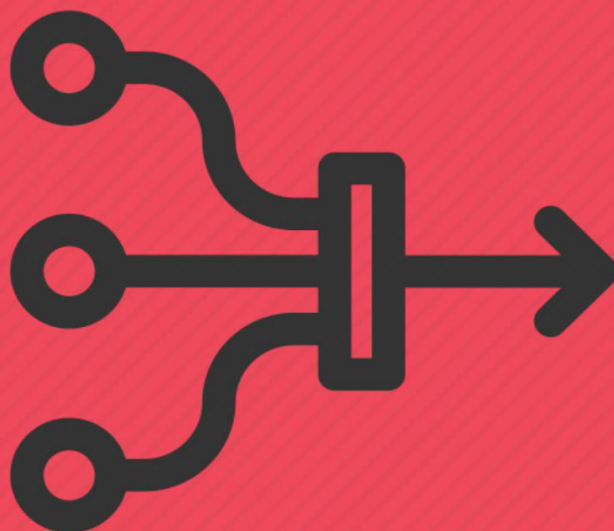


# Design & Analysis of Algorithms



**LEARN IN 1 DAY**

**KRISHNA RUNGTA**

# Learn Design and Analysis of Algorithms in 1 Day

By Krishna Rungta

Copyright 2022 - All Rights Reserved – Krishna Rungta

**ALL RIGHTS RESERVED.** No part of this publication may be reproduced or transmitted in any form whatsoever, electronic, or mechanical, including photocopying, recording, or by any informational storage or retrieval system without express written, dated and signed permission from the author.

# Table Of Content

## **Chapter 1: Greedy Algorithm with Example: What is, Method and Approach**

1. [What is a Greedy Algorithm?](#)
2. [History of Greedy Algorithms](#)
3. [Greedy Strategies and Decisions](#)
4. [Characteristics of the Greedy Approach](#)
5. [Why use the Greedy Approach?](#)
6. [How to Solve the activity selection problem](#)
7. [Architecture of the Greedy approach](#)
8. [Disadvantages of Greedy Algorithms](#)

## **Chapter 2: Circular Linked List: Advantages and Disadvantages**

1. [What is a Circular Linked List?](#)
2. [Basic Operations in Circular Linked lists](#)
3. [Insertion Operation](#)
4. [Deletion Operation](#)
5. [Traversal of a Circular Linked List](#)
6. [Advantages of Circular Linked List](#)
7. [Disadvantages of Circular Linked List](#)
8. [Singly Linked List as a Circular Linked List](#)
9. [Applications of the Circular Linked List](#)

## **Chapter 3: Array in Data Structure: What is, Arrays Operations [Examples]**

1. [What are Arrays?](#)

2. [Concept of Array](#)
3. [Why do we need arrays?](#)
4. [Creating an Array in Python](#)
5. [Ways to Declare an Array in Python](#)
6. [Array Operations](#)
7. [Creating an Array in C++](#)
8. [Array Operations in C++](#)
9. [Array Operations in Java](#)

## **Chapter 4: B TREE in Data Structure: Search, Insert, Delete Operation Example**

1. [What is a B Tree?](#)
2. [Why use B-Tree](#)
3. [History of B Tree](#)
4. [Search Operation](#)
5. [Insert Operation](#)
6. [Delete Operation](#)

## **Chapter 5: B+ TREE : Search, Insert and Delete Operations Example**

1. [What is a B+ Tree?](#)
2. [Rules for B+ Tree](#)
3. [Why use B+ Tree](#)
4. [B+ Tree vs. B Tree](#)
5. [Search Operation](#)
6. [Insert Operation](#)
7. [Delete Operation](#)

## **Chapter 6: Breadth First Search (BFS) Algorithm with EXAMPLE**

1. [What is BFS Algorithm \(Breadth-First Search\)?](#)
2. [What is Graph traversals?](#)
3. [The architecture of BFS algorithm](#)
4. [Why do we need BFS Algorithm?](#)
5. [How does BFS Algorithm Work?](#)
6. [Example BFS Algorithm](#)
7. [Rules of BFS Algorithm](#)
8. [Applications of BFS Algorithm](#)

## **Chapter 7: Binary Search Tree (BST) with Example**

1. [What is a Binary Search Tree?](#)
2. [Attributes of Binary Search Tree](#)
3. [Why do we need a Binary Search Tree?](#)
4. [Types of Binary Trees](#)
5. [How Binary Search Tree Works?](#)
6. [Important Terms](#)

## **Chapter 8: Binary Search Algorithm with EXAMPLE**

1. [What is Search?](#)
2. [What is Binary Search?](#)
3. [How Binary Search Works?](#)
4. [Example Binary Search](#)
5. [Why Do We Need Binary Search?](#)

## **Chapter 9: Linear Search: Python, C++ Example**

1. [What is Searching Algorithm?](#)
2. [What is Linear Search?](#)
3. [What does Linear Search Function do?](#)

4. [How does Linear Search work?](#)
5. [Pseudo Code for Sequential Search Algorithm](#)
6. [C++ Code Example Linear Search](#)
7. [Python Code Example Linear Search](#)
8. [Complexity Analysis of Linear Search Algorithm](#)
9. [How to improve Linear Search Algorithm](#)
10. [Application of Linear Search Algorithm](#)

## **Chapter 10: Bubble Sort Algorithm with Python using List Example**

1. [What is a Bubble Sort?](#)
2. [Implementing the Bubble Sort Algorithm](#)
3. [Optimized Bubble Sort Algorithm](#)
4. [Visual Representation](#)
5. [Python Examples](#)
6. [Code Explanation](#)
7. [Bubble sort advantages](#)
8. [Bubble sort Disadvantages](#)
9. [Complexity Analysis of Bubble Sort](#)

## **Chapter 11: Selection Sort: Algorithm explained with Python Code Example**

1. [What is Selection Sort?](#)
2. [How does selection sort work?](#)
3. [Problem Definition](#)
4. [Solution \(Algorithm\)](#)
5. [Visual Representation](#)
6. [Selection Sort Program using Python 3](#)
7. [Code Explanation](#)
8. [Time Complexity Of Selection Sort](#)
9. [When to use selection sort?](#)

10. [Advantages of Selection Sort](#)
11. [Disadvantages of Selection Sort](#)

## **Chapter 12: Hash Table in Data Structure: Python Example**

1. [What is Hashing?](#)
2. [What is a Hash Table?](#)
3. [Hash functions](#)
4. [Qualities of a good hash function](#)
5. [Collision](#)
6. [Hash table operations](#)
7. [Hash Table Implementation with Python Example](#)
8. [Hash Table Code Explanation](#)
9. [Python Dictionary Example](#)
10. [Complexity Analysis](#)
11. [Real-world Applications](#)
12. [Advantages of hash tables](#)
13. [Disadvantages of hash tables](#)

## **Chapter 13: Tree Traversals (Inorder, Preorder, Postorder): C,Python, C++ Examples**

1. [What is Tree Traversal?](#)
2. [Types of Tree Traversal](#)
3. [Breadth-First Traversal](#)
4. [Inorder Traversal Binary Tree](#)
5. [Post-Order Traversal](#)
6. [Preorder Traversal](#)
7. [Implementation in Python:](#)
8. [Implementation in C:](#)
9. [Implementation of C++ \(Using std:queue for level order\):](#)

## **Chapter 14: Binary Tree in Data Structure (EXAMPLE)**

1. [What is a Binary Tree?](#)
2. [What are the Differences Between Binary Tree and Binary Search Tree?](#)
3. [Example of Binary Search Trees](#)
4. [Types of Binary Tree:](#)
5. [Implementation of Binary Tree in C and C++:](#)
6. [Implementation of Binary Tree in Python](#)
7. [Application of Binary Tree:](#)

## **Chapter 15: Combination Algorithm: Print all possible combinations of r | C,C++,Python**

1. [What is the Combination?](#)
2. [The time complexity analysis for Combination](#)
3. [METHOD-1: Fixed element with recursion](#)
4. [Method 2 \(Include and Exclude every element\):](#)
5. [Handling Duplicate Combinations](#)
6. [Using a dictionary or unordered map to track duplicate combinations](#)

## **Chapter 16: Longest Common Subsequence: Python, C++ Example**

1. [What is Longest Common Subsequence?](#)
2. [Naive Method](#)
3. [Optimal Substructure](#)
4. [Recursive Method of Longest Comm Sequence](#)
5. [Dynamic Programming method of Longest Common Subsequence \(LCS\)](#)



## **Chapter 17: Dijkstra's Algorithm: C++, Python** **Code Example**

1. [What is the shortest path or shortest distance?](#)
2. [How Dijkstra's Algorithm Works](#)
3. [Difference Between Dijkstra and BFS, DFS](#)
4. [2D grid demonstration of how BFS works](#)
5. [Example of Dijkstra's Algorithm](#)
6. [C++ implementation Dijkstra's Algorithm](#)
7. [Python implementation Dijkstra's Algorithm](#)
8. [Application of Dijkstra Algorithm](#)
9. [Limitation of Dijkstra's Algorithm](#)

# Chapter 1: Greedy Algorithm with Example: What is, Method and Approach

## What is a Greedy Algorithm?

In **Greedy Algorithm** a set of resources are recursively divided based on the maximum, immediate availability of that resource at any given stage of execution. To solve a problem based on the greedy approach, there are two stages

1. Scanning the list of items
2. Optimization

These stages are covered parallelly in this Greedy algorithm tutorial, on course of division of the array.

To understand the greedy approach, you will need to have a working knowledge of recursion and context switching. This helps you to understand how to trace the code. You can define the greedy paradigm in terms of your own necessary and sufficient statements. Two conditions define the greedy paradigm.

- Each stepwise solution must structure a problem towards its best-accepted solution.
- It is sufficient if the structuring of the problem can halt in a finite number of greedy steps.

With the theorizing continued, let us describe the history associated with the Greedy search approach.

# History of Greedy Algorithms

Here is an important landmark of greedy algorithms:

- Greedy algorithms were conceptualized for many graph walk algorithms in the 1950s.
- Esdger Dijkstra conceptualized the algorithm to generate minimal spanning trees. He aimed to shorten the span of routes within the Dutch capital, Amsterdam.
- In the same decade, Prim and Kruskal achieved optimization strategies that were based on minimizing path costs along weighed routes.
- In the '70s, American researchers, Cormen, Rivest, and Stein proposed a recursive substructuring of greedy solutions in their classical introduction to algorithms book.
- The Greedy search paradigm was registered as a different type of optimization strategy in the NIST records in 2005.
- Till date, protocols that run the web, such as the open-shortest-path-first (OSPF) and many other network packet switching protocols use the greedy strategy to minimize time spent on a network.

## Greedy Strategies and Decisions

Logic in its easiest form was boiled down to “greedy” or “not greedy”. These statements were defined by the approach taken to advance in each algorithm stage. For example, Djikstra’s algorithm utilized a stepwise greedy strategy identifying hosts on the Internet by calculating a cost function. The value returned by the cost function determined whether the next path is “greedy” or “non-greedy”. In short, an algorithm ceases to be greedy if at any stage it takes a step that is not locally greedy. The Greedy problems halt with no further scope of greed.



- In the activity selection problem, the “recursive division” step is achieved by scanning a list of items only once and considering certain activities.

## How to Solve the activity selection problem

In the activity scheduling example, there is a “start” and “finish” time for every activity. Each Activity is indexed by a number for reference. There are two activity categories.

1. **considered activity:** is the Activity, which is the reference from which the ability to do more than one remaining Activity is analyzed.
2. **remaining activities:** activities at one or more indexes ahead of the considered activity.

The total duration gives the cost of performing the activity. That is (finish - start) gives us the duration as the cost of an activity. You will learn that the greedy extent is the number of remaining activities you can perform in the time of a considered activity.

## Architecture of the Greedy approach

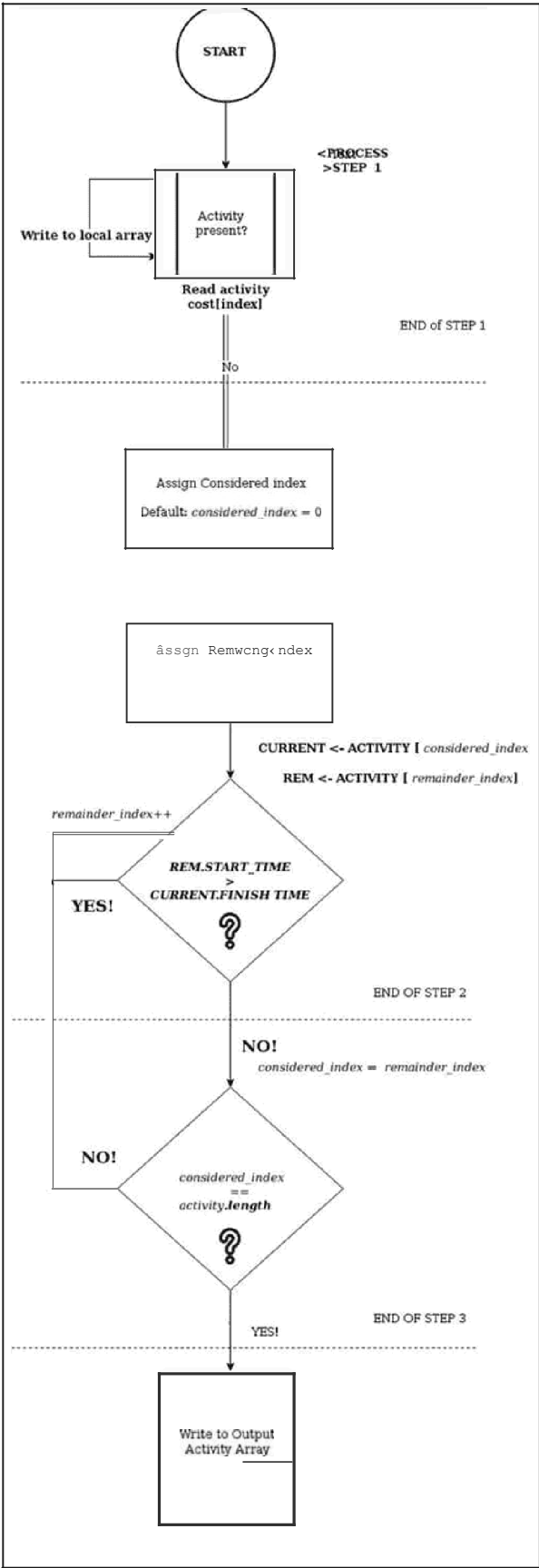
**STEP 1)** Scan the list of activity costs, starting with index 0 as the considered Index.

**STEP 2)** When more activities can be finished by the time, the considered activity finishes, start searching for one or more remaining activities.

**STEP 3)** If there are no more remaining activities, the current remaining activity becomes the next considered activity.

Repeat step 1 and step 2, with the new considered activity. If there are no remaining activities left, go to step 4.

**STEP 4 )** Return the union of considered indices. These are the activity indices that will be used to maximize throughput.

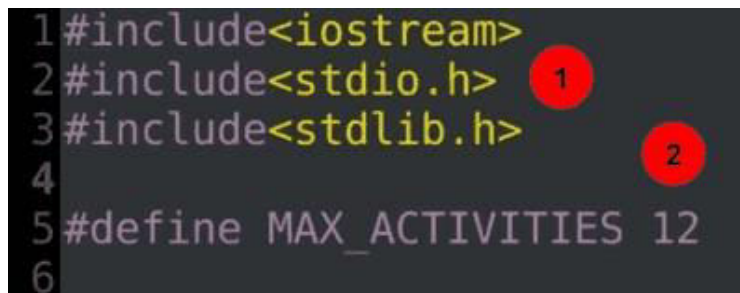


## Architecture of the Greedy Approach

### Code Explanation

```
#include<iostream>
#include<stdio.h>
#include<stdlib.h>

#define MAX_ACTIVITIES 12
```



```
1#include<iostream>
2#include<stdio.h>
3#include<stdlib.h>
4
5#define MAX_ACTIVITIES 12
6
```

#### Explanation of code:

1. Included header files/classes
2. A max number of activities provided by the user.

```
using namespace std;

class TIME
{
public:
int hours;

public: TIME()
{
hours = 0;
}
};
```



```
8 using namespace std; 1
9
10 class TIME 2
11 {
12     public: 3
13     int hours;
14
15     public: TIME() 4
16     {
17         hours = 0;
18     } 5
19};
```

### Explanation of code:

1. The namespace for streaming operations.
2. A class definition for TIME
3. An hour timestamp.
4. A TIME default constructor
5. The hours variable.

```
class Activity
{
public:
int index;
TIME start;
TIME finish;

public: Activity()
{
start = finish = TIME();
}
};
```

```
21 class Activity 1
22 {
23     public:
24     int index; 2
25     TIME start;
26     TIME finish;
27
28     public: Activity() 3
29     {
30         start = finish = TIME();
31     }
32 };
```

### Explanation of code:

1. A class definition from activity
2. Timestamps defining a duration
3. All timestamps are initialized to 0 in the default constructor

```
class Scheduler
{
public:
int considered_index,init_index;
Activity *current_activities = new
Activity[MAX_ACTIVITIES];
Activity *scheduled;
```

```
35 class Scheduler 1
36 {
37     public: 2
38     int considered_index,init_index; 3
39     Activity *current_activities = new Activity[MAX_ACTIVITIES]; 4
40     Activity *scheduled; 5
41 }
```

### Explanation of code:

1. Part 1 of the scheduler class definition.
2. Considered Index is the starting point for scanning the array.
3. The initialization index is used to assign random timestamps.
4. An array of activity objects is dynamically allocated using the new operator.

5. The scheduled pointer defines the current base location for greed.

```
Scheduler()  
{  
    considered_index = 0;  
    scheduled = NULL;  
    ...  
    ...  
}
```

```
42 Scheduler() 1  
43 { 2  
44     considered_index = 0;  
45     scheduled = NULL; 3  
46 }
```

### Explanation of code:

1. The scheduler constructor - part 2 of the scheduler class definition.
2. The considered index defines the current start of the current scan.
3. The current greedy extent is undefined at the start.

```
for(init_index = 0; init_index < MAX_ACTIVITIES;  
    init_index++)  
{  
    current_activities[init_index].start.hours =  
        rand() % 12;  
  
    current_activities[init_index].finish.hours =  
        current_activities[init_index].start.hours +  
        (rand() % 2);  
  
    printf("\nSTART:%d END %d\n",  
        current_activities[init_index].start.hours,  
        current_activities[init_index].finish.hours);  
}  
...  
...
```

```
47 1 for(init_index = 0; init_index < MAX_ACTIVITIES; init_index++)
48 {
49     2 current_activities[init_index].start.hours =
50         rand() % 12;
51
52     3 current_activities[init_index].finish.hours =
53         current_activities[init_index].start.hours +
54         (rand() % 2);
55
56     4 printf("\nSTART:%d END %d\n",
57         current_activities[init_index].start.hours,
58         current_activities[init_index].finish.hours);
59 }
60
```

### Explanation of code:

1. A for loop to initialize start hours and end hours of each of the activities currently scheduled.
2. Start time initialization.
3. End time initialization always after or exactly at the start hour.
4. A debug statement to print out allocated durations.

```
public:
Activity * activity_select(int);
};
```

```
62 1 public:
63     Activity * activity_select(int); 2
64};
```

### Explanation of code:

1. Part 4 - the last part of the scheduler class definition.
2. Activity select function takes a starting point index as the base and divides the greedy quest into greedy subproblems.

```
Activity * Scheduler :: activity_select(int
considered_index)
{
this->considered_index = considered_index;
int greedy_extent = this->considered_index + 1;
...
...
}
```

```
72 Activity * Scheduler :: activity_select(int considered_index)
73 {
74     1 this->considered_index = considered_index; 2
75     int greedy_extent = this->considered_index + 1; 3
```

1. Using the scope resolution operator (::), the function definition is provided.
2. The considered Index is the Index called by value. The greedy\_extent is the initialized just an index after the considered Index.

```
Activity * Scheduler :: activity_select(int
considered_index)
{
while( (greedy_extent < MAX_ACTIVITIES ) &&
((this->current_activities[greedy_extent]).start.hours <
(this->current_activities[considered_index]).finish.hours
))
{
printf("\nSchedule start:%d \nfinish%d\n activity:%d\n",
(this->current_activities[greedy_extent]).start.hours,
(this->current_activities[greedy_extent]).finish.hours,
greedy_extent + 1);
greedy_extent++;
}
...
...
```

```
76 while( (greedy_extent < MAX_ACTIVITIES ) &&
77         ((this->current_activities[greedy_extent]).start.hours <
78         2 (this->current_activities[considered_index]).finish.hour
79         {
80             printf("\nSchedule start:%d \nfinish%d\n activity:%d\n",
81                 3 (this->current_activities[greedy_extent]).start.hours,
82                 (this->current_activities[greedy_extent]).finish.hours,
83                 greedy_extent + 1);
84             4 greedy_extent++;
85         }
```

### Explanation of code:

1. The core logic- The greedy extent is limited to the number of activities.
2. The start hours of the current Activity are checked as schedulable before the considered Activity (given by considered index) would finish.
3. As long as this possible, an optional debug statement is printed.

#### 4. Advance to next index on the activity array

```
...
if ( greedy_extent <= MAX_ACTIVITIES )
{

return activity_select(greedy_extent);
}
else
{
return NULL;
}
}
```

```
86      1 if(greedy_extent <= MAX_ACTIVITIES)
87      {
88
89          2 return activity_select(greedy_extent);
90      }
91      else
92      {
93          3 return NULL;
94      }
95 }
```

#### Explanation of code:

1. The conditional checks if all the activities have been covered.
2. If not, you can restart your greedy with the considered Index as the current point. This is a recursive step that greedily divides the problem statement.
3. If yes, it returns to the caller with no scope for extending greed.

```
int main()
{
Scheduler *activity_sched = new Scheduler();
activity_sched->scheduled = activity_sched->
activity_select(
activity_sched->considered_index);
return 0;
}
```

```
97 int main() 1
98 {
99     Scheduler *activity_sched = new Scheduler(); 2
100     activity_sched->scheduled = activity_sched->activity_select(
101         activity_sched->considered_index);
102     return 0; 3
103 }
```

### Explanation of code:

1. The main function used to invoke the scheduler.
2. A new Scheduler is instantiated.
3. The activity select function, which returns a pointer of type activity comes back to the caller after the greedy quest is over.

### Output:

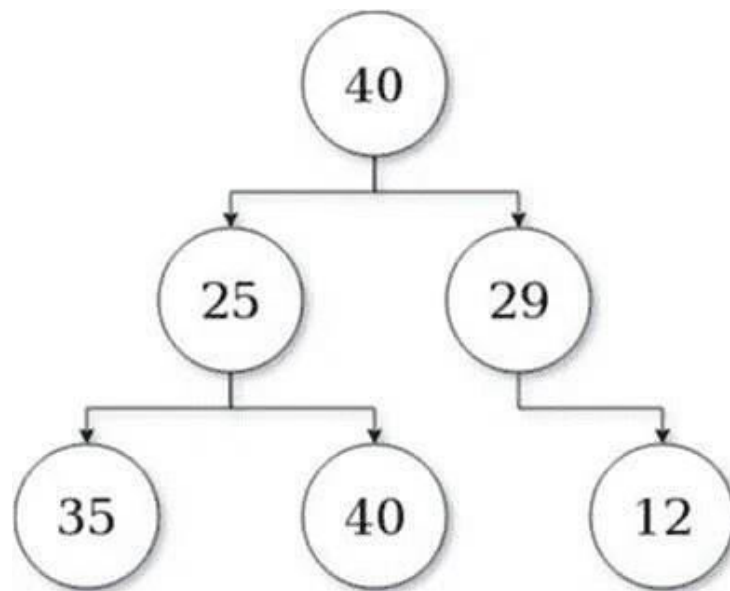
```
START:7 END 7
START:9 END 10
START:5 END 6
START:10 END 10
START:9 END 10
Schedule start:5
finish6
activity:3
Schedule start:9
finish10
activity:5
```

## Limitations of Greedy Technique

It is not suitable for Greedy problems where a solution is required for every subproblem like sorting. In such Greedy algorithm practice problems, the Greedy method can be wrong; in the worst case even lead to a non-optimal solution.

Therefore the disadvantage of greedy algorithms is using not knowing what lies ahead of the current greedy state.

Below is a depiction of the disadvantage of the Greedy method:



In the greedy scan shown here as a tree (higher value higher greed), an algorithm state at value: 40, is likely to take 29 as the next value. Further, its quest ends at 12. This amounts to a value of 41. However, if the algorithm took a sub-optimal path or adopted a conquering strategy. then 25 would be followed by 40, and the overall cost improvement would be 65, which is valued 24 points higher as a suboptimal decision.

## Examples of Greedy Algorithms

Most networking algorithms use the greedy approach. Here is a list of few Greedy algorithm examples:

- Prim's Minimal Spanning Tree Algorithm
- Travelling Salesman Problem
- Graph - Map Coloring
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Vertex Cover



- Knapsack Problem
- Job Scheduling Problem

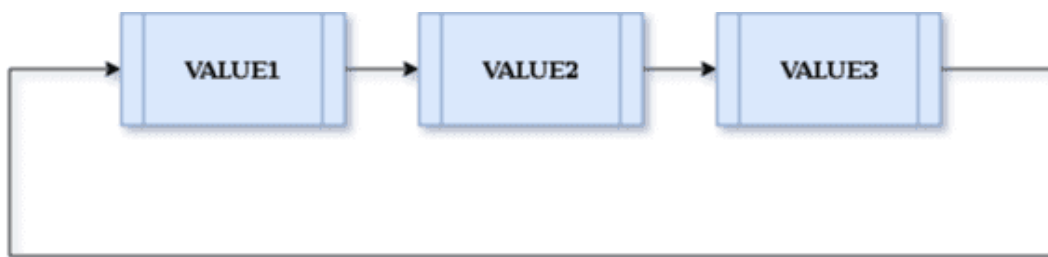
### **Summary:**

To summarize, the article defined the greedy paradigm, showed how greedy optimization and recursion, can help you obtain the best solution up to a point. The Greedy algorithm is widely taken into application for problem solving in many languages as Greedy algorithm Python, C, C#, PHP, Java, etc. The activity selection of Greedy algorithm example was described as a strategic problem that could achieve maximum throughput using the greedy approach. In the end, the demerits of the usage of the greedy approach were explained.

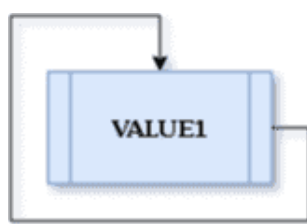
# Chapter 2: Circular Linked List: Advantages and Disadvantages

## What is a Circular Linked List?

A circular linked list is a sequence of nodes arranged such a way that each node can be retraced to itself. Here a “node” is a self-referential element with pointers to one or two nodes in it’s immediate vicinity. Below is a depiction of a circular linked list with 3 nodes.



Here, you can see that each node is retraceable to itself. The example shown above is a circular singly linked list. Note: The most simple circular linked list, is a node which traces only to itself as shown



In this circular linked list tutorial, you will learn:

## Basic Operations in Circular Linked lists

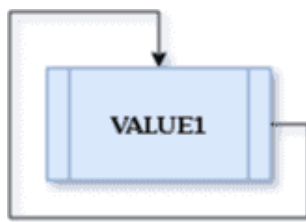
The basic operations on a circular linked list are:

1. Insertion
  2. Deletion and
  3. Traversal
- Insertion is the process of placing a node at a specified position in the circular linked list.
  - Deletion is the process of removing an existing node from the linked list. The node can be identified by the occurrence of its value or by its position.
  - Traversal of a circular linked list is the process of displaying the entire linked list's contents and retracing back to the source node.

In the next section, you will understand insertion of a node, and the types of insertion possible in a Circular Singly Linked List.

## Insertion Operation

Initially, you need to create one node which points to itself as shown in this image. Without this node, insertion creates the first node.



Next, there are two possibilities:

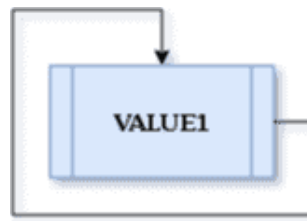
- Insertion at the current position of the circular linked list. This corresponds to insertion at the beginning of the end of a regular singular linked list. In a circular linked list, the beginning and the end are the same.

- Insertion after an indexed node. The node should be identified by an index number corresponding to its element value.

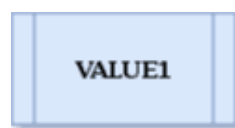
For inserting at the beginning/end of the circular linked list, that is at the position where the first-ever node was added,

- You will have to break the existing self-link to the existing node
- The new node's next pointer will link to the existing node.
- The last node's next pointer will point to the inserted node.

NOTE: The pointer that is the token master or the beginning/end of the circle can be changed. It will still return to the same node on a traversal, discussed ahead. Steps in (a) i-iii are shown below:



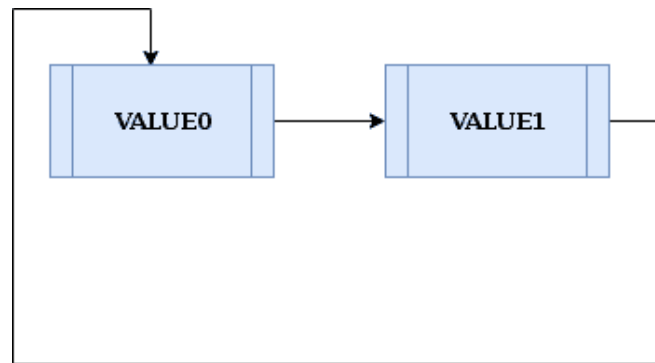
(Existing node)



**STEP 1)** Break the existing link



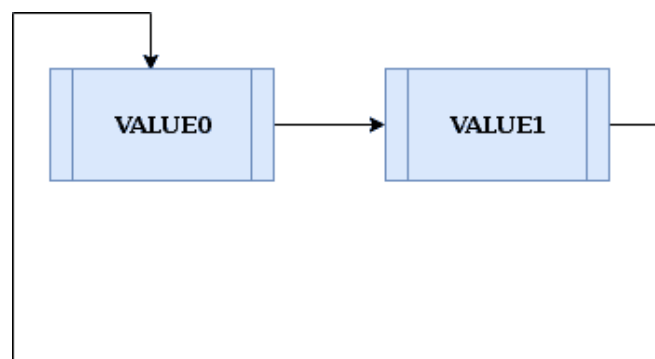
**STEP 2)** Create a forward link (from new node to an existing node)



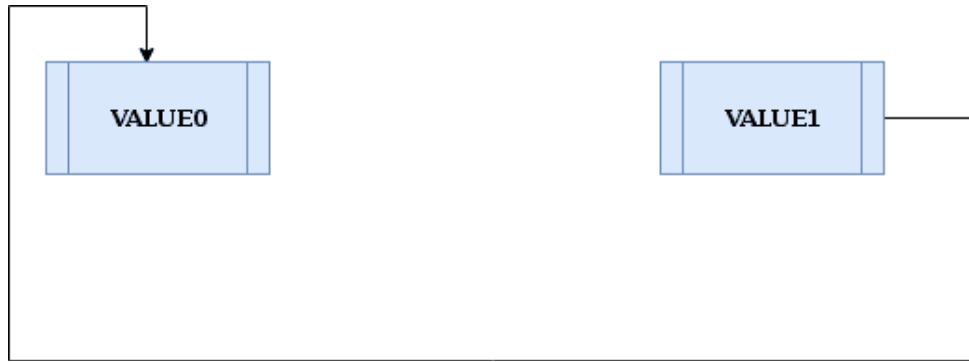
**STEP 3)** Create a loop link to the first node. Next, you will try insertion after a node. For example, let us insert “VALUE2” after the node with “VALUE0”. Let us assume that the starting point is the node with “VALUE0”.

- You will have to break the line between the first and second node and place the node with “VALUE2” in between.
- The first node’s next pointer must link to this node, and this node’s next pointer must link to the earlier second node.
- The rest of the arrangement remains unchanged. All nodes are retraceable to themselves.

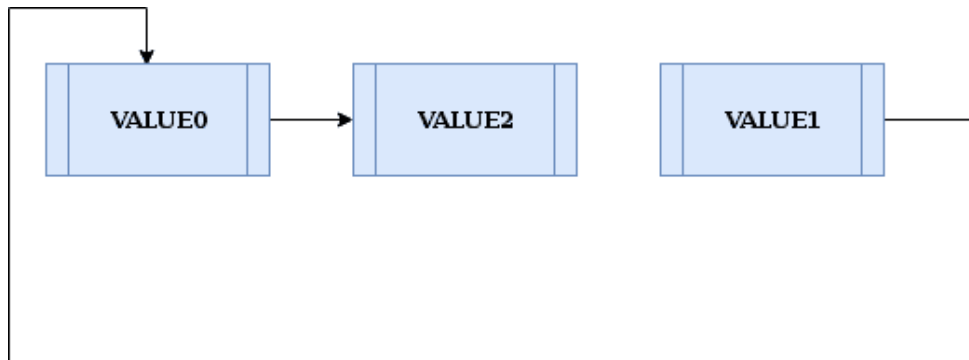
NOTE: Since there is a cyclic arrangement, inserting a node involves the same procedure for any node. The pointer that completes a cycle completes the cycle just like any other node. This is shown below:



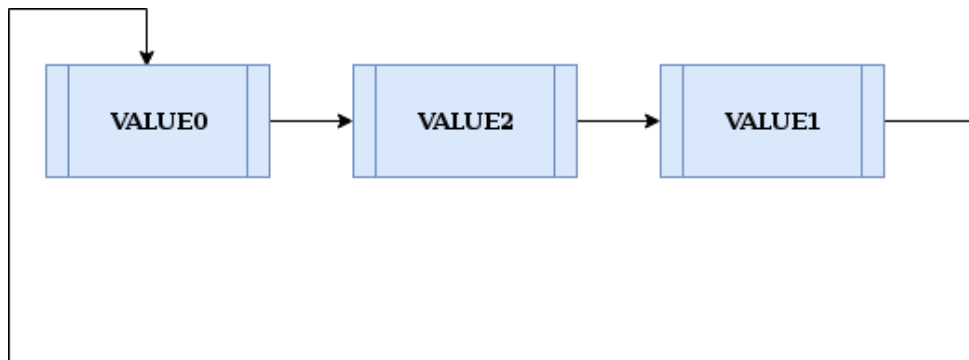
(Let us say there are only two nodes. This is a trivial case)



**STEP 1)** Remove the inner link between the connected nodes



**STEP 2)** Connect the left-hand side node to the new node



**STEP 3)** Connect the new node to the right hand side node.

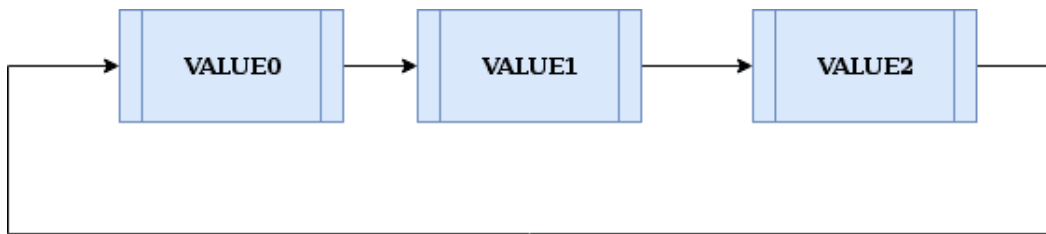
## Deletion Operation

Let us assume a 3-node circular linked list. The cases for deletion are given below:

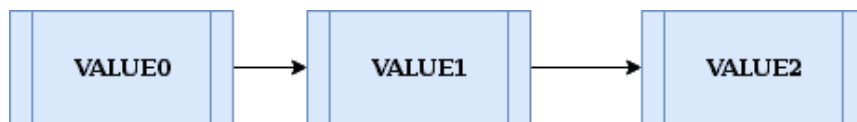
- Deleting the current element
- Deletion after an element.

Deletion at the beginning/end:

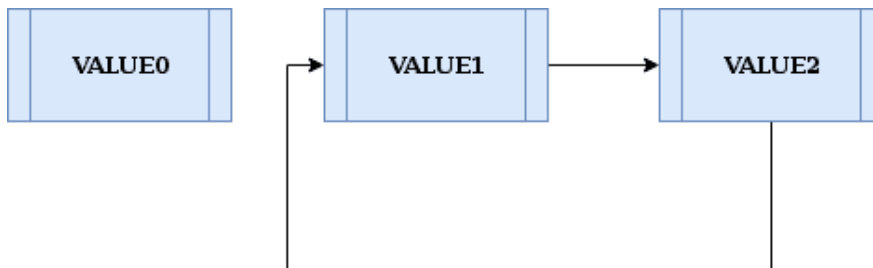
1. Traverse to the first node from the last node.
2. To delete from the end, there should be only one traversal step, from the last node to the first node.
3. Delete the link between the last node and the next node.
4. Link the last node to the next element of the first node.
5. Free the first node.



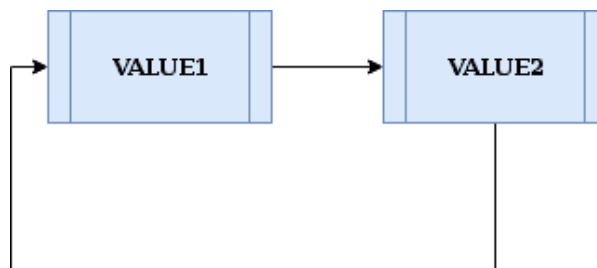
(Existing setup)



**STEP 1)** Remove the circular link



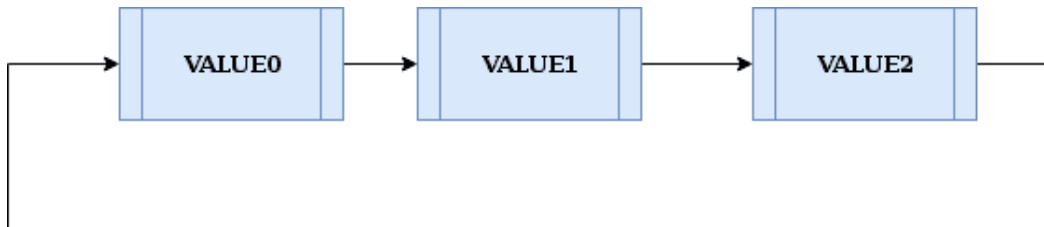
**STEPS 2)** Remove the link between the first and next, link the last node, to the node following the first



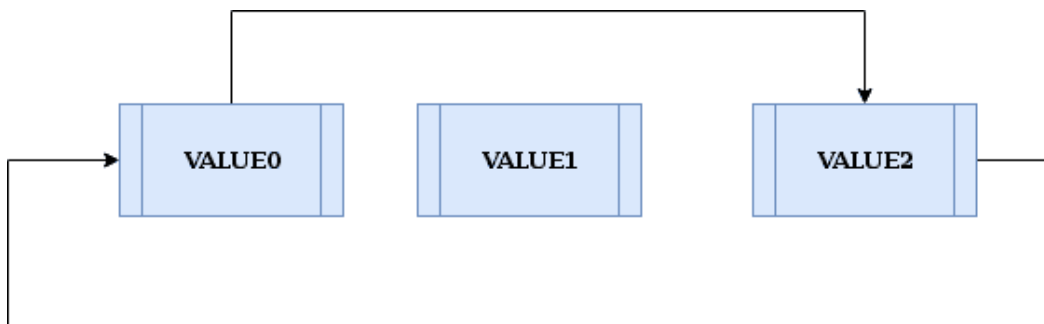
**STEP 3)** Free /deallocate the first node Deletion after a node:

1. Traverse till the next node is the node to be deleted.

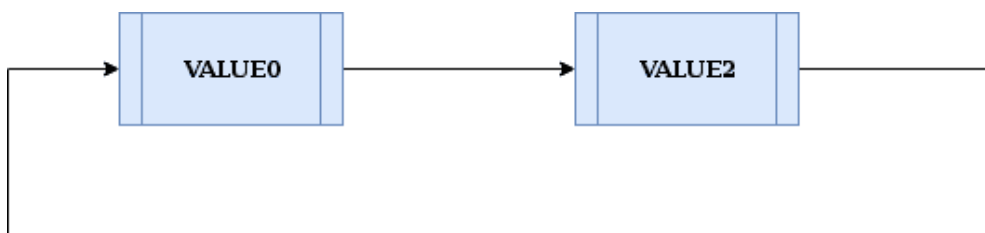
2. Traverse to the next node, placing a pointer on the previous node.
3. Connect the previous node to the node after the present node, using its next pointer.
4. Free the current (delinked) node.



**STEP 1)** Let us say that we need to delete a node with “VALUE1.”



**STEP 2)** Remove the link between the previous node and the current node. Link its previous node with the next node pointed by the current node (with VALUE1).

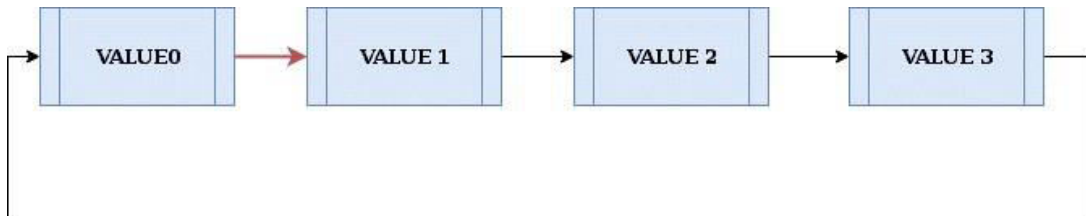


**STEP 3)** Free or deallocate the current node.

## Traversal of a Circular Linked List



To traverse a circular linked list, starting from a last pointer, check if the last pointer itself is NULL. If this condition is false, check if there is only one element. Otherwise, traverse using a temporary pointer till the last pointer is reached again, or as many times as needed, as shown in the GIF below.



## Advantages of Circular Linked List

Some of the advantages of circular linked lists are:

1. No requirement for a NULL assignment in the code. The circular list never points to a NULL pointer unless fully deallocated.
2. Circular linked lists are advantageous for end operations since beginning and end coincide. Algorithms such as the Round Robin scheduling can neatly eliminate processes which are queued in a circular fashion without encountering dangling or NULL-referential pointers.
3. Circular linked list also performs all regular functions of a singly linked list. In fact, circular doubly linked lists discussed below can even eliminate the need for a full-length traversal to locate an element. That element would at most only be exactly opposite to the start, completing just half the linked list.

## Disadvantages of Circular Linked List

The disadvantages in using a circular linked list are below:

1. Circular lists are complex as compared to singly linked lists.
2. Reverse of circular list is a complex as compared to singly or doubly lists.
3. If not handled carefully, then the code may go in an infinite loop.
4. Harder to find the end of the list and loop control.
5. Inserting at Start, we have to traverse the complete list to find the last node. (Implementation Perspective)

## Singly Linked List as a Circular Linked List

You are encouraged to attempt to read and implement the code below. It presents the pointer arithmetic associated with circular linked lists.

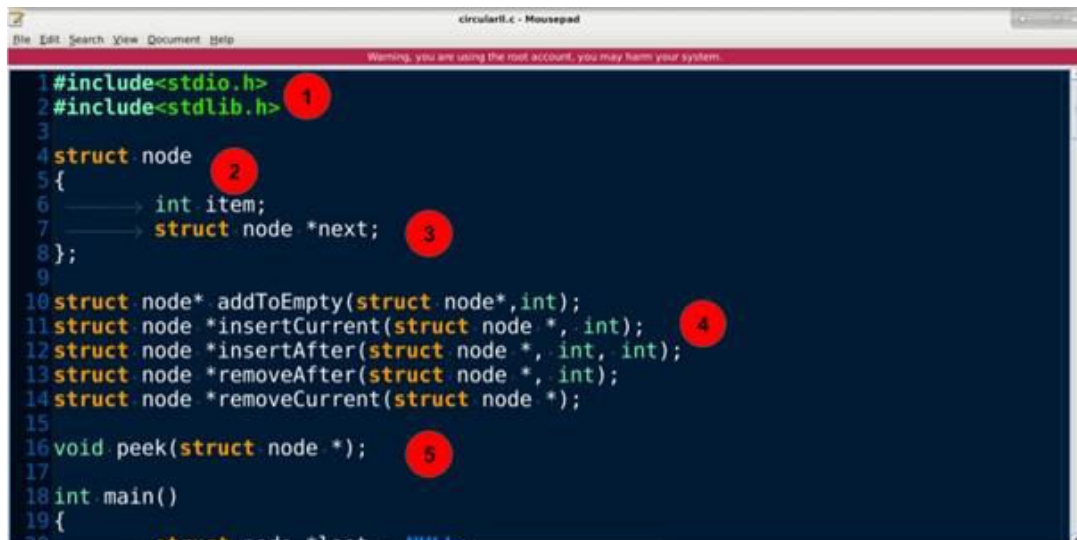
```
#include<stdio.h>
#include<stdlib.h>

struct node
{
int item;
struct node *next;
};

struct node* addToEmpty(struct node*,int);
struct node *insertCurrent(struct node *, int);
struct node *insertAfter(struct node *, int, int);
struct node *removeAfter(struct node *, int);
struct node *removeCurrent(struct node *);

void peek(struct node *);

int main()
{
...
}
```



```
1#include<stdio.h>
2#include<stdlib.h>
3
4struct node
5{
6    int item;
7    struct node *next;
8};
9
10struct node* addToEmpty(struct node*,int);
11struct node *insertCurrent(struct node *, int);
12struct node *insertAfter(struct node *, int, int);
13struct node *removeAfter(struct node *, int);
14struct node *removeCurrent(struct node *);
15
16void peek(struct node *);
17
18int main()
19{
20    struct node *last = NULL;
```

The screenshot shows a code editor window titled 'circularll.c - Mousepad'. The code is for a circular linked list. Red circles with numbers 1 through 5 are placed next to specific lines of code: 1 is next to the first include, 2 is next to the struct definition, 3 is next to the next pointer declaration, 4 is next to the insertCurrent function prototype, and 5 is next to the peek function prototype.

## Explanation of code:

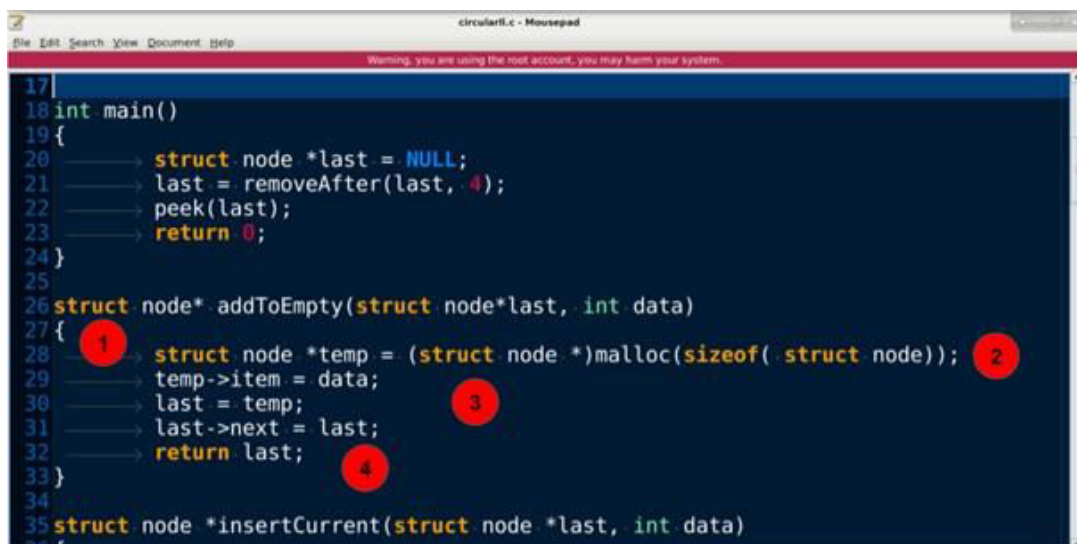
1. The first two lines of code are the necessary included header files.
2. The next section describes the structure of each self-referential node. It contains a value and a pointer of the same type as the structure.
3. Each structure repeatedly links to structure objects of the same type.
4. There are different function prototypes for:
  1. Adding an element to an empty linked list
  2. Inserting at the **currently pointed** position of a circular linked list.
  3. Inserting after a particular **indexed** value in the linked list.
  4. Removing/Deleting after a particular **indexed** value in the linked list.
  5. Removing at the currently pointed position of a circular linked list
5. The last function prints each element through a circular traversal at any state of the linked list.

```
int main()
{
```

```
struct node *last = NULL;
last = insertCurrent(last,4);
last = removeAfter(last, 4);
peek(last);
return 0;
}

struct node* addToEmpty(struct node*last, int data)
{
    struct node *temp = (struct node *)malloc(sizeof( struct
    node));
    temp->item = data;
    last = temp;
    last->next = last;
    return last;
}

struct node *insertCurrent(struct node *last, int data)
```



```
17|
18| int main()
19| {
20|     struct node *last = NULL;
21|     last = removeAfter(last, 4);
22|     peek(last);
23|     return 0;
24| }
25|
26| struct node* addToEmpty(struct node*last, int data)
27| {
28|     1 struct node *temp = (struct node *)malloc(sizeof( struct node)); 2
29|     temp->item = data;
30|     last = temp; 3
31|     last->next = last;
32|     return last; 4
33| }
34|
35| struct node *insertCurrent(struct node *last, int data)
```

### Explanation of code:

1. For the addEmpty code, allocate an empty node using the malloc function.
2. For this node, place the data to the temp variable.
3. Assign and link the only variable to the temp variable
4. Return the last element to the main() / application context.

```
struct node *insertCurrent(struct node *last, int data)
{
```

```
if(last == NULL)
{
return    addToEmpty(last, data);
}
struct node *temp = (struct node *)malloc(sizeof( struct
node));
temp -> item = data;
temp->next = last->next;
last->next = temp;
return last;
}
struct node *insertAfter(struct node *last, int data, int
item)
{
struct node *temp = last->next, *prev = temp, *newnode
=NULL;
...

```

```
circularll.c - Mousepad
Warning, you are using the root account, you may harm your system.
33 last->next = last;
34 return last;
35 }
36
37 struct node *insertCurrent(struct node *last, int data)
38 {
39     if(last == NULL)
40     {
41         return addToEmpty(last, data);
42     }
43     struct node *temp = (struct node *)malloc(sizeof( struct node));
44     temp -> item = data;
45     temp->next = last->next;
46     last->next = temp;
47     return last;
48 }
49
50 struct node *insertAfter(struct node *last, int data, int item)
51 {
52     struct node *temp = last->next, *prev = temp, *newnode = NULL;

```

## Explanation of code

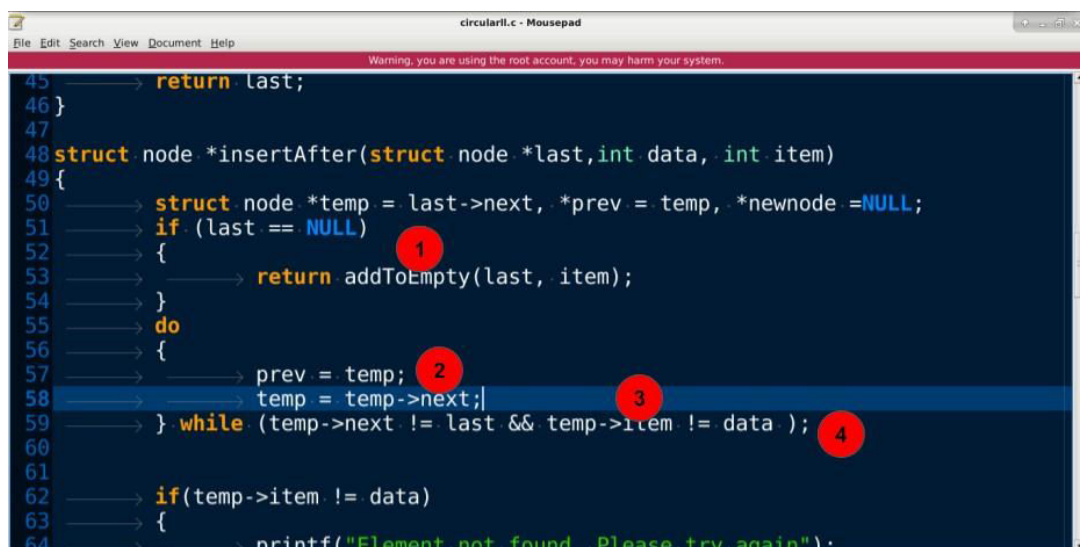
1. If there is no element to insert, then you should make sure to add to an empty list and return control.
2. Create a temporary element to place after the current element.
3. Link the pointers as shown.
4. Return the last pointer as in the previous function.

```
...
struct node *insertAfter(struct node *last, int data, int
item)

```

```
{
struct node *temp = last->next, *prev = temp, *newnode
=NULL;
if (last == NULL)
{
return addToEmpty(last, item);
}
do
{
prev = temp;
temp = temp->next;
} while (temp->next != last && temp->item != data );

if(temp->item != data)
{
printf("Element not found. Please try again");
...
}
```

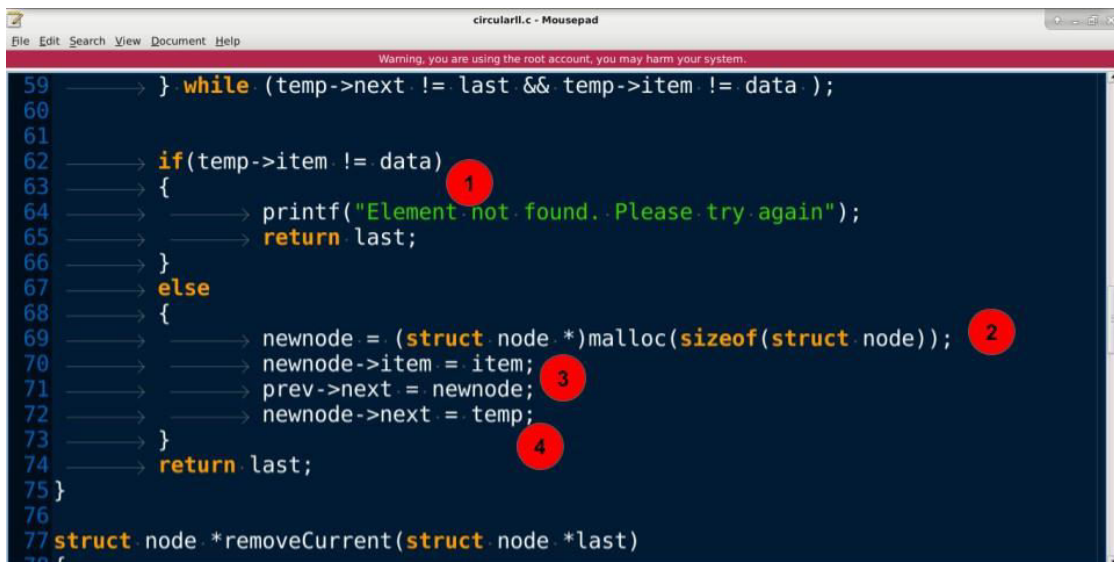


### Explanation of code:

1. If there is no element in the list, ignore the data, add the current item as the last item in the list and return control
2. For every iteration in the do-while loop ensure that there is a previous pointer that holds the last-traversed result.
3. Only then can the next traversal occur.
4. If the data is found, or temp reaches back to the last pointer, the do-while terminates. The next section of code decides what has to be done with the item.

```
...
if(temp->item != data)
{
printf("Element not found. Please try again");
return last;
}
else
{
newnode = (struct node *)malloc(sizeof(struct node));
newnode->item = item;
prev->next = newnode;
newnode->next = temp;
}
return last;
}

struct node *removeCurrent(struct node *last)
...
```



```
59  } while (temp->next != last && temp->item != data );
60
61
62  if(temp->item != data) 1
63  {
64      printf("Element not found. Please try again");
65      return last;
66  }
67  else
68  {
69      newnode = (struct node *)malloc(sizeof(struct node)); 2
70      newnode->item = item;
71      prev->next = newnode; 3
72      newnode->next = temp;
73  } 4
74  return last;
75 }
76
77 struct node *removeCurrent(struct node *last)
78 {
```

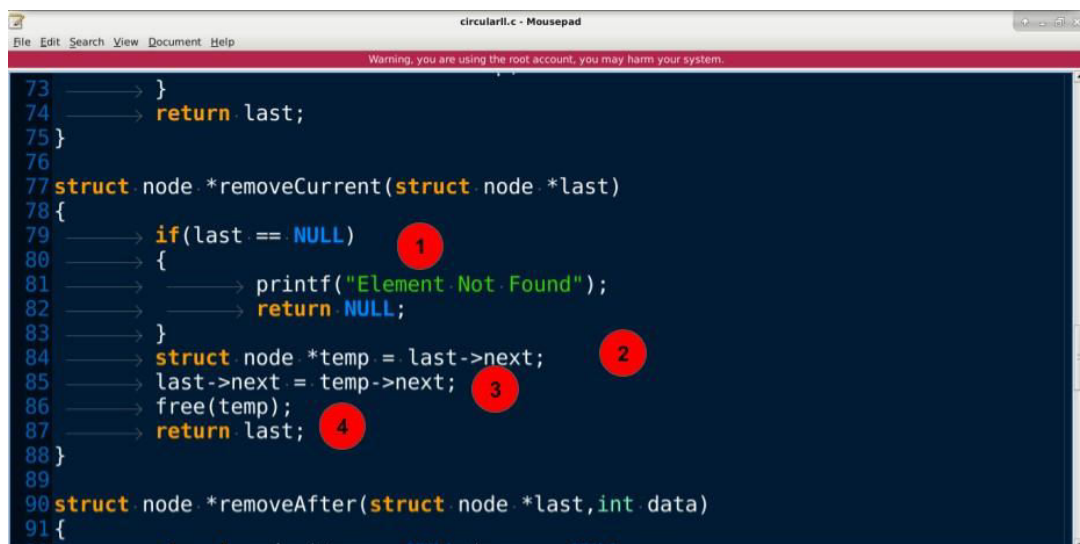
## Explanation of code:

1. If the entire list has been traversed, yet the item is not found, display an “item not found” message and then return control to the caller.
2. If there is a node found, and/or the node is not yet the last node, then create a new node.
3. **Link** the previous node with the new node. Link the current node with temp (the traversal variable).

4. This ensures that the element is placed after a particular node in the circular linked list. Return to the caller.

```
struct node *removeCurrent(struct node *last)
{
    if(last == NULL)
    {
        printf("Element Not Found");
        return NULL;
    }
    struct node *temp = last->next;
    last->next = temp->next;
    free(temp);
    return last;
}

struct node *removeAfter(struct node *last, int data)
```



## Explanation of code

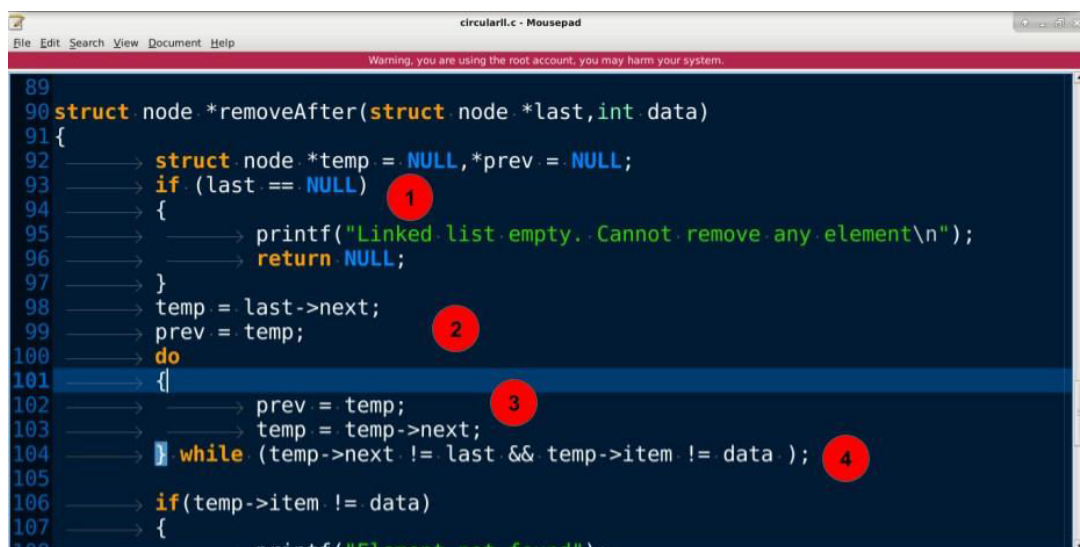
1. To remove only the last (current) node, check if this list is empty. If it is, then no element can be removed.
2. The temp variable just traverses one link forward.
3. Link the last pointer to the pointer after the first.
4. Free the temp pointer. It deallocates the un-linked last pointer.

```
struct node *removeAfter(struct node *last, int data)
{
```



```
struct node *temp = NULL,*prev = NULL;
if (last == NULL)
{
printf("Linked list empty. Cannot remove any element\n");
return NULL;
}
temp = last->next;
prev = temp;
do
{
prev = temp;
temp = temp->next;
} while (temp->next != last && temp->item != data );

if(temp->item != data)
{
printf("Element not found");
...
}
```



The screenshot shows a code editor window titled 'circularll.c - Mousepad'. The code is the same as the one in the previous block, but with four red circles and numbers 1 through 4 highlighting specific parts of the code:

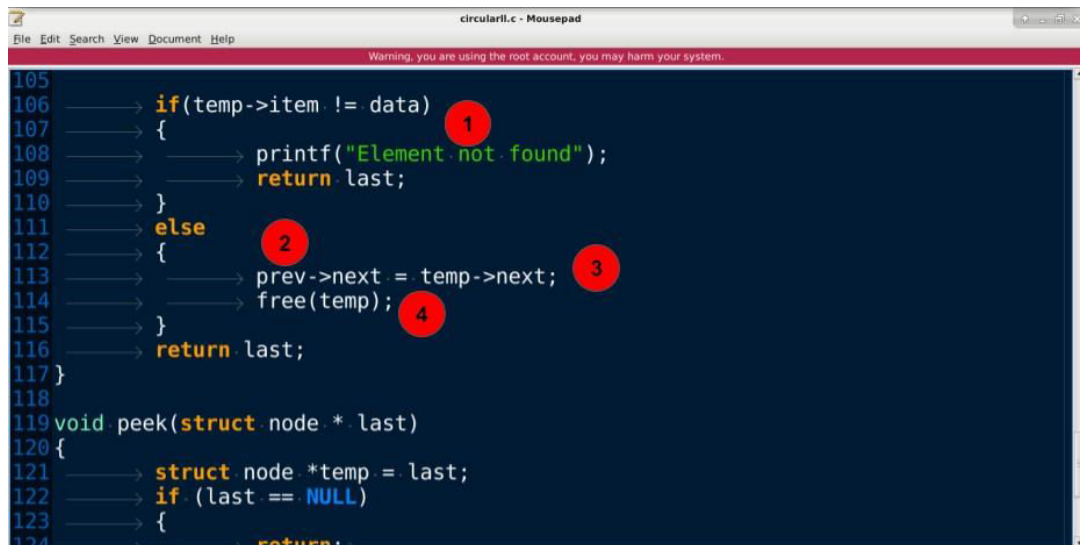
- 1: `if (last == NULL)`
- 2: `temp = last->next;`
- 3: `prev = temp;`
- 4: `while (temp->next != last && temp->item != data );`

## Explanation of code

1. As with the previous removal function, check if there is no element. If this is true, then no element can be removed.
2. Pointers are assigned specific positions to locate the element to be deleted.
3. The pointers are advanced, one behind the other. (prev behind temp)
4. The process continues until an element is found, or the next element retraces to the last pointer.

```
        if(temp->item != data)
        {
            printf("Element not found");
            return last;
        }
        else
        {
            prev->next = temp->next;
            free(temp);
        }
        return last;
    }

    void peek(struct node * last)
    {
        struct node *temp = last;
        if (last == NULL)
        {
            return;
        }
    }
```



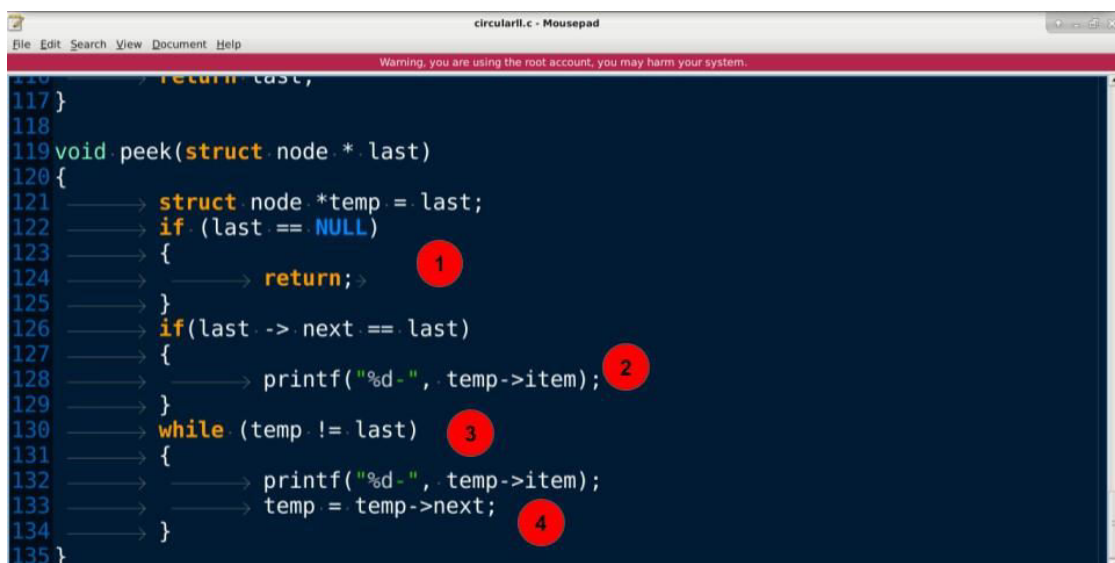
```
105
106  if(temp->item != data)
107  {
108      printf("Element not found");
109      return last;
110  }
111  else
112  {
113      prev->next = temp->next;
114      free(temp);
115  }
116  return last;
117 }
118
119 void peek(struct node * last)
120 {
121     struct node *temp = last;
122     if (last == NULL)
123     {
124         return;
125     }
126 }
```

## Explanation of program

1. If the element found after traversing the entire linked list, an error message is displayed saying the item was not found.
2. Otherwise, the element is delinked and freed in steps 3 and 4.

3. The previous pointer is linked to the address pointed as “next” by the element to be deleted (temp).
4. The temp pointer is therefore deallocated and freed.

```
...
void peek(struct node * last)
{
    struct node *temp = last;
    if (last == NULL)
    {
        return;
    }
    if(last -> next == last)
    {
        printf("%d-", temp->item);
    }
    while (temp != last)
    {
        printf("%d-", temp->item);
        temp = temp->next;
    }
}
```



## Explanation of code

1. The peek or traversal is not possible if there are zero needed. The user needs to allocate or insert a node.
2. If there is only one node, there is no need to traverse, the node's content can be printed, and the while loop does not

execute.

3. If there is more than one node, then the temp prints all the item till the last element.
4. Moment the last element is reached, the loop terminates, and the function returns call to the main function.

## **Applications of the Circular Linked List**

- Implementing round-robin scheduling in system processes and circular scheduling in high-speed graphics.
- Token rings scheduling in computer networks.
- It is used in display units like shop boards that require continuous traversal of data.

**Buy Now \$9.99**