0;i<MAX_STACK;i++)
        top[i] = NULL;

    return 0;
}
```

Linked List

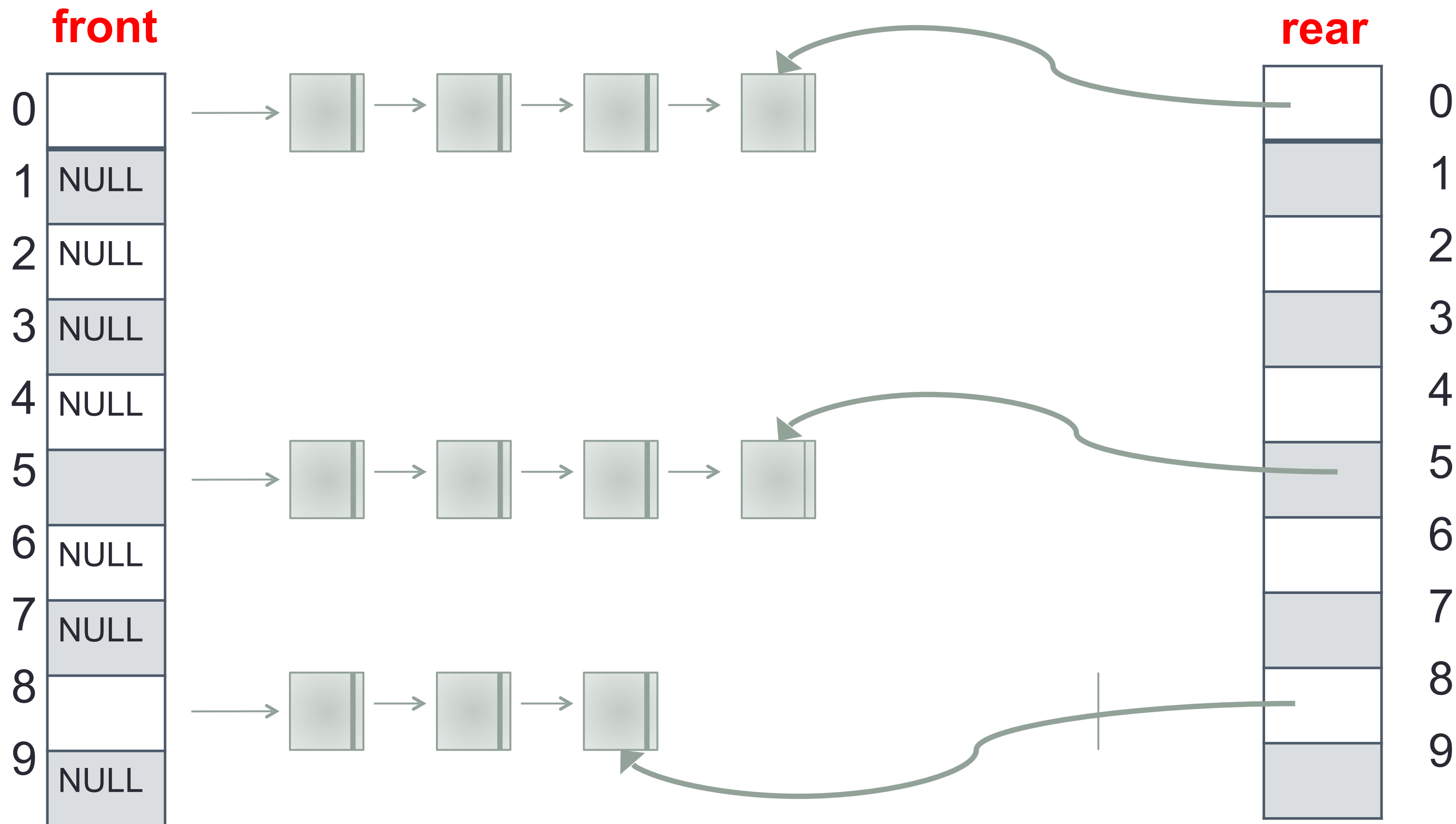
```
void add_stack(STACKPTR *top, item x)
{
        STACKPTR p = new stack;
        if(!p)
        {
              cout<<"Memory allocation failed<<endl;
              exit(1); // throw an exception here..
        }
        p->x = x;
        p->next = *top;
        *top = p;
}
```

Linked List

```
item remove_stack(STACKPTR *top)
{
        STACKPTR p = *top;
        item x;

        if(!p)
        {
                cout<<"Stack is empty!"<<endl;
                exit(1);
        }
        x = p->x;
        *top = p->next;
        delete p;

        return(x);
}
```

Multiple Queue Implementation Using Linked List

Multiple Queue Implementation Using Linked List



Linked List

```
#define MAX_QUEUE 10 // number of max queues
#define ISFULL(p) (!p)
#define ISEMPTY(p) (!p)

typedef struct{
       int  value;
       //other fields
}item;

struct queue{
       item  x;
       queue * link;
};

typedef struct queue *QUEUEPTR;

QUEUEPTR front[MAX_QUEUE],rear[MAX_QUEUE];

for(int i=0;i<MAX_QUEUE;i++)
      front[i]=NULL;
```

Linked List

```
void add_queue(QUEUEPTR *front, QUEUEPTR* rear, item x)
{
     QUEUEPTR p = new queue;
     if(ISFULL(p)){
         cout<<"Memory allocation failed!!!"<<endl;
         exit(1); // throw an exception here..
     }
     p->x = x;
     p->link = NULL;
     if(*front)
        (*rear)->link = p;
     else
         *front = p;
     *rear = p;
}
```

Linked List

```
item remove_queue(queue_pointer *front)
{
     QUEUEPTR p = *front;
     item x;
     if(ISEMPTY(*front)){
          cout<<“Queue is empty!!!”<<endl;
          exit(1);
     }

     x = p->x;
     *front = p->link;
     delete p;
     return(x);
}
```

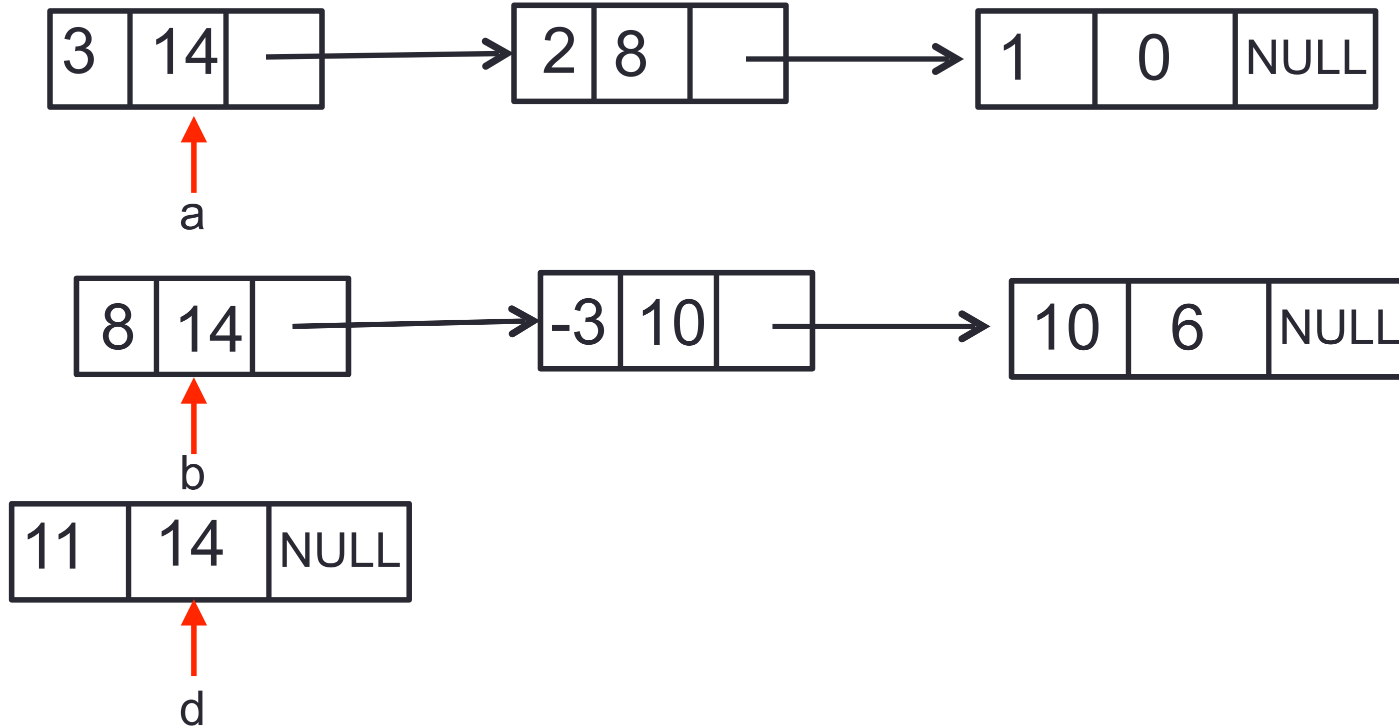
Addition of polynomials by using singly linked lists

Adding Polynomials using linked list

$$a(x) = 3x^{14} + 2x^{8} + 1$$
$$b(x) = 8x^{14} - 3x^{10} + 10x^{6}$$

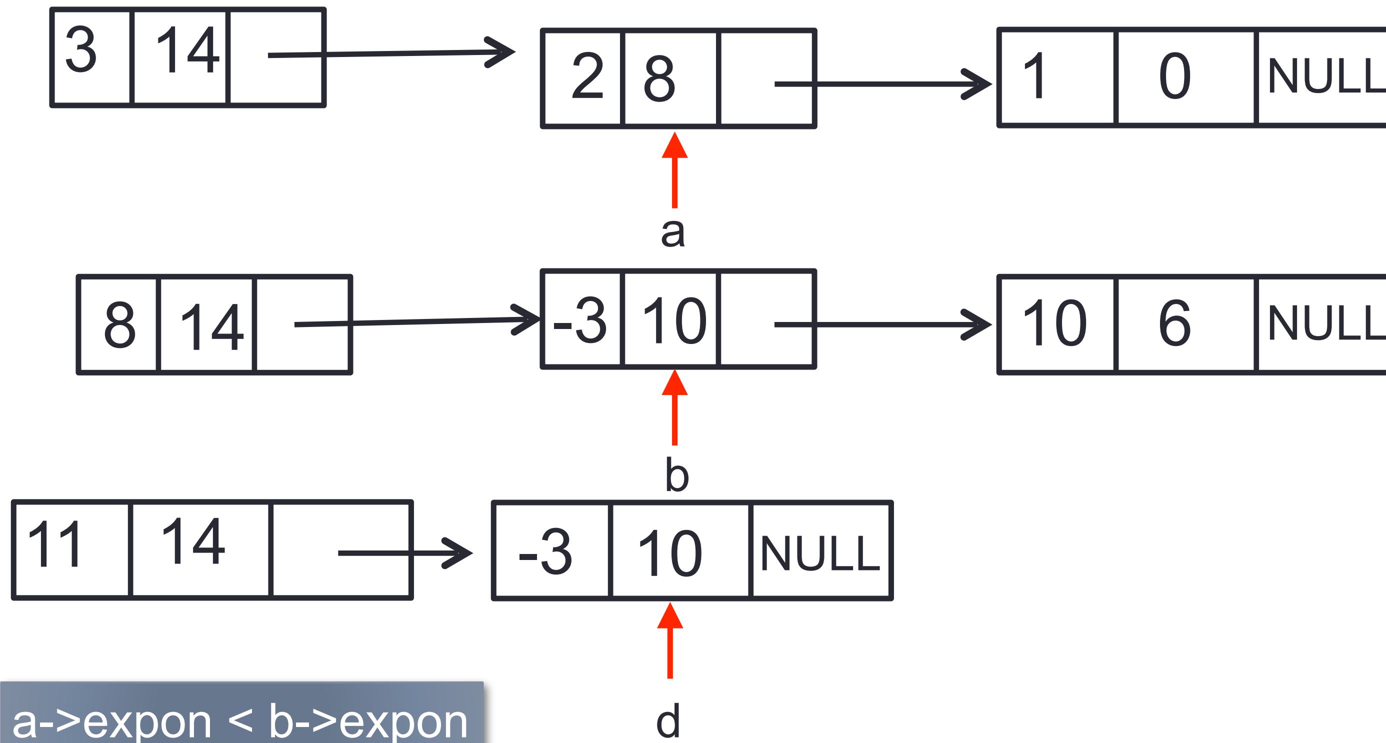
```
typedef struct poly_node* POLYPTR;
struct poly_node{
    int coef; // coefficient
    int expon; // exponent
    POLYPTR link;
};
POLYPTR a,b,d;
```

If the exponents of two terms are equal, we add the two coefficients and create a new term, also move the pointers to the next nodes in a and b.



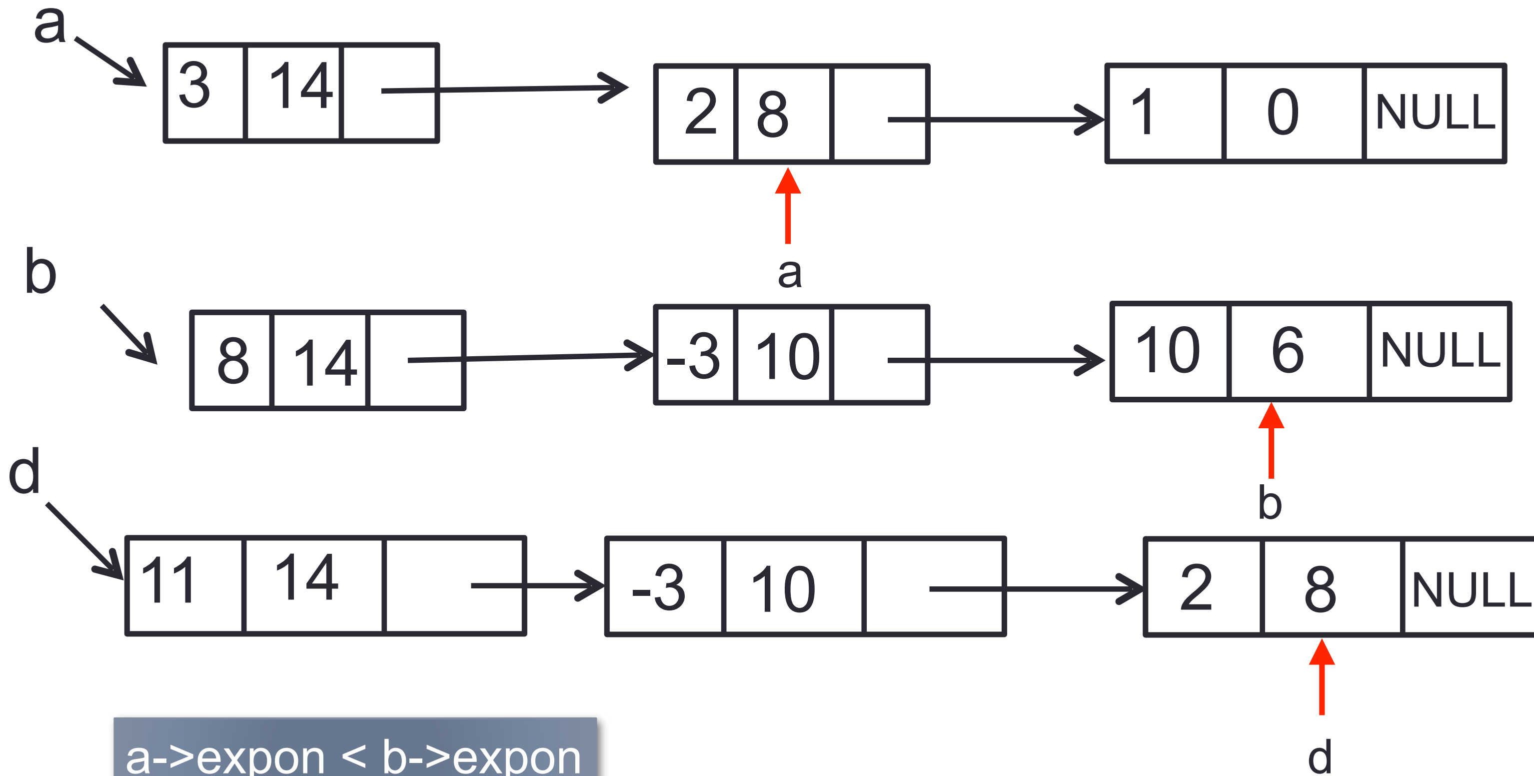
a->expon == b->expon

If the exponent of the current term in a is less than the exponent of the current term in b, then we create a duplicate term of b and attach this term to the result, called d. Then move the pointer to the next term in b. (Similarly, if the exponent of the current term in a is larger than the exponent of the current term in b.)



To avoid having to search for the last node in d, each time we add a new node, we keep a pointer, “rear”, which points to the current last node in d.

Complexity of p.add: O(m+n).,
m and n, the number of terms in the polynomials a and b. Why?



```
void attach(float coefficient,int exponent,POLYPTR *rear){
/* create a new node with coef = coefficient and expon =
exponent, attach it to the node pointed to by rear, current
last node in d. rear is updated to point to this new node */

    POLYPTR temp;
    temp = new poly_node;
    if (IS_FULL(temp)){
      cerr<<"The memory is full\n"<<endl;
      exit(1);
    }
    temp->coef = coefficient;
    temp->expon = exponent;
    (*rear)->link = temp;
    *rear = temp;
}
```

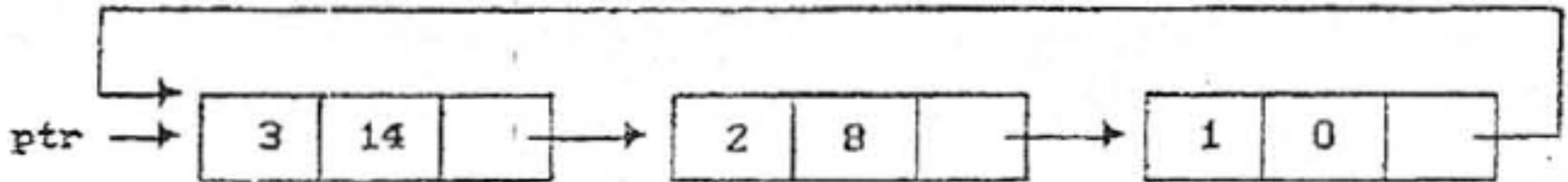
```
poly_pointer padd(poly_pointer a, POLYPTR b){
/* return a polynomial which is the sum of a and b */
    POLYPTR front, rear, temp;
    int sum;
    rear = new poly_node;
    if (IS_FULL(rear)){
        cerr<<"The memory is full\n"<<endl;
        exit(1);
    }
    front = rear;
    while (a && b)
      switch (COMPARE(a->expon,b->expon)){
        case -1: /* a->expon < b->expon */
            attach(b->coef,b->expon, &rear);
            b = b->link; break;
        case 0: /* a->expon == b->expon */
            sum = a->coef + b->coef;
            if(sum)
                attach(sum,a->expon,&rear);
            a = a->link; b = b->link; break;
        case 1: /* a->expon > b->expon */
            attach(a->coef,a->expon,&rear);
            a = a->link; break;
      }
```

```
    /* copy rest of list a and then list b */
    for (; a; a=a->link)
         attach(a->coef,a->expon,&rear);
    for (; b; b=b->link)
         attach(b->coef,b->expon,&rear);
    rear->link = NULL;

    /* delete extra initial node */
    temp = front;
    front = front->link;
    delete temp;

    return front;
}
```

Singly-linked circular lists

Representing polynomials as circularly linked List

We can free the nodes of a polynomial more efficiently if we have a circular list:
In a circular list, the last node is linked to the first node.



$ptr(x) = 3x^{14} + 2x^{8} + 1$

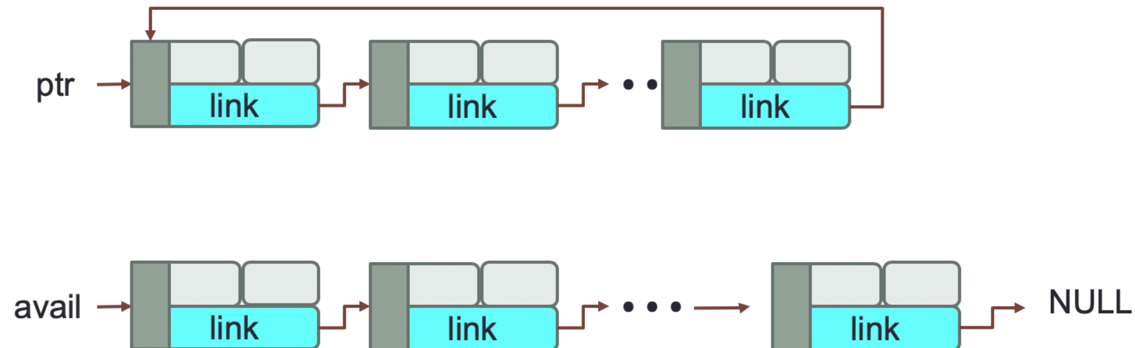
Methodology:

We free nodes that are no longer in use so that we may reuse these nodes later as follows:

- Obtain an efficient erase algorithm for circular lists
- Maintain our own list of nodes that have been "freed" as a chain. Let's call this list of free nodes as the available list.
- When we need a new node, examine the available list and use one of the free nodes if the available list is not empty
- Use "new" only if the available list is empty

- Let avail be a variable of type POLYPTR that points to the first node in the available list, our list of free nodes.
- Instead of using "new" and "delete", we use get_node and ret_node.

(See next slide)



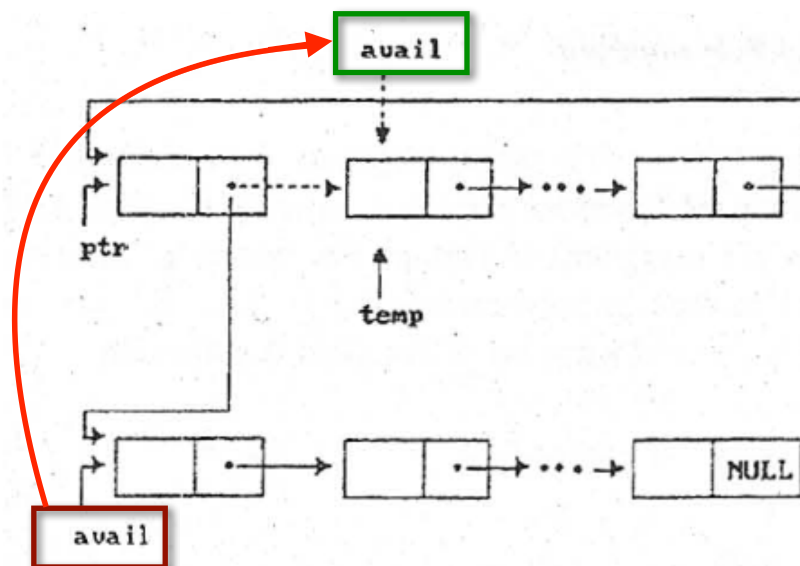
```
POLYPTR get_node(){
/*provide a node for use*/
    POLYPTR node; /*free node available, use that*/
    if (avail){
        node = avail;
        avail = avail->link;
    }
    else{/*there is no free node available, create one*/
        node = new poly_node;
        if(!node){
            cerr<<"No sufficient memory!"<<endl;
            exit(1);
        }
    }
    return node;
}
```

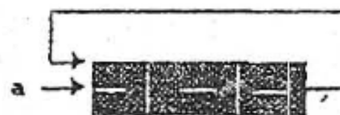
```
void ret_node(POLYPTR ptr){
/*return a node to the available list*/
    ptr->link = avail;
    avail = ptr;
}
```

Erasing in a circular list:

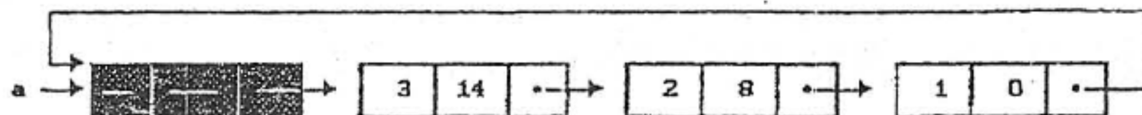
- We may erase a circular list in a fixed amount of time independent of the number of nodes in the list using *cerase.*
- To handle the special case of zero polynomial, a head node is introduced for each polynomial. Therefore, also the zero polynomial contains a node.

```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```





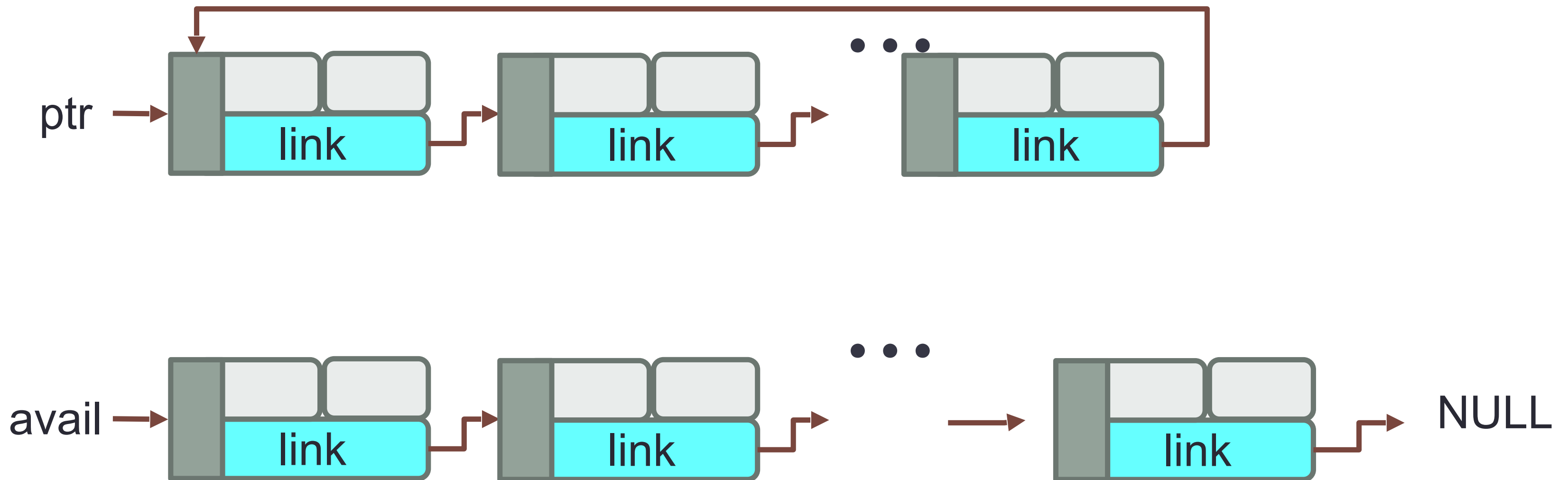
(a) Zero polynomial



(b) $3x^{14}+2x^8+1$

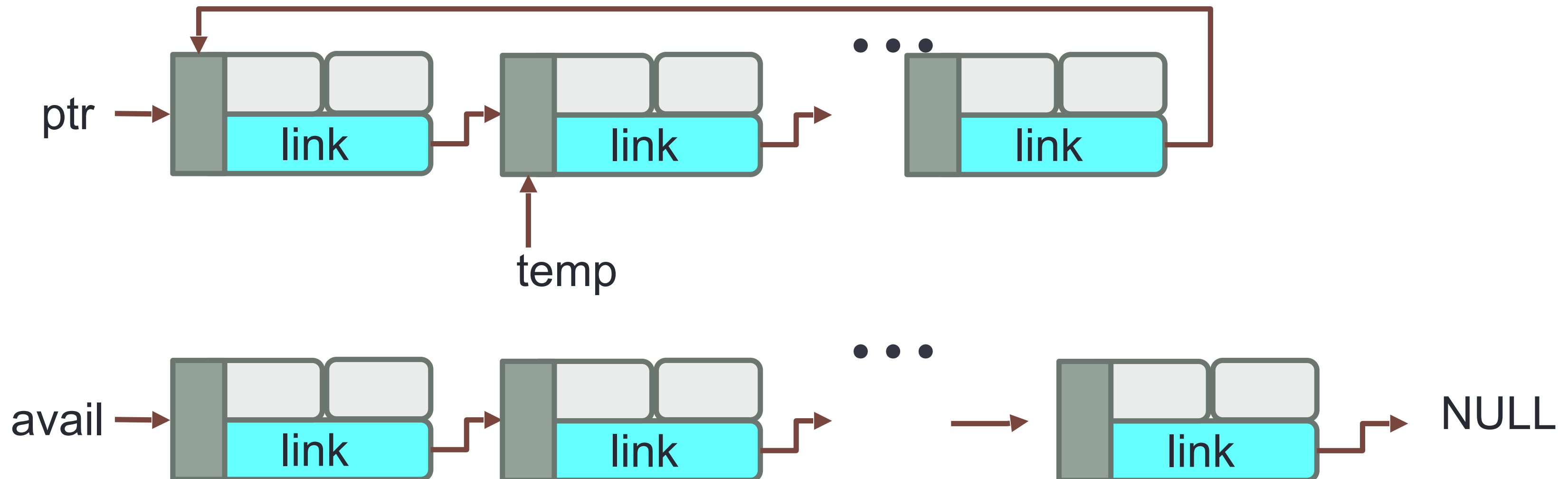
Erasing in a circular list:

```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



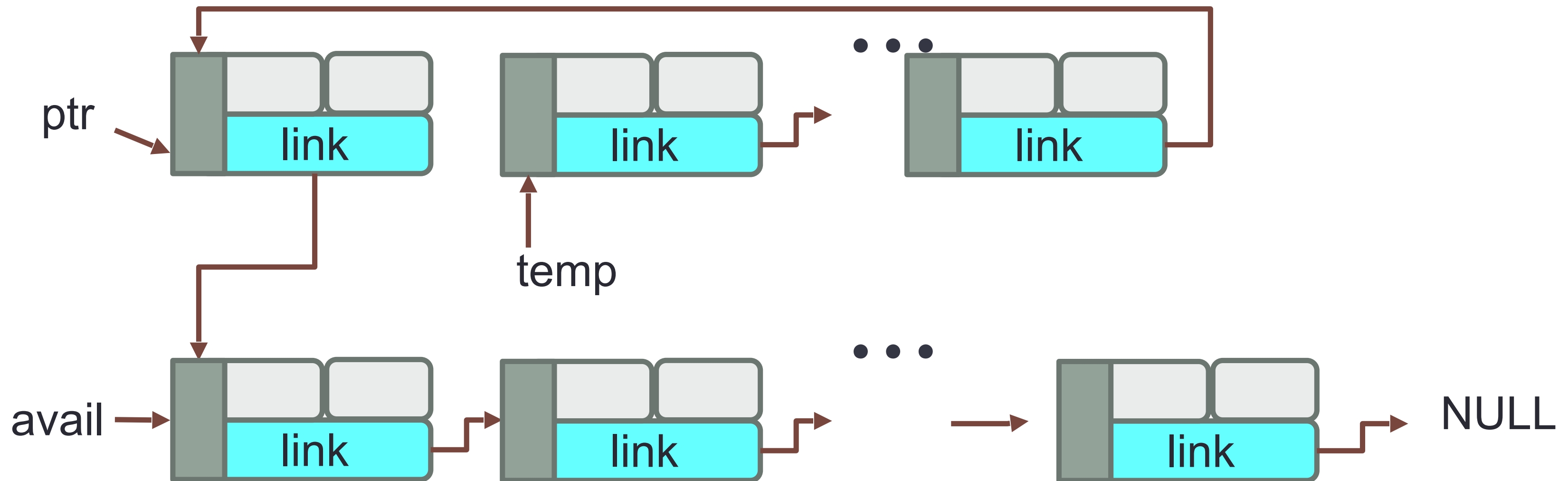
Erasing in a circular list:

```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



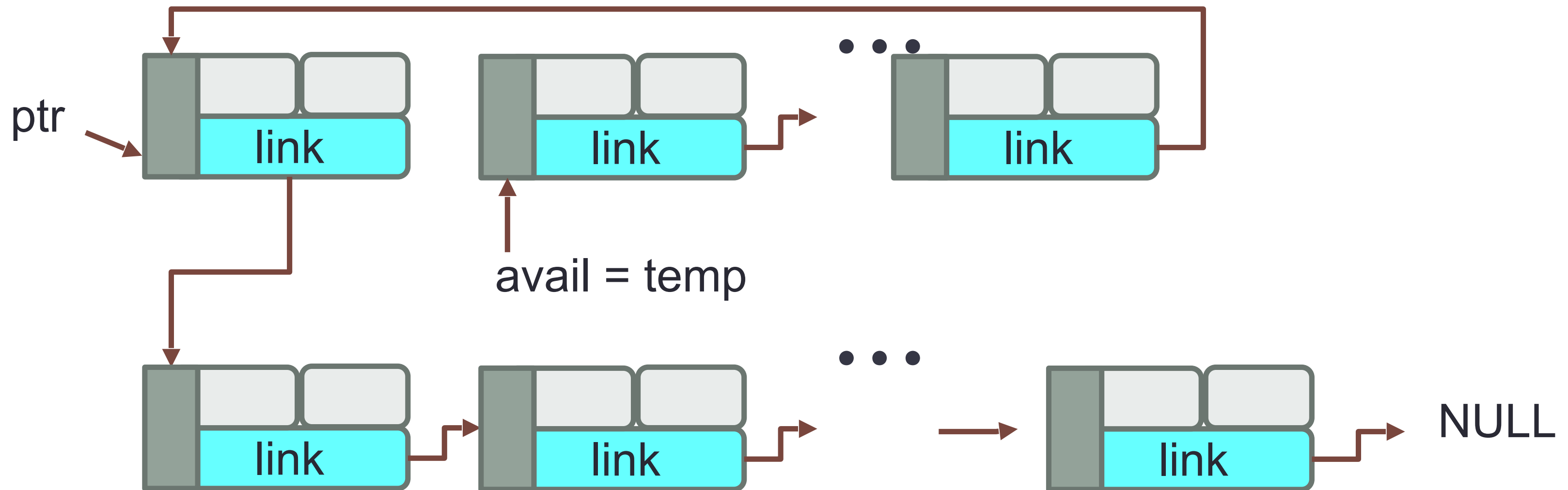
Erasing in a circular list:

```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



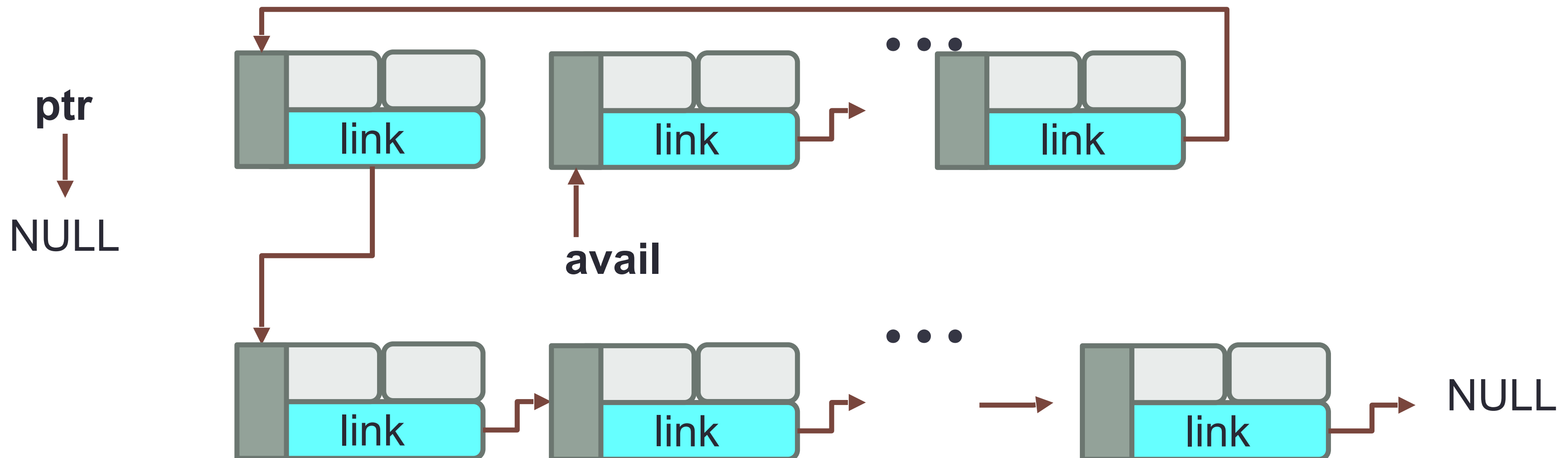
Erasing in a circular list:

```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



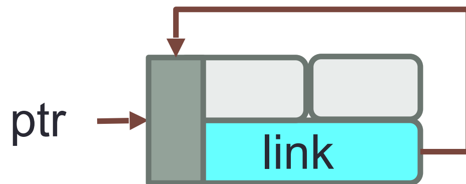
Erasing in a circular list:

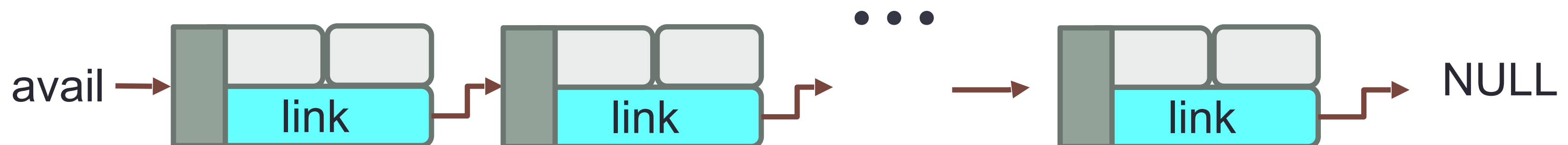
```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



Erasing in a circular list: **Special case of zero polynomial only**

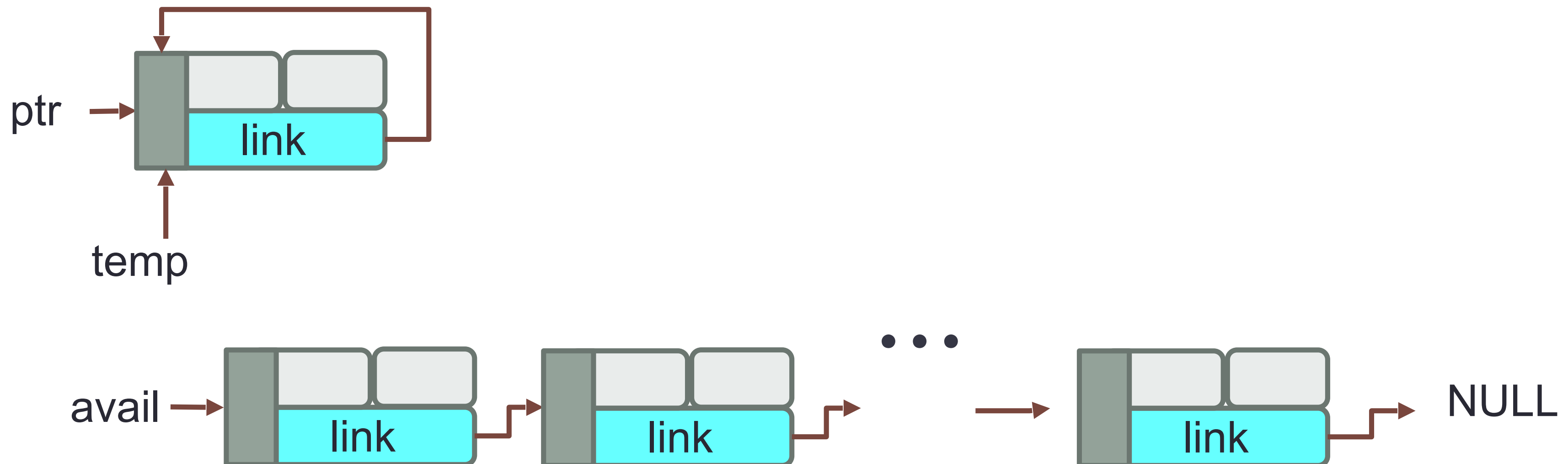
```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```





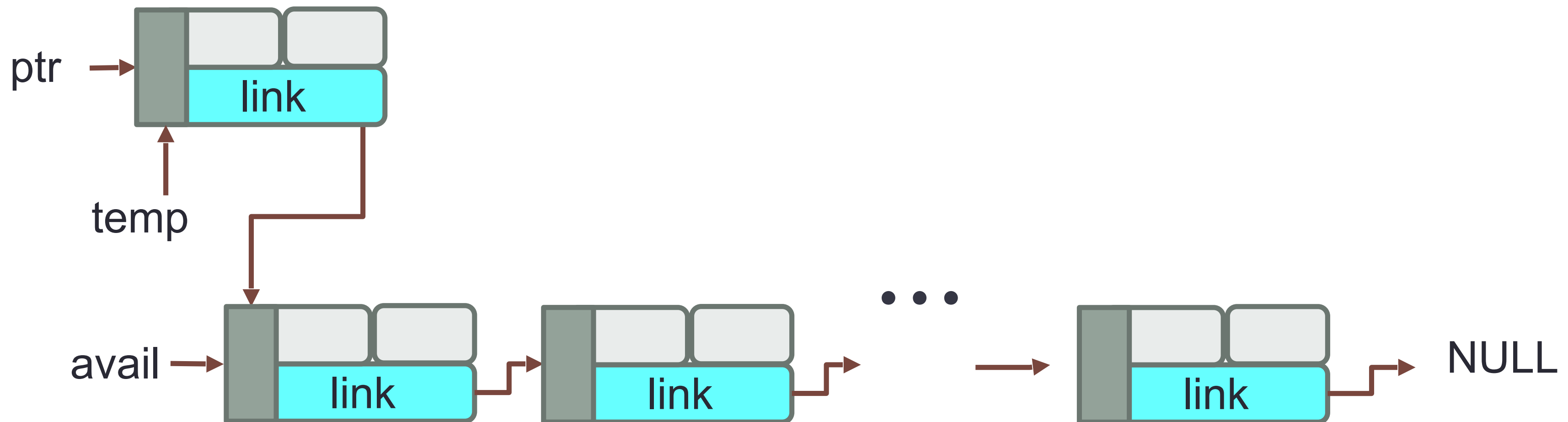
Erasing in a circular list: **Special case of zero polynomial only**

```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



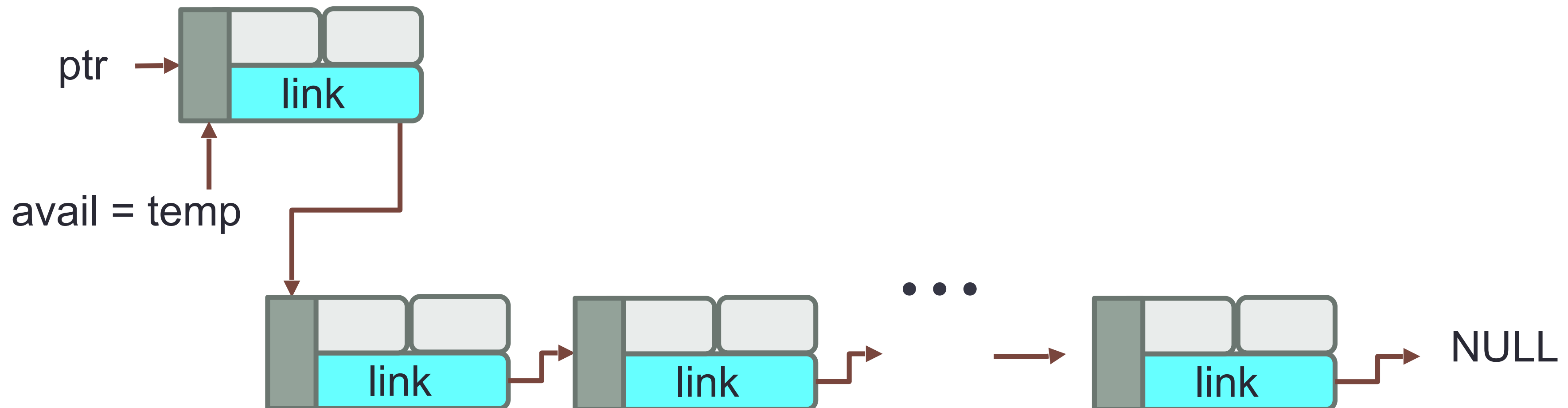
Erasing in a circular list: Special case of zero polynomial only

```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



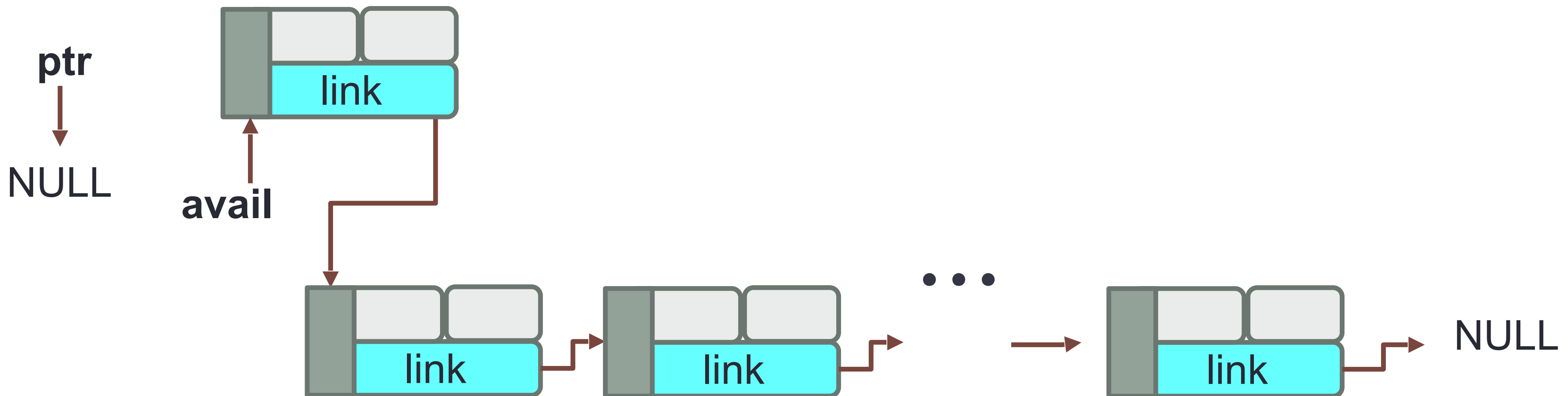
Erasing in a circular list: **Special case of zero polynomial only**

```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



Erasing in a circular list: **Special case of zero polynomial only**

```
void cerase(POLYPTR *ptr){
/*erase the circular list ptr*/
    POLYPTR temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



```c
POLYPTR cpadd(POLYPTR a, POLYPTR b){
/* polynomials a and b are singly linked circular lists with a head node.
Return a polynomial which is the sum of a and b */
     POLYPTR start_a, d, last_d;
     int sum, done = FALSE;
     start_a = a;            /*record start of a*/
     a = a->link;            /*skip headnode for a and b*/
     b = b->link;
     d = get_node();         /*get a head node for sum*/
     d->expon = -1; last_d = d;
     do{
       switch (COMPARE(a->expon,b->expon)){
         case -1: /* a->expon < b->expon */
              attach(b->coef, b->expon, &last_d);
              b = b->link; break;
         case 0: /* a->expon == b->expon */
              if(start_a == a) done = TRUE;
              else{
                   sum = a->coef + b->coef;
                   if(sum) attach(sum,a->expon,&last_d);
                   a = a->link; b = b->link; }
              break;
         case 1: /* a->expon > b->expon */
              attach(a->coef,a->expon,&last_d);
              a = a->link;
       }
     }while(!done);
     last_d->link = d;
     return d;
}
```