Data Structures

Dilip Kumar Maripuri
Computer Applications

Data Structures

Session : Linked Lists - Circular Linked Lists

Dilip Kumar Maripuri
Computer Applications

- A circular linked list is a type of linked list in which the nodes are connected in a circle, rather than having a definite start and end.
- This structure allows for continuous traversal and has unique properties that differentiate it from other linked lists, such as singly or doubly linked lists.

Basic Structure and Characteristics

- **Node Structure:** Each node in a circular linked list contains two parts: the data and a pointer to the next node.
- **Circularity:** In a traditional singly linked list, the last node points to `NULL`, indicating the end of the list.
  In a circular linked list, the last node points back to the first node, creating a circular connection.
- **Traversal:** Unlike other linked lists, traversal can begin from any node and continue until reaching the starting node again.
  There is no natural end, as the structure is continuous.

Detailed Characteristics

- **Head Pointer:** A pointer (often named `head`) is typically maintained to indicate the starting point of the list.
  It helps in keeping track of the list and is crucial for operations like insertion and deletion at specific positions.

- **No Terminal Node:** Circular linked lists do not have a `NULL` reference as a termination point.
  This absence allows for uniform handling of nodes, where every node is connected in a circular way without any special end node.

- **Flexibility in Starting Point:** Due to circularity, traversal can start from any node, making it useful for applications that require repeated access to all nodes without reinitializing from a fixed starting point.

Insertion and Deletion Complexity

- **Insertion:** Adding a node to a circular linked list is similar to a singly linked list but requires careful handling of the pointers to maintain the circular structure, especially when inserting at the beginning or end.

- **Deletion:** Deleting a node is more complex as it requires adjusting the next pointers in a way that maintains the circularity.
  Special cases include deleting the head node, which often requires updating the head pointer.

Traversal Complexity

- Circular linked lists can lead to infinite loops if the traversal is not managed carefully.
- Typically, a check against the `head` pointer is used to ensure a full cycle has been completed.
- This allows traversal to detect when the starting point has been reached again, signaling the end of a complete iteration.

Pros of Using Singly Circular Linked Lists

- **Continuous Traversal:** Useful for applications needing repeated cycles through data (e.g., round-robin scheduling).
- **Consistency in Node Structure:** Since every node points to the next, with the last node pointing to the first, there are no special cases for handling the end of the list.
- **Memory Efficiency:** Similar to standard singly linked lists, memory usage is efficient as each node contains only one pointer.

Cons of Using Singly Circular Linked Lists

- **Complexity in Traversal Logic:** Traversal requires extra logic to avoid infinite loops, as the structure has no natural end.
- **Difficulty in Finding the End:** Identifying the logical ”end” of the list can be challenging since there is no `NULL` pointer to mark it.
- **Insertion and Deletion Overheads:** These operations are more complicated, especially at the beginning or end, as the pointers need careful adjustment to maintain circularity.

Circular Linked Lists - Implementation

- Node Structure remains the same
- Create_Node : Instead of initializing Temp->link to NULL, Make it point to itself.
- Insert and Delete operations on intermediate nodes remain the same.
- Care to be taking while performing operations with nodes at the ends of the Lists
  - Insert at Front / Delete at Front
  - Insert at End / Delete at End

Changes in create_node

```
    NODE create_node(int data)
{
    NODE Temp = (NODE)malloc(sizeof(NODE));
    if(Temp!=NULL)
    {
        Temp->data = data;
        Temp->link = Temp;
        printf("\n\t\t Node with %d created !!",
            data);
    }
    else
        printf("\n\t\t Node not created !!");
    return Temp;
}
```

- **Algorithm go_last(HEADER):**
  1. Initialize Temp $\leftarrow$ HEADER
  2. **while** (Temp $\rightarrow$ link) $\neq$ HEADER **do**
     2.1 Set Temp $\leftarrow$ (Temp $\rightarrow$ link)
  3. **end while**
  4. **return** Temp

  End Algorithm

go_last - Function to return last node reference

```
NODE go_last(NODE HEADER)
{
    NODE Temp = HEADER;
    for(;Temp->link!=HEADER;Temp=Temp->link);
    return Temp;
}
```

- **Algorithm insert_front(HEADER, data):**
  1. Create a new node: new_node $\leftarrow$ create_node(data)
  2. **if** new_node $=$ NULL **then**
     2.1 **return** HEADER
  3. **if** HEADER $=$ NULL **then**
     3.1 **return** new_node
  4. Set (go_last(HEADER) $\rightarrow$ link) $\leftarrow$ new_node
  5. Set (new_node $\rightarrow$ link) $\leftarrow$ HEADER
  6. Update HEADER $\leftarrow$ new_node
  7. **return** HEADER

  End Algorithm

Insert a node at Front

```c
 NODE insert_front(NODE HEADER, int data)
{
    NODE new_node = create_node(data);
    if(new_node==NULL)
        return HEADER;
    if(HEADER==NULL)
        return new_node;

    (go_last(HEADER))->link=new_node;
    new_node->link=HEADER;
    HEADER=new_node;

    return HEADER;
}
```

- **Algorithm delete_front(HEADER):**
  1. Initialize Temp $\leftarrow$ HEADER
  2. **if** HEADER $=$ NULL **then**
     - 2.1 Print “Empty List !!”
  3. **else**
     - 3.1 Set (go_last(HEADER) $\rightarrow$ link) $\leftarrow$ (HEADER $\rightarrow$ link)
     - 3.2 Update HEADER $\leftarrow$ (HEADER $\rightarrow$ link)
     - 3.3 Print (Temp $\rightarrow$ data)
     - 3.4 Free Temp
  4. **return** HEADER

  End Algorithm

Delete the First Node

```
    NODE delete_front(NODE HEADER)
{
    NODE Temp = HEADER;
    if(HEADER==NULL)
        printf("\n\t\t Empty List !!");
    else
    {
        go_last(HEADER)->link=HEADER->link;
        HEADER = HEADER->link;
        printf("\n\t\t Delete %d",Temp->data);
        free(Temp);
    }
    return HEADER;
}
```

- **Algorithm insert_end(HEADER, data):**
  1. Create a new node: new_node $\leftarrow$ create_node(data)
  2. **if** new_node $=$ NULL **then**
     2.1 **return** HEADER
  3. **if** HEADER $=$ NULL **then**
     3.1 **return** new_node
  4. Set (go_last(HEADER) $\rightarrow$ link) $\leftarrow$ new_node
  5. Set (new_node $\rightarrow$ link) $\leftarrow$ HEADER
  6. **return** HEADER

End Algorithm

Insert a node at the end

```
    NODE insert_end(NODE HEADER,int data)
{
    NODE new_node = create_node(data);
    if(new_node==NULL)
        return HEADER;
    if(HEADER==NULL)
        return new_node;

    go_last(HEADER)->link=new_node;
    new_node->link = HEADER;

    return HEADER;
}
```

- **Algorithm delete_end(HEADER):**
  1. Initialize Temp $\leftarrow$ HEADER
  2. **if** HEADER $=$ NULL **then**
     - 2.1 Print "Empty List !!"
  3. **else**
     - 3.1 **while** (Temp $\rightarrow$ link) $\rightarrow$ links $\neq$ HEADER **do**
     - 3.1.1 Set Temp $\leftarrow$ (Temp $\rightarrow$ link)
     - 3.2 **end while**
     - 3.3 Print (Temp $\rightarrow$ link) $\rightarrow$ data
     - 3.4 Free (Temp $\rightarrow$ link)
     - 3.5 Set (Temp $\rightarrow$ link) $\leftarrow$ HEADER
     - 3.6 Call **display(HEADER)**
  4. **return** HEADER

  End Algorithm

Delete a node at the end

```
NODE delete_end(NODE HEADER)
{
    NODE Temp = HEADER;
    if(HEADER==NULL)
        printf("\n\t\t Empty List !!");
    else
    {
        for(;Temp->link->link!=HEADER; Temp=Temp
            ->link);
        printf("\n\t\t Delete %d",Temp->link->
            data);
        free(Temp->link);
        Temp->link=HEADER;
        display(HEADER);
    }
    return HEADER;
}
```

- **Algorithm display(HEADER):**
  1. Initialize Temp $\leftarrow$ HEADER
  2. **if** HEADER $=$ NULL **then**
     - 2.1 Print "Empty List"
     - 2.2 **return**
  3. **while** (Temp $\rightarrow$ link) $\neq$ HEADER **do**
     - 3.1 Print (Temp $\rightarrow$ data)
     - 3.2 Set Temp $\leftarrow$ (Temp $\rightarrow$ link)
  4. **end while**
  5. Print (Temp $\rightarrow$ data)

  End Algorithm

Display the contents of the List

```
void display(NODE HEADER)
{
    NODE Temp = HEADER;
    if(HEADER==NULL)
    {
        printf("\n\t\t Empty List");
        return;
    }
    printf("\nH-> ");
    for(;Temp->link!=HEADER; Temp=Temp->link)
        printf("%d ->",Temp->data);
    printf("%d -> %d\n",Temp->data,Temp->link->
       data);
}
```

Thank You

Dilip Kumar Maripuri
Associate Professor
Department of Computer Applications
dilip.maripuri@pes.edu
8073212026