UNIT II

UNIT II: ARRAYS AND LIST

Array implementation of List, Traversing, Insertion, Deletion, Application of List, Polynomial Arithmetic - Linked list, Implementation, Insertion, Deletion and Search, Sparse Matrix, Circular Linked List, Applications, Josephus Problem, Double linked list, Cursor based implementation

INTRODUCTION

Array

- An Array is a data structure which can store a fixed-size sequential collection of elements of the same type.

- An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

- A specific element in an array is accessed by an index.

- All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

What is List???

A number of connected items or names written or printed consecutively, typically one below the other.

List plays an important role in our daily life.
List Operations

1. Add

We can add new item/row in our list at first or at last or at any position.

2. Edit

If we entered wrong value for an item/row, don't worry we can edit/enter the right value at any point of time.

3. Search

If we want to search for a value in a big List, We can use CTRL+F to search and we can find the value which we need or position of row/item which contains the value which we need.

What is LIST ADT?

A List is a dynamic, ordered tuple of homogeneous elements.

- List will be in the form: A1, A2, A3, .... An
- First Element of the List: A1
- Last Element of the List: An
- ith Element of the List: Ai ,
- Position of Element Ai : i ,
- Position ranges from 0 to N
- Size of the List: N
- Empty List: If Size of the list is Zero (i.e N=0), it is Empty List.
- Precedes and Succeeds: For any list except empty list, we say that Ai+1  follows (or succeeds) Ai (i<N) and that Ai-1  precedes Ai (i>1)

LIST Operations:

L==> List
x==>Element
p==>Position
1. **Insert** \((x,p,L)\)
   Insert \(x\) at position \(p\) in list \(L\)
   If \(p = \text{END}(L)\), insert \(x\) at the end
   If \(L\) does not have position \(p\), result is undefined

2. **Locate** \((x,L)\)
   returns position of \(x\) on \(L\)
   returns \(\text{END}(L)\) if \(x\) does not appear

3. **Retrieve** \((p,L)\)
   returns element at position \(p\) on \(L\)
   undefined if \(p\) does not exist or \(p = \text{END}(L)\)

4. **Delete** \((p,L)\)
   delete element at position \(p\) in \(L\)
   undefined if \(p = \text{END}(L)\) or does not exist

5. **Next** \((p,L)\)
   returns the position immediately following position \(p\)

6. **Prev** \((p,L)\)
   returns the position previous to \(p\)

7. **Makenull** \((L)\)
   causes \(L\) to become an empty list and returns position \(\text{END}(L)\)

8. **First** \((L)\)
   returns the first position on \(L\)

9. **Printlist** \((L)\)
   print the elements of \(L\) in order of occurrence
Array Based Implementation of LIST

List Structure:

Operations:

1. Is Empty(LIST)
2. Is Full(LIST)
3. Insert Element to End of the LIST.
4. Delete Element from End of the LIST.
5. Insert Element to front of the LIST.
6. Delete Element from front of the LIST.
7. Insert Element to nth position of LIST.
8. Delete Element from nth Position
9. Search Element in the LIST.
10. Print the Elements in the LIST.

Fresh LIST:
1. Is Empty(LIST)

If (Current Size==0) "LIST is Empty"

else "LIST is not Empty"

2. Is Full(LIST)

If (Current Size=Max Size) "LIST is FULL" else "LIST is not FULL"

3. Insert Element to End of the LIST.

1. Check that weather the List is full or not
   1. If List is full return error message "List is full. Can't Insert".
   2. If List is not full.
      1. Get the position to insert the new element by Position=Current Size+1
      2. Insert the element to the Position
      3. Increase the Current Size by 1 i.e. Current Size=Current Size+1
4. Delete Element from End of the LIST.

1. Check that weather the List is empty or not

   1. If List is empty return error message "List is Empty. Can't Delete".

   2. If List is not Empty.

       1. Get the position of the element to delete by Position=Current Size

       2. Delete the element from the Position

       3. Decrease the Current Size by 1 i.e. Current Size=Current Size-1
5. Insert Element to front of the LIST.

1. Check that weather the List is full or not

   1. If List is full return error message “List is full. Can't Insert”.

   2. If List is not full.

      1. Free the 1st Position of the list by moving all the Element to one position forward i.e. New Position=Current Position + 1.

      2. Insert the element to the 1st Position

      3. Increase the Current Size by 1 i.e. Current Size=Current Size+1
6. Delete Element from front of the LIST.

1. Check that weather the List is empty or not

   1. If List is empty return error message "**List is Empty. Can't Delete**".

   2. If List is not Empty.

      1. Move all the elements except one in the 1st position to one position backward i.e **New Position= Current Position -1**

      2. After the 1st step, element in the **1st position will be automatically deleted**.

      3. Decrease the Current Size by 1 i.e. **Current Size=Current Size-1**
7. Insert Element to nth Position of the LIST.

1. Check that whether the List is full or not
   1. If List is full return error message "List is full. Can't Insert".
   2. If List is not full.
      1. If List is Empty, Insert element at Position 1.
      2. If (nth Position > Current Size)
         1. Return message "nth Position Not available in List"
      3. else
         1. Free the nth Position of the list by moving all Elements to one position forward except n-1,n-2,... 1 Position i.e move only from n to current size position Elements. i.e New Position=Current Position + 1.
         2. Insert the element to the nth Position
         3. Increase the Current Size by 1 i.e. Current Size=Current Size+1
8. Delete Element from nth Position of the LIST.

1. Check that whether the List is Empty or not

   1. If List is Empty return error message "List is Empty."

   2. If List is not Empty.

      1. If (nth Position > Current Size)

         1. Return message "nth Position Not available in List"

      2. If (nth Position == Current Size)

         1. Delete the element from nth Position

         2. Decrease the Current Size by 1 i.e. Current Size=Current Size-1

   3. If (nth Position < Current Size)

      1. Move all the Elements to one position backward except n,n-1,n-2,... 1 Position i.e move only from n+1 to current size position Elements, i.e New Position=Current Position - 1.

      2. After the previous step, nth element will be deleted automatically.

      3. Decrease the Current Size by 1 i.e. Current Size=Current Size-1

---

### Table: Element to nth Position of the List

<table>
<thead>
<tr>
<th>Before Insert Element (100) to 3rd Position of the List.</th>
<th>After Insert Element (100) to 3rd Position of the List.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>List</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>List [0]</td>
<td>10</td>
</tr>
</tbody>
</table>

### Diagram:

- Insert Element to nth Position of the List
- Move Elements from only nth to Current Size Position i.e, 3 to 4 Position
- Increase the Current Size by 1
- Decrease the Current Size by 1 i.e. Current Size=Current Size-1

---

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Size</td>
<td>5</td>
</tr>
<tr>
<td>Current Size</td>
<td>4</td>
</tr>
</tbody>
</table>
Delete Element From $n^{th}$ Position of the List

$n^{th}$ Position = Current Size

**Before Delete Element (50) from 5$^{th}$ Position of the List.**

<table>
<thead>
<tr>
<th>Index</th>
<th>Array</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>List [0]</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>List [1]</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>List [2]</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>List [3]</td>
<td>40</td>
<td>4</td>
</tr>
</tbody>
</table>

1. $n^{th}$ Position = 5
   - Current Size = 5
2. $n^{th}$ Position = Current Size
   - So Delete Element from $n^{th}$ Position
   - i.e. Delete Element from 5$^{th}$ Position
3. Decrease the Current Size by 1
   - Current Size = Current Size - 1
   - Current Size = 5 - 1 = 4

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Size</td>
<td>5</td>
</tr>
<tr>
<td>Current Size</td>
<td>5</td>
</tr>
</tbody>
</table>

**After Delete Element (10) from 5$^{th}$ Position of the List.**

<table>
<thead>
<tr>
<th>Index</th>
<th>Array</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>List [0]</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>List [1]</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>List [2]</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>List [3]</td>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>List [4]</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Size</td>
<td>5</td>
</tr>
<tr>
<td>Current Size</td>
<td>4</td>
</tr>
</tbody>
</table>

Delete Element From $n^{th}$ Position of the List

$n^{th}$ Position < Current Size

**Before Delete Element (20) from 2$^{nd}$ Position of the List.**

<table>
<thead>
<tr>
<th>Index</th>
<th>Array</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>List [0]</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>List [1]</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>List [2]</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>List [3]</td>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>List [4]</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

1. $n^{th}$ Position : 2
   - Current Size : 4
2. $n^{th}$ Position < Current Size
   - So Move Elements from only $(n+1)^{th}$ to Current Size Position i.e., 3 to 4 Position Backward by one position
   - Position 3 $\rightarrow$ Position 2
   - Position 4 $\rightarrow$ Position 3
3. Element at $n^{th}$ Position is automatically deleted
   - Element at 2$^{nd}$ Position deleted
4. Decrease the Current Size by 1
   - Current Size = Current Size - 1

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Size</td>
<td>5</td>
</tr>
<tr>
<td>Current Size</td>
<td>4</td>
</tr>
</tbody>
</table>

**After Delete Element (10) from 2$^{nd}$ Position of the List.**

<table>
<thead>
<tr>
<th>Index</th>
<th>Array</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>List [0]</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>List [1]</td>
<td>30</td>
<td>2</td>
</tr>
<tr>
<td>List [2]</td>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>List [3]</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>List [4]</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Size</td>
<td>5</td>
</tr>
<tr>
<td>Current Size</td>
<td>3</td>
</tr>
</tbody>
</table>
9. Search Element in the LIST.

1. Check that whether the list is empty or not.
   1. If List is empty, return error message “List is Empty”.
   2. If List is not Empty
      1. Find the Position where the last element available in the List by Last Position = Current Size
      2. For( Position 1 to Last Position)
         1. If(Element @ Position== Search Element)//If Element matches the search element
         2. return the Position by message “Search Element available in Position”
      3. Else return message “Search Element not available in the List”
10. Print the Elements in the LIST.

1. Check that whether the list is empty or not.
   1. If List is empty, return error message "List is Empty".
   2. If List is not Empty
      1. Find the Position where the last element available in the List by Last Position = Current Size
      2. For (Position 1 to Last Position)
      3. Print the Position and Element available at the position of List.

Applications of Linked List

1. Applications that have an MRU list (a linked list of file names)
2. The cache in your browser that allows you to hit the BACK button (a list of URLs)
3. Undo functionality in Photoshop or Word (a linked list of state)
4. A stack, hash table, and binary tree can be implemented using a doubly linked list.
5. A great way to represent a deck of cards in a game.
6. For representing polynomials. It means in addition/subtraction/multiplication of two polynomials. Eg: \( p1 = 2x^2 + 3x + 7 \) and \( p2 = 3x^3 + 5x + 2 \); \( p1 + p2 = 3x^3 + 2x^2 + 8x + 9 \)
7. Dynamic Memory Management in allocation and releasing memory at runtime.
Advantages of Array Implementation of LIST:

- No need to declare Large number of variables Individually
- Variables are not scattered in Memory, they are stored in contiguous memory.
- Ease the handling of large number of variables of same datatype.

Disadvantages of Array Implementation of LIST:

- Rigid Structure
- Can be hard to add/remove elements.
- Cannot be dynamically re-sized in most Languages.
- Memory Loss

Polynomial Arithmetic Linked list

This experiment guides you using Linked lists in a practical example. This is useful if you want to understand linked lists or if you want to see a realistic, applied example of pointer-intensive code. In this experiment, you will essentially use linked lists to add and multiply (sparse) polynomials.

Linked lists are useful to study for two reasons. Most obviously, linked lists are a data structure which you may want to use in real programs. Seeing the strengths and weaknesses of linked lists will give you an appreciation of the some of the time, space, and code issues which are useful to thinking about any data structures in general. Somewhat less obviously, linked lists are great way to learn about pointers. In fact, you may never use a linked list in a real program, but you are certain to use lots of pointers. Linked list problems are a nice combination of algorithms and pointer manipulation. Traditionally, linked lists have been the domain where beginning programmers get the practice to really understand pointers.

Application of linked lists is to polynomials

We now use linked lists to perform operations on polynomials. Let \( f(x) = \sum_{i=0}^{d} a_i x^i \). The quantity \( d \) is called as the degree of the polynomial, with the assumption that \( a_d \neq 0 \). A polynomial of degree \( d \) may however have missing terms i.e., powers \( j \) such that \( 0 \leq j < d \) and \( a_j = 0 \). The standard operations on a polynomial are addition and multiplication. If we store the coefficient of each term of the polynomials in an array of size \( d + 1 \), then these operations can be supported in a straight forward way. However, for sparse polynomials, i.e., polynomials where there are few non-zero coefficients, this is not efficient. One possible solution is to use linked lists.
to store degree, coefficient pairs for non-zero coefficients. With this representation, it makes it easier if we keep the list of such pairs in decreasing order of degrees.

A polynomial is a sum of terms. Each term consists of a coefficient and a (common) variable raised to an exponent. We consider only integer exponents, for now.

Example: $4x^3 + 5x - 10$.

How to represent a polynomial? Issues in representation, should not waste space, should be easy to use it for operating on polynomials. Any case, we need to store the coefficient and the exponent.

```c
struct node
{
    float coefficient;
    int exponent;
    struct node *next;
}
```

Can we use a linked list?. Each node of the linked list stores the coefficient and the exponent. Should also store in the sorted order of exponents.

Let us now see how two polynomials can be added. Let $P_1$ and $P_2$ be two polynomials stored as linked lists in sorted (decreasing) order of exponents. The addition operation is defined as follows. Add terms of like-exponents. We have $P_1$ and $P_2$ arranged in a linked list in decreasing order of exponents. We can scan these and add like terms. Need to store the resulting term only if it has non-zero coefficient. The number of terms in the result polynomial $P_1+P_2$ need not be known in advance. We’ll use as much space as there are terms in $P_1+P_2$.

**Adding polynomials:**

\[
(3x^5 - 9x^3 + 4x^2) + (-8x^5 + 8x^3 + 2)
\]
\[
= 3x^5 - 8x^5 - 9x^3 + 8x^3 + 4x^2 + 2
\]
\[
= -5x^5 - x^3 + 4x^2 + 2
\]

**Multiplying polynomials:**

\[
(2x - 3)(2x^2 + 3x - 2)
\]
\[
= 2x(2x^2 + 3x - 2) - 3(2x^2 + 3x - 2)
\]
Refer: http://cstar.iiit.ac.in/~kkishore/DSVL/exp4/polynomial.swf

Linked List Basics

A linked-list is a sequence of data structures which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list the second most used data structure after array. Following are important terms to understand the concepts of Linked List.

- **Link** – Each Link of a linked list can store a data called an element.
- **Next** – Each Link of a linked list contain a link to next link called Next.
- **LinkedList** – A LinkedList contains the connection link to the first Link called First.
Linked List Representation

As per above shown illustration, following are the important points to be considered.

- LinkedList contains an link element called first.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Last Link carries a Link as null to mark the end of the list.

Types of Linked List

Following are the various flavors of linked list.

Types of Linked Lists

- Singly Linked List: Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.

- Doubly Linked List: In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.
Circular Linked List: In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – add an element at the beginning of the list.
- **Deletion** – delete an element at the beginning of the list.
- **Display** – displaying complete list.
- **Search** – search an element using given key.
- **Delete** – delete an element using given key.

**Insertion Operation**

Insertion is a three step process –

- Create a new Link with provided data.
- Point New Link to old First Link.
- Point First Link to this New Link.
//insert link at the first location
void insertFirst(int key, int data){
    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;
    //point it to old first node
    link->next = head;
    //point first to new first node
    head = link;
}

**Deletion Operation**

Deletion is a two step process –

- Get the Link pointed by First Link as Temp Link.
- Point First Link to Temp Link's Next Link.
//delete first item
struct node* deleteFirst() {
    //save reference to first link
    struct node* tempLink = head;
    //mark next to first link as first
    head = head->next;

    //return the deleted link
    return tempLink;
}

Display Operation

- Get the Link pointed by First Link as Current Link.
- Check if Current Link is not null and display it.
- Point Current Link to Next Link of Current Link and move to above step.
//display the list
void printList(){
    struct node *ptr = head;
    printf("n[ ");
    //start from the beginning
    while(ptr != NULL){
        printf("(%d,%d )",ptr->key,ptr->data);
        ptr = ptr->next;
    }
    printf( " ]");
}

Searching for an Element in the List

In searching we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the data of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

node* LinkedList :: search(int x) {
    node *ptr = head;
    while(ptr != NULL && ptr->data != x) {
        //until we reach the end or we find a Node with data x, we keep moving
        ptr = ptr->next;
    }
    return ptr;
}
Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with \textbf{data} which we want to delete and then we delete it. In our approach, we will define a method which will take the \textbf{data} to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following:

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.

```c
node* LinkedList :: deleteNode(int x) {
    //searching the Node with data x
    node *n = search(x);
    node *ptr = head;
    if(ptr == n) {
        ptr->next = n->next;
        return n;
    } else {
        while(ptr->next != n) {
            ptr = ptr->next;
        }
        ptr->next = n->next;
        return n;
    }
}
```
Sparse matrix

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a mXn matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

Sparse matrix is a matrix which contains very few non-zero elements.

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation
2. Linked Representation

Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...
In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

**Linked Representation**

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely **header node** and **element node**. Header node consists of three fields and element node consists of five fields as shown in the image...

Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image...
In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to it's respective header node.
Doubly Linked List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways either forward and backward easily as compared to Single Linked List. Following are important terms to understand the concepts of doubly Linked List

- **Link** – Each Link of a linked list can store a data called an element.
- **Next** – Each Link of a linked list contain a link to next link called Next.
- **Prev** – Each Link of a linked list contain a link to previous link called Prev.
- **LinkedList** – A LinkedList contains the connection link to the first Link called First and to the last link called Last.

Doubly Linked List Representation

As per above shown illustration, following are the important points to be considered.

- Doubly LinkedList contains an link element called first and last.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Each Link is linked with its previous link using its prev link.
- Last Link carries a Link as null to mark the end of the list.

Basic Operations

Following are the basic operations supported by an list.

- **Insertion** – add an element at the beginning of the list.
- **Deletion** – delete an element at the beginning of the list.
- **Insert Last** – add an element in the end of the list.
- **Delete Last** – delete an element from the end of the list.
• **Insert After** – add an element after an item of the list.
• **Delete** – delete an element from the list using key.
• **Display forward** – displaying complete list in forward manner.
• **Display backward** – displaying complete list in backward manner.

**Insertion Operation**

Following code demonstrate insertion operation at beginning in a doubly linked list.

```c
//insert link at the first location
void insertFirst(int key, int data) {

    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;

    if(isEmpty()) {
        //make it the last link
        last = link;
    }
    else {
        //update first prev link
        head->prev = link;
    }

    //point it to old first link
    link->next = head;

    //point first to new first link
    head = link;
}
```

**Deletion Operation**
Following code demonstrate deletion operation at beginning in a doubly linked list.

```c
//delete first item
struct node* deleteFirst() {

    //save reference to first link
    struct node *tempLink = head;

    //if only one link
    if(head->next == NULL) {
        last = NULL;
    } else {
        head->next->prev = NULL;
    }

    head = head->next;

    //return the deleted link
    return tempLink;
}
```

**Insertion at End Operation**

Following code demonstrate insertion operation at last position in a doubly linked list.

```c
//insert link at the last location
void insertLast(int key, int data) {

    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));
    link->key = key;
    link->data = data;

    if(isEmpty()) {
        //make it the last link
```
last = link;

} else {
    // make link a new last link
    last->next = link;
    // mark old last node as prev of new link
    link->prev = last;
}

// point last to new last node
last = link;

Circular Linked List

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.
Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Implementing Circular Linked List

Implementing a circular linked list is very easy and almost similar to linear linked list implementation, with the only difference being that, in circular linked list the last Node will have it's next point to the Head of the List. In Linear linked list the last Node simply holds NULL in it's next pointer.

So this will be oue Node class, as we have already studied in the lesson, it will be used to form the List.

```cpp
class Node {
    public:
        int data;
        //pointer to the next node
```
Circular Linked List

Circular Linked List class will be almost same as the Linked List class that we studied in the previous lesson, with a few difference in the implementation of class methods.

class CircularLinkedList {
  public:
    node *head;

    //declaring the functions

    //function to add Node at front
    int addAtFront(node *n);

    //function to check whether Linked list is empty
    int isEmpty();
}
//function to add Node at the End of list
int addAtEnd(node *n);

//function to search a value
node* search(int k);

//function to delete any Node
node* deleteNode(int x);

CircularLinkedList() {
    head = NULL;
}

---

**Insertion at the Beginning**

Steps to insert a Node at beginning:

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL (in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

int CircularLinkedList :: addAtFront(node *n) {
    int i = 0;
Insertion at the End

Steps to insert a Node at the end:

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, and make the next of the Newly added Node point to the Head of the List.

```c
/* If the list is empty */
if(head == NULL) {
    n->next = head;
    //making the new Node as Head
    head = n;
    i++;
}
else {
    n->next = head;
    //get the Last Node and make its next point to new Node
    Node* last = getLastNode();
    last->next = n;
    //also make the head point to the new first Node
    head = n;
    i++;
}
//returning the position where Node is added
return i;
}
```

```
int CircularLinkedList :: addAtEnd(node *n) {
```
// If list is empty
if(head == NULL) {
    // making the new Node as Head
    head = n;
    // making the next pointer of the new Node as Null
    n->next = NULL;
}
else {
    // getting the last node
    node *last = getLastNode();
    last->next = n;
    // making the next pointer of new node point to head
    n->next = head;
}

*Searching for an Element in the List*

In searching we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the data of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

node* CircularLinkedList:: search(int x) {
    node *ptr = head;
    while(ptr != NULL && ptr->data != x) {
        // until we reach the end or we find a Node with data x, we keep moving
        ptr = ptr->next;
    }
    return ptr;
Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with data which we want to delete and then we delete it. In our approach, we will define a method which will take the data to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following:

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted. And update the next pointer of the Last Node as well.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.
- If the Node is at the end, then remove it and make the new last node point to the head.

```c++
node* CircularLinkedList :: deleteNode(int x) {
    //searching the Node with data x
    node *n = search(x);
    node *ptr = head;
    if(ptr == NULL) {
        cout << "List is empty";
        return NULL;
    }
    else if(ptr == n) {
        ptr->next = n->next;
        return n;
    }
    else if(ptr != n) {
        ptr->next = n->next;
        return n;
    }
}
```
Josephus Problem

In computer science and mathematics, the Josephus Problem (or Josephus permutation) is a theoretical problem. Following is the problem statement:

There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number k which indicates that k-1 persons are skipped and kth person is killed in circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

For example, if n = 5 and k = 2, then the safe position is 3. Firstly, the person at position 2 is killed, then person at position 4 is killed, then person at position 1 is killed. Finally, the person at position 5 is killed. So the person at position 3 survives.

If n = 7 and k = 3, then the safe position is 4. The persons at positions 3, 6, 2, 7, 5, 1 are killed in order, and person at position 4 survives.

The problem has following recursive structure.

\[
\text{josephus}(n, k) = (\text{josephus}(n - 1, k) + k - 1) \mod n + 1
\]

\[
\text{josephus}(1, k) = 1
\]
After the first person (kth from beginning) is killed, n-1 persons are left. So we call josephus(n - 1, k) to get the position with n-1 persons. But the position returned by josephus(n - 1, k) will consider the position starting from k%n + 1. So, we must make adjustments to the position returned by josephus(n - 1, k).

Following is simple recursive implementation of the Josephus problem. The implementation simply follows the recursive structure mentioned above.

```c
#include <stdio.h>
int josephus(int n, int k)
{
    if (n == 1)
        return 1;
    else
        /* The position returned by josephus(n - 1, k) is adjusted because the
           recursive call josephus(n - 1, k) considers the original position
           k%n + 1 as position 1 */
        return (josephus(n - 1, k) + k - 1) % n + 1;
}

// Driver Program to test above function
int main()
{
    int n = 14;
    int k = 2;
    printf("The chosen place is %d", josephus(n, k));
    return 0;
}
```

Output:

The chosen place is 13
Time Complexity: O(n)
The problem can also be solved in $O(k \log n)$ time complexity which is a better solution for large $n$ and small $k$. 