1. **The records of the Indian citizens are stored in a singly linked list. As you start traversing this list, at some point in time you start suspecting that the list has become cyclic. Find out whether the list has indeed become cyclic and the node which is causing the problem. You’re working in a resource constrained environment and hence cannot create copies of this list, store addresses of already traversed nodes, .. Enumerate all the possible solutions that you can think of, and also implement the most optimal solution in C++. A working solution along with instructions on how to compile / build and execute, is expected. Also give details of the algorithmic complexity for each solution.**

Solution 1: **Brute Force**

Approach:

We can store nodes in a hash table so that, if a loop exists, the head will encounter the same node again. This node will be present in the table and hence, we can detect the loop. The steps are:-

Iterate the given list.

For each node visited by the head pointer, check if the node is present in the hash table.

If yes, the loop detected

If not, insert the node in the hash table and move the head pointer ahead.

If the head reaches null, then the given list does not have a cycle in it.

Time Complexity: O(N)

Reason: Iterating the entire list once.

Space Complexity: O(N)

Reason: We store all nodes in a hash table.

This solution can not be used here as the value of N is very big in this case.

As O(N) extra space is being used.

Solution 2: **Slow and Fast Pointer Method**

Approach:

The following steps are required:

Initially take two pointers, fast and slow. The fast pointer takes two steps ahead while the slow pointer will take a single step ahead for each iteration.

We know that if a cycle exists, fast and slow pointers will collide.

If the cycle does not exist, the fast pointer will move to NULL

Else, when both slow and fast pointer collides, it detects a cycle exists.

Take another pointer, say entry. Point to the very first of the linked list.

Move the slow and the entry pointer ahead by single steps until they collide.

Once they collide we get the starting node of the linked list.

But why use another pointer, or xentry?

Let’s say a slow pointer covers the L2 distance from the starting node of the cycle until it collides with a fast pointer. L1 is the distance traveled by the entry pointer to the starting node of the cycle. So, in total, the slow pointer covers the L1+L2 distance. We know that a fast pointer covers some steps more than a slow pointer. Therefore, we can say that a fast pointer will surely cover the L1+L2 distance. Plus, a fast pointer will cover more steps which will accumulate to nC length where cC is the length of the cycle and n is the number of turns. Thus, the fast pointer covers the total length of L1+L2+nC.

We know that the slow pointer travels twice the fast pointer. So this makes the equation to

2(L1+L2) = L1+L2+nC. This makes the equation to

L1+L2 = nC. Moving L2 to the right side

L1 = nC-L2 and this shows why the entry pointer and the slow pointer would collide.

**Code:**

#include<bits/stdc++.h>

using namespace std;

class node {

public:

int num;

node\* next;

node(int val) {

num = val;

next = NULL;

}

};

void insertNode(node\* &head,int val) {

node\* newNode = new node(val);

if(head == NULL) {

head = newNode;

return;

}

node\* temp = head;

while(temp->next != NULL) temp = temp->next;

temp->next = newNode;

return;

}

void createCycle(node\* &head,int pos) {

node\* ptr = head;

node\* temp = head;

int cnt = 0;

while(temp->next != NULL) {

if(cnt != pos) {

++cnt;

ptr = ptr->next;

}

temp = temp->next;

}

temp->next = ptr;

}

//process as per mentioned in solution

node\* detectCycle(node\* head) {

if(head == NULL||head->next == NULL) return NULL;

node\* fast = head;

node\* slow = head;

node\* entry = head;

while(fast->next != NULL&&fast->next->next != NULL) {

slow = slow->next;

fast = fast->next->next;

if(slow == fast) {

while(slow != entry) {

slow = slow->next;

entry = entry->next;

}

return slow;

}

}

return NULL;

}

int main() {

node\* head = NULL;

insertNode(head,1);

insertNode(head,2);

insertNode(head,3);

insertNode(head,4);

insertNode(head,3);

insertNode(head,6);

insertNode(head,10);

createCycle(head,2);

node\* nodeRecieve = detectCycle(head);

if(nodeRecieve == NULL) cout<<"No cycle";

else {

node\* temp = head;

int pos = 0;

while(temp!=nodeRecieve) {

++pos;

temp = temp->next;

}

cout<<"Tail connects at pos "<<pos<<endl;

}

return 0;

}

Explanation of the above code:

The above code have mainly four function and a class node.

Class node: It is used to create the skeleton of the linked list node.

void **insertNode** (node\* &head,int val) function – This function takes head of the linked list and the value with which the node is to be created. It inserts node at the end of the linked list.

void **createCycle**(node\* &head,int pos) function- This function creates a cycle in the linked list at the position passed as parameter in the function.

node\* **detectCycle**(node\* head) – This function detects if there is any cycle in the linked list and return the culprit node (the node which is causing the problem).

Int **main**()- This is the driver function where the execution starts, linked list is created, cycle is created and cycle is detected and culprit node is found.

Compiling the code:

The code can be compiled in any online compiler or in using any offline compiler.

Steps to compile in Linux using g++ compiler:

**g++ <file\_name> -o <object\_name>**

Running the executable object created - **./<object\_name>**

1. **Explain in detail the priority based pre-emptive scheduling algorithm. What are the pros and cons of this, and how can these shortcomings be overcome?**

In pre-emptive priority scheduling algorithm, every time a process with higher priority arrives in the waiting queue, the CPU cycle is shifted to the process with the highest priority. This is preemptive because a process that’s already being executed can be stopped to execute a process with higher priority.

Preemptive Priority CPU Scheduling Algorithm is a pre-emptive method of CPU scheduling algorithm that works based on the priority of a process. In this algorithm, the scheduler schedules the tasks to work as per the priority, which means that a higher priority process should be executed first. In case of any conflict, i.e., when there is more than one process with equal priorities, then the pre-emptive priority CPU scheduling algorithm works on the basis of FCFS (First Come First Serve) algorithm.

**Deciding Priority of the process**:

This algorithm uses a rank-based system to define a rank for each process, where lower rank processes have higher priority and higher rank processes have lower priority. For instance, if there are 10 processes to be executed using this Preemptive Algorithm, then process with rank 1 will have the highest priority, the process with rank 2 will have comparatively lesser priority, and process with rank 10 will have least priority.

Advantages of priority based pre-emptive scheduling algorithm:

* Good way to ensure processes with higher priorities are handled first.
* Good when the resources are limited and priorities for each process are defined beforehand

Disadvantages of priority based pre-emptive scheduling algorithm:

* Processes with lower priority may be starved
* Difficult to objectively decide which processes are given higher priority
* Low priority processes will be lost if the computer crashes

**Overcoming the problem of Starvation: Aging**

Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priority range from 127(low) to 0(high), we could increase the priority of a waiting process by 1 Every 15 minutes. Eventually, even a process with an initial priority of 127 would take no more than 32 hours for the priority 127 process to age to a priority-0 process.

This technique will also ensure to some extent that Low priority processes will not be lost if the computer crashes.

**Overcoming the problem of priority deciding**

Resource requirements: Memory, time, or other resource requirements

I/O to CPU burst time ratio: The average I/O to average CPU burst time ratio

Internal priorities: Assigned by the OS based on average burst time, CPU to I/O activity ratio, and other factors

External priorities: Assigned by users based on the importance of the job, fees paid, and other factors

Using the above factors effectively can be used to overcome the problem of priority deciding