MANGALAYATAN UNIVERSITY
Learn Today to Lead Tomorrow
NAAC GRADE A+ Accredited University

BACHELOR OF COMPUTER APPLICATION (BCA)

CSB-1112 (Problem Solving Using C)

Block 4: Data Structures and Basic Algorithms in C

Unit 14: Implementing Data Structures

14.0 Introduction

Data structures are fundamental concepts in computer science and play a crucial role in organizing and managing data efficiently. They provide a means to store and arrange data in a way that enables efficient access and modification. Understanding different data structures and their implementations is essential for solving complex computational problems and optimizing algorithms. This chapter delves into various data structures, including linked lists, stacks, and queues, providing both theoretical knowledge and practical implementations.

Linked lists are a type of data structure where elements, known as nodes, are linked using pointers. Unlike arrays, linked lists allow for dynamic memory allocation, making them suitable for scenarios where the size of the data set is not known in advance. This chapter explores three types of linked lists: singly linked lists, doubly linked lists, and circularly linked lists. Each type has unique characteristics and use cases, which will be discussed in detail, including algorithms for insertion, deletion, and traversal operations.

In addition to linked lists, this chapter covers stacks and queues, which are linear data structures with specific operational rules. A stack follows the Last In, First Out (LIFO) principle, making it ideal for applications such as expression evaluation and function call management. On the other hand, a queue operates on the First In, First Out (FIFO) principle, suitable for scheduling and buffering tasks. The circular queue, an extension of the basic queue, addresses the limitations of a fixed-size queue by treating the queue as a circular buffer. Each data structure is examined through both theoretical perspectives and

practical implementations, providing a comprehensive understanding of their importance and applications in computer science.

14.1 Objectives

By the end of this course, students will be able to:

- Understanding of various data structures, including linked lists, stacks, and queues, and their importance in computer science.
- Implement different types of data structures in C, including singly linked lists, doubly linked lists, circularly linked lists, stacks, and queues.
- Learn to analyze the performance of different data structures in terms of time and space complexity, and understand their suitability for various applications.
- Knowledge of data structures to solve real-world computational problems efficiently, making informed decisions on which data structure to use.

14.2 Data Strictures and its Implementation

Implementing data structures involves writing code to define and manipulate various types of data structures such as arrays, linked lists, stacks, queues, trees, graphs, and hash tables. Here's a brief overview of how you can implement some common data structures in C:

- ❖ Linked Lists
- ❖ Stacks
- ❖ Queues
- ❖ Circular Queues

14.3 Linked Lists

Definition:

- A linked list is a linear data structure consisting of a sequence of elements called nodes.
- Each node contains a data element and a reference (pointer) to the next node in the sequence.

The linked allocation has the following draw backs:

1. No direct access to a particular element.
2. Additional memory required for pointers.

Types of Linked Lists:

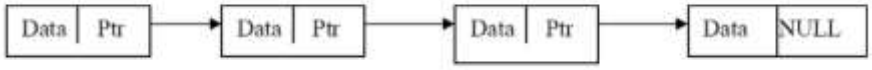
14.3.1 Singly Linked List:

A singly linked list, or simply a linked list, is a linear collection of data items. The linear order is given by means of POINTERS. These types of lists are often referred to as linear linked list.

- Each item in the list is called a node.
- Each node of the list has two fields:

1. Information- contains the item being stored in the list.
2. Next address- contains the address of the next item in the list.

* The last node in the list contains NULL pointer to indicate that it is the end of the list.

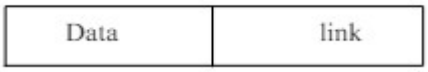


Operations on Singly linked list:

- **Insertion of a node**
- **Deletions of a node**
- **Traversing the list**

Method -1:

```
struct node
{
        int data;
        struct node *link;
};
```



Method -2:

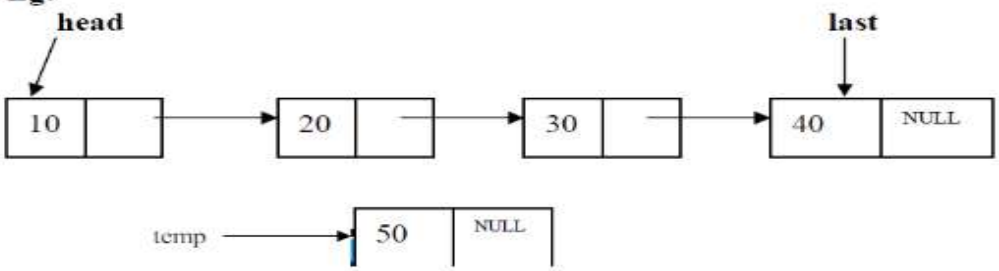
```
class node
{
public:
        int data;
        node *link;
};
```

Insertions: To place an elements in the list

- At the beginning
- End of the list
- At a given position

case 1:Insert at the beginning

Eg:

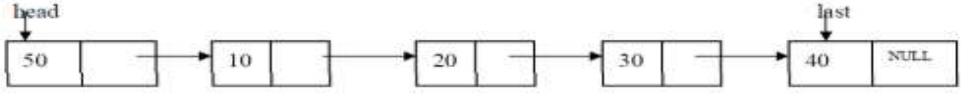


head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
temp->link=head;
head=temp;
```

After insertion:

Insert at the beginning of list.



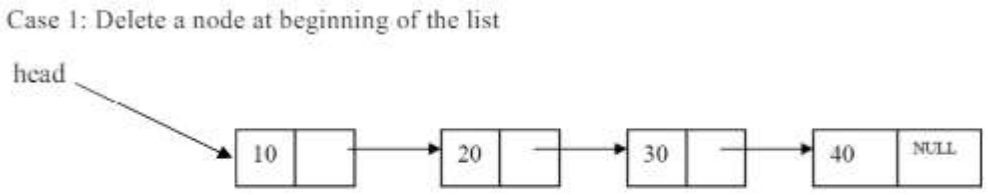
Code for insert front:-

```
template <class T>
void list<T>::insert_front()
{
struct node <T>*t,*temp;
cout<<"Enter data into node:";
cin>>item;
temp=create_node(item);
if(head==NULL)
head=temp;
else
{
temp->link=head;
head=temp;
}
}
```

Deletions: Removing an element from the list, without destroying the integrity of the list itself.

- Delete a node at beginning of the list
- Delete a node at end of the list
- Delete a node at a given position

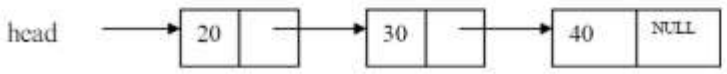
Delete a node at beginning of the list



head is the pointer variable which contains address of the first node

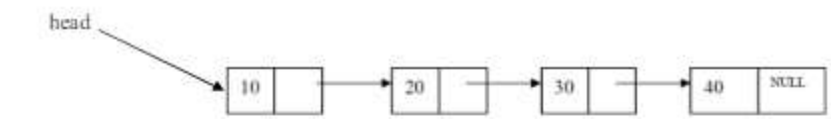
sample code is

```
t=head;
head=head->link;
cout<<"node "<<t->data<<" Deletion is sucess";
delete(t);
```



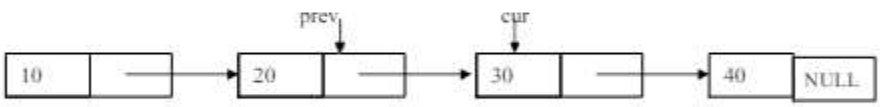
Code for Deleting a Node at Front

```
template <class T>
void list<T>::delete_front()
{
struct node<T>*t;
if(head==NULL)
cout<<"List is Empty\n";
else
{ t=head;
head=head->link;
cout<<"node "<<t->data<<" Deletion is sucess";
delete(t);
}
}
```



Delete node at position 3
head is the pointer variable which contains address of the first node. Node to be deleted is node containing value 30.
Finding node at position 3

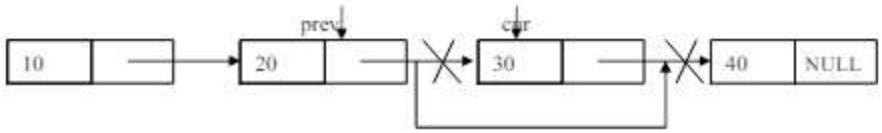
```
c=1;
while(c<pos)
{       c++;
        prev=cur;
        cur=cur->link;
}
```



cur is the node to be deleted . before deleting update links

code to update links

```
prev->link=cur->link;
cout<<cur->data <<"is deleted successfully";
delete cur;
```



Traversing the list: Assuming we are given the pointer to the head of the list, how do we get the end of the list.

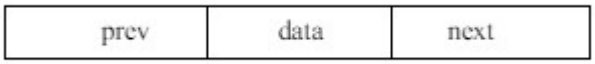
```
template <class T>
void list<T>:: display()
{
struct node<T>*t;
if(head==NULL)
{
cout<<"List is Empty\n";
}
else
{ t=head;
while(t!=NULL)
{ cout<<t->data<<"->";
t=t->link;
}
}
}
```

14.3.2 Doubly Linked List:

A singly linked list has the disadvantage that we can only traverse it in one direction. Many applications require searching backwards and forwards through sections of a list. A useful refinement that can be made to the singly linked list is to create a doubly linked list. The distinction made between the two list types is that while singly linked list have pointers going in one direction, doubly linked list have pointer both to the next and to the previous element in the list. The main advantage of a doubly linked list is that, they permit traversing or searching of the list in both directions.

In this linked list each node contains three fields.

- One to store data
- Remaining are self referential pointers which points to previous and next nodes in the list Algorithm:



Implementation of node using structure

Method -1:

```
struct node
{
int data;
struct node *prev;
struct node * next;
```

};

Implementation of node using class

Method -2:

```
class node
{
public:
int data;
node *prev;
node * next;
};
```

Operations on Doubly linked list:

- Insertion of a node
- Deletions of a node
- Traversing the list

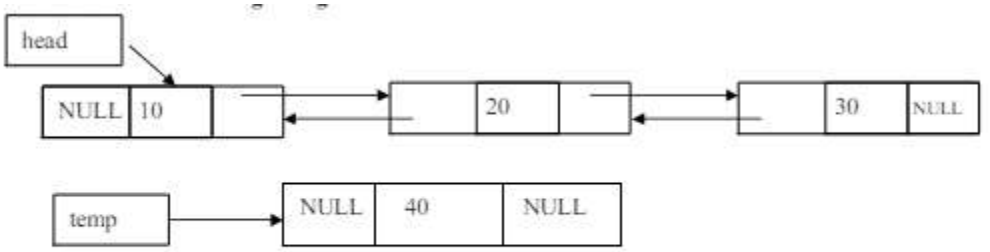
Doubly linked list ADT:

```
template <class T>
class dlist
{
public:
int data;
struct dnode<T>*head;
dlist()
{
head=NULL;
}
void display();
struct dnode<T>*create_dnode(int n);
void insert_end();
void insert_front();
void delete_end();
void delete_front();
void dnode_count();
```

```
void Insert_at_pos(int pos);
void Delete_at_pos(int pos);
};
```

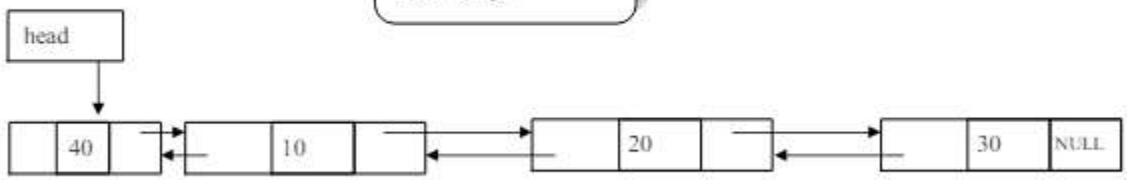
Insertions: To place an elements in the list there are 3 cases

Insert at the beginning



head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
temp->next=head;
head->prev=temp;
head=temp;
```



Code for insert front:-

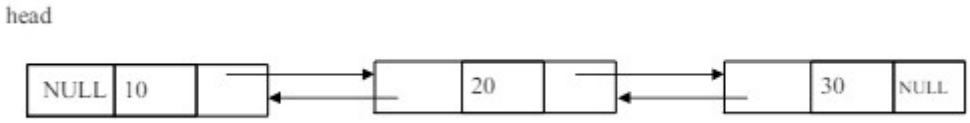
```
template <class T>
void DLL<T>::insert_front()
{
struct dnode <T>*t,*temp;
cout<<"Enter data into node:";
cin>>data;
temp=create_dnode(data);
if(head==NULL)
head=temp;
else
{ temp->next=head; head-
>prev=temp;
head=temp;
}
}
```

Deletions: Removing an element from the list, without destroying the integrity of the list itself.

To place an element from the list there are 3 cases:

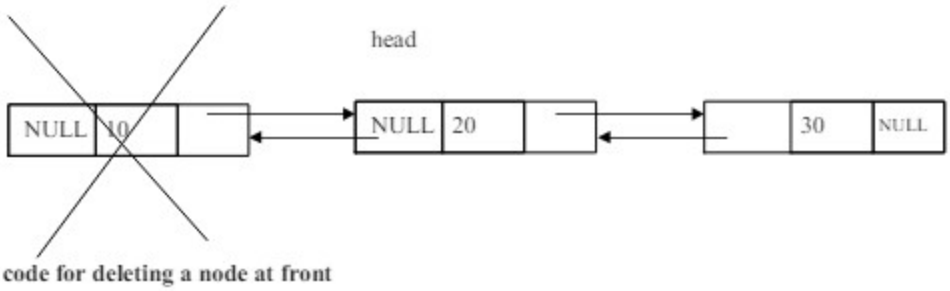
- Delete a node at beginning of the list
- Delete a node at end of the list
- Delete a node at a given position

Delete a node at beginning of the list



head is the pointer variable which contains address of the first node sample code is

```
t=head;
head=head->next;
head->prev=NULL;
cout<<"dnode "<<t->data<<" Deletion is sucess";
delete(t);
```



code for deleting a node at front

Code for Deleting a Node at Front

```
template <class T>
void dlist<T>:: delete_front()
{struct dnode<T>*t;
if(head==NULL)
cout<<"List is Empty\n";
else
{ t=head;
head=head->next;
head->prev=NULL;
cout<<"dnode "<<t->data<<" Deletion is sucess";
```

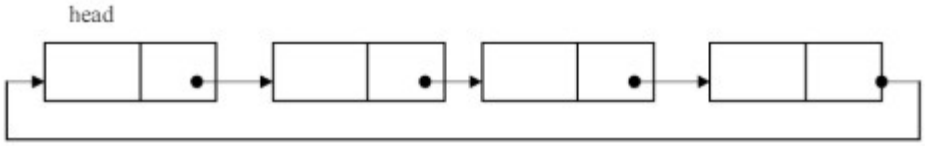
```
delete(t);
}
}
```

14.3.3 Circularly Linked List

A circularly linked list, or simply circular list, is a linked list in which the last node is always points to the first node. This type of list can be build just by replacing the NULL pointer at the end of the list with a pointer which points to the first node. There is no first or last node in the circular list.

Advantages:

- Any node can be traversed starting from any other node in the list.
- There is no need of NULL pointer to signal the end of the list and hence, all pointers contain valid addresses.
- In contrast to singly linked list, deletion operation in circular list is simplified as the search for the previous node of an element to be deleted can be started from that i



InsertAtBeginning(headRef, data)

Input: Pointer to the head of the circular linked list (headRef), data to be inserted (data)

Output: None

1. Create a new node (newNode) with the given data.
2. If the list is empty (headRef is NULL):

   Set the next pointer of newNode to point to itself.
3. Otherwise:

   Set the next pointer of newNode to point to the current first node (headRef->next).
4. Update the head pointer (headRef) to point to newNode.

14.4 Stack

A Stack is a linear data structure where insertion and deletion of items takes place at one end called top of the stack. A Stack is defined as a data structure which operates on a last-in first-out basis. So it is also is referred as Last-in First-out(LIFO).

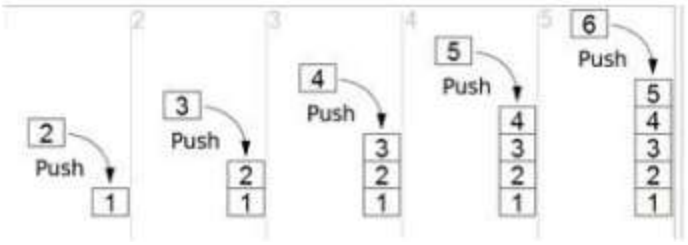
Stack uses a single index or pointer to keep track of the information in the stack. The basic operations associated with the stack are:

- Push(insert) an item onto the stack.
- Pop(remove) an item from the stack.

The general terminology associated with the stack is as follows:

A stack pointer keeps track of the current position on the stack. When an element is placed on the stack, it is said to be pushed on the stack. When an object is removed from the stack, it is said to be popped off the stack. Two additional terms almost always used with stacks are overflow, which occurs when we try to push more information on a stack that it can hold, and underflow, which occurs when we try to pop an item off a stack which is empty.

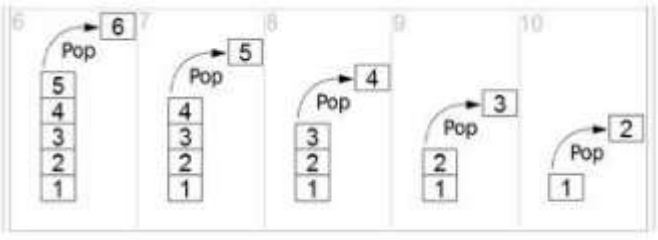
Pushing items onto the stack:




```
//code to push an element on to stack;
template<class T>
void stack<T>::push()
{
if(top==max-1)
cout<<"Stack Overflow...\n";
else
{
 cout<<"Enter an element to be pushed:";
top++;
cin>>data;
stk[top]=data;
cout<<"Pushed Sucesfully....\n";
}
}
```

Popping an element from stack:

To remove an item, first extract the data from top position in the stack and then decrement the stack pointer, top.




```
//code to remove an element from stack
template<class T>
void stack<T>::pop()
{
if(top= =-1)
```

```
cout<<"Stack is Underflow";
else
{
data=stk[top];
top--;
cout<<data<<" is poped Sucesfully ....\n";
}
}
```

Applications of Stack:

- Stacks are used in conversion of infix to postfix expression.
- Stacks are also used in evaluation of postfix expression.
- Stacks are used to implement recursive procedures.
- Stacks are used in compilers.
- Reverse String

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are−

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

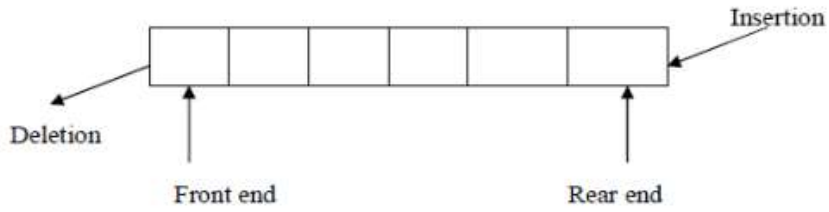
Expression	Example	Note
Infix	a + b	Operator Between Operands
Prefix	+ a b	Operator before Operands
Postfix	a b +	Operator after Operands

Conversion of Infix Expressions to Prefix and Postfix

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

14.5 Queue

A queue is an ordered collection of data such that the data is inserted at one end and deleted from another end. The key difference when compared stacks is that in a queue the information stored is processed first-in first-out or FIFO. In other words the information receive from a queue comes in the same order that it was placed on the queue.



Representing a Queue:

One of the most common way to implement a queue is using array. An easy way to do so is to define an array Queue, and two additional variables front and rear. The rules for manipulating these variables are simple:

- Each time information is added to the queue, increment rear.
- Each time information is taken from the queue, increment front.
- Whenever front >rear or front=rear=-1 the queue is empty.

Array implementation of a Queue does have drawbacks. The maximum queue size has to be set at compile time, rather than at run time. Space can be wasted, if we do not use the full capacity of the array.

Operations on Queue:

A queue has two basic operations:

- Adding new item to the queue
- Removing items from queue.

The operation of adding new item on the queue occurs only at one end of the queue called the rear or back. The operation of removing items of the queue occurs at the other end called the front. For insertion and deletion of an element from a queue, the array elements begin at 0 and the maximum elements of the array is maxSize. The variable front will hold the index of the item that is considered the front of the queue, while the rear variable will hold the index of the last item in the queue.

Assume that initially the front and rear variables are initialized to -1. Like stacks, underflow and overflow conditions are to be checked before operations in a queue.

Queue empty or underflow condition is

```
if((front>rear)||front= =-1)
    cout<"Queue is empty";
```

Queue Full or overflow condition is

```
if((rear==max)
    cout<"Queue is full";
```

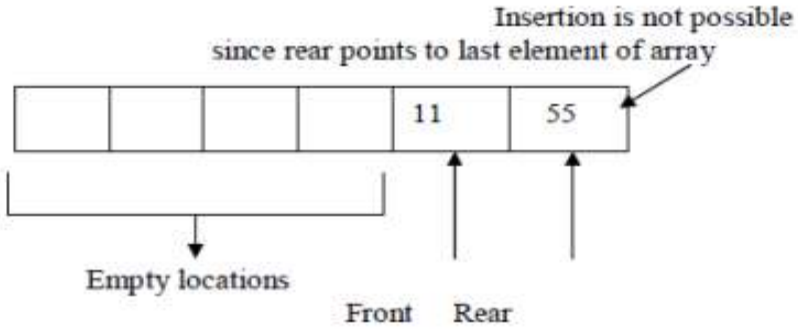
Application of Queue:

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

14.6 CIRCULAR QUEUE

Once the queue gets filled up, no more elements can be added to it even if any element is removed from it consequently. This is because during deletion, rear pointer is not adjusted.



When the queue contains very few items and the rear pointer points to last element. i.e. rear=maxSize-1, we cannot insert any more items into queue because the overflow condition satisfies. That means a lot of space is wasted. Frequent reshuffling of elements is time consuming. One solution to this is arranging all elements in a circular fashion. Such structures are often referred to as Circular Queues.

A circular queue is a queue in which all locations are treated as circular such that the first location CQ[0] follows the last location CQ[max-1]. Circular Queue empty or underflow condition is

Circular Queue empty or underflow condition is

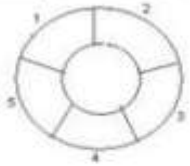
```
if(front==-1)
        cout<<"Queue is empty";
```

Circular Queue Full or overflow condition is

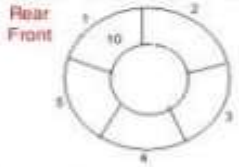
```
if(front==(rear+1)%max)
{
        cout<<"Circular Queue is full\n";
}
```

Example: Consider the following circular queue with N = 5.

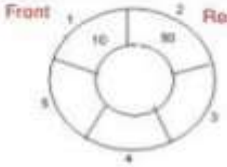
1. Initially, Rear = 0, Front = 0.



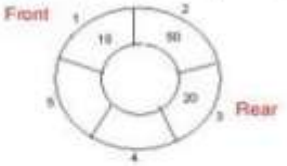
2. Insert 10, Rear = 1, Front = 1.



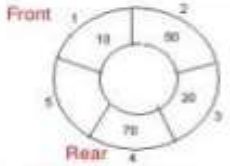
3. Insert 50, Rear = 2, Front = 1.



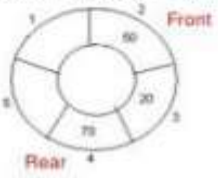
4. Insert 20, Rear = 3, Front = 0.



5. Insert 70, Rear = 4, Front = 1.



6. Delete front, Rear = 4, Front = 2.



Insertion into a Circular Queue:

Algorithm CQueueInsertion(Q,maxSize,Front,Rear,item)

Step 1: If Rear = maxSize-1 then

Rear = 0

else

Rear=Rear+1

Step 2: If Front = Rear then

print “Queue Overflow”

Return

Step 3: Q[Rear] = item

Step 4: If Front = 0 then

Front = 1

Step 5: Return

Deletion from Circular Queue:

Algorithm CQueueDeletion(Q,maxSize,Front,Rear,item)

Step 1: If Front = 0 then

print “Queue Underflow”

Return

Step 2: K=Q[Front]

Step 3: If Front = Rear then

begin

Front = -1

Rear = -1

end

else

If Front = maxSize-1 then

Front = 0

else

Front = Front + 1

Step 4: Return K

14.7 Summary

In this chapter, we delved into various fundamental data structures that are essential for efficient data management and problem-solving in computer science. We started by exploring linked lists, including singly linked lists, doubly linked lists, and circularly linked lists. Each type of linked list offers unique advantages and use cases, from the simplicity of singly linked lists to the bidirectional traversal capabilities of doubly linked lists and the continuous loop structure of circularly linked lists. Detailed algorithms for common operations such as insertion, deletion, and traversal were provided, along with practical implementations in C.

Next, we examined stacks and queues, two linear data structures with distinct operational principles. Stacks, adhering to the Last In, First Out (LIFO) method, are particularly useful in scenarios such as expression evaluation, backtracking algorithms, and function call management. We implemented basic stack operations, including push, pop, and peek, highlighting their applications and importance. Queues, following the First In, First Out (FIFO) principle, are crucial for scheduling processes, managing tasks in buffers, and handling requests in sequential order. The circular queue, an extension of the basic queue, was also covered, addressing the limitations of fixed-size queues by implementing a circular buffer approach.

Throughout the chapter, theoretical concepts were reinforced with practical code examples in C, ensuring a deep understanding of each data structure's implementation and usage. By mastering these data structures, students are equipped with the knowledge and skills necessary to tackle a wide range of computational problems efficiently, making informed decisions about the most appropriate data structures to use in different scenarios. This foundational knowledge paves the way for further exploration and application of advanced data structures and algorithms in computer science.

14.8 Questions

1. **What is a singly linked list and how does it differ from a doubly linked list?**

Ans.1 A singly linked list is a type of linked list where each node contains a data element and a reference (or pointer) to the next node in the sequence. The last node in a singly linked list points to NULL, indicating the end of the list. A doubly linked list, on the other hand, has nodes that contain references to both the next and the previous nodes. This allows traversal in both directions (forward and backward) in the list, whereas singly linked lists can only be traversed in one direction (forward).

2. **Explain how insertion at the beginning of a singly linked list is performed.**

Ans.2 To insert a node at the beginning of a singly linked list, follow these steps:

- Create a new node and assign the data to it.
- Set the new node's next pointer to the current head of the list.
- Update the head pointer to point to the new node. This operation has a constant time complexity of O(1) because it involves updating only a few pointers.

3. Describe the difference between a queue and a circular queue.

Ans.3 A queue follows the First In, First Out (FIFO) principle, where elements are added at the rear and removed from the front. A circular queue is an extension of the basic queue where the last position is connected back to the first position to form a circle. This allows for efficient use of space by reusing vacated positions, avoiding the need for shifting elements and handling overflow scenarios more gracefully.

4. How do you implement the enqueue operation in a circular queue using an array?

Ans.4 To implement the enqueue operation in a circular queue using an array, follow these steps:

1. Check if the queue is full by checking if the next position of the rear index is equal to the front index.
2. If the queue is not full, add the new element at the rear index.
3. Update the rear index to the next position in a circular manner.

Example in C:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
struct CircularQueue {
    int items[MAX_SIZE];
    int front;
    int rear;
};
void initializeQueue(struct CircularQueue* queue) {
    queue->front = -1;
    queue->rear = -1;
}
int isFull(struct CircularQueue* queue) {
    return (queue->rear + 1) % MAX_SIZE == queue->front;
}
int isEmpty(struct CircularQueue* queue) {
    return queue->front == -1;
}
void enqueue(struct CircularQueue* queue, int value) {
    if (isFull(queue
```

5. Provide a use case where a stack would be preferred over a queue.

Ans.5 A stack would be preferred over a queue in situations where the most recently added elements need to be accessed first. One common use case is in the implementation of the undo feature in text editors. Each action performed by the user is pushed onto the stack. When the user wishes to undo

an action, the most recent action is popped from the stack and reversed, adhering to the LIFO principle.

6. **Describe an application where a circular queue is particularly useful.**

Ans.6 A circular queue is particularly useful in scenarios involving fixed-size buffers that operate in a continuous loop. One common application is in network buffering, where data packets are received and processed in a cyclical manner. The circular queue ensures that the buffer space is utilized efficiently and prevents overflow by wrapping around to the beginning of the array when the end is reached, allowing for smooth data handling in high-throughput systems.

14.9 References

- Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.
- King, K. N. (2008). C Programming: A Modern Approach (2nd ed.). W. W. Norton & Company.
- Prata, S. (2014). C Primer Plus (6th ed.). Addison-Wesley Professional.
- Deitel, P., & Deitel, H. (2016). C How to Program (8th ed.). Pearson.

***