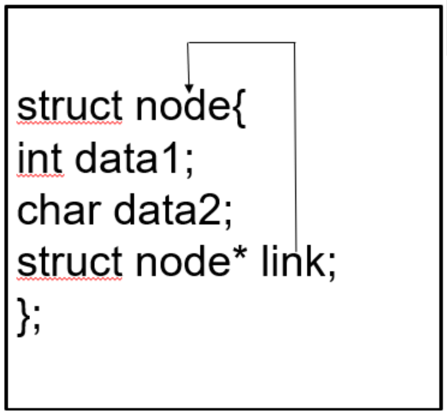
Module 2: Linked List and Memory Management

Singly Linked List - Operations on Linked List, Stacks and Queues using Linked List, Polynomial representation using Linked List; Doubly Linked List; Circular Linked List;
Memory allocation - First-fit, Best-fit, and Worst-fit allocation schemes; Garbage collection and compaction.

Self-Referential Structures

Self-referential structures, also known as recursive structures, allow a data structure to contain a reference to the same type of structure. This recursive property enables the creation of intricate and hierarchical data representations, such as linked lists, trees, and graphs. Self-referential structures play a fundamental role in designing complex and interconnected data models within programming.
Self-Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member. In other words, structures pointing to the same type of structures are self-referential in nature. One or more pointers points to the structure of same type is what a self-referential structure is.



```
struct node{
int data1;
char data2;
struct node* link;
};
```

For example,

```
struct node{
        int data1;
        char data2;
        struct node* link;
};

void main()
```

```
{
struct node *ob;
}
```

Here,

- link' is a pointer to a structure of type 'node'.
- Hence, the structure 'node' is a self-referential structure with 'link' as the referencing pointer.
- An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

Types of Self-Referential Structures

1. Self-Referential Structure with Single Link: In this type, a data structure contains a reference (or link) to another instance of the same structure. This commonly appears in linked lists, where each element (node) points to the next one. The last node typically points to a null or a special marker, indicating the end of the structure.
2. Self-Referential Structure with Multiple Links: This type involves a structure where each instance has references to multiple instances of the same structure. A classic example is a tree, where a node has references to its child nodes. This allows for hierarchical and branched data representations, commonly found in structures like binary trees or n-ary trees. Each link signifies a relationship or connection between different elements within the structure.

Applications of Self-Referential Structure

- Linked Lists: Self-referential structures are key components in the creation of linked lists, where each element holds a reference to the next, allowing for dynamic and non-contiguous memory allocation.
- Stacks and Queues: Implementations of stacks and queues often leverage self-referential structures, enabling easy addition and removal of elements based on last-in, first-out (LIFO) or first-in, first-out (FIFO) principles.
- Trees: Self-referential designs are instrumental in constructing tree structures, where nodes contain references to their child nodes, forming hierarchical arrangements like binary trees or n-ary trees.

- Graphs: Graphs, representing complex relationships between entities, utilize self-referential structures where nodes point to other nodes, facilitating the modeling and analysis of intricate networks.

Dynamic memory allocation

Dynamic memory allocation is crucial for creating and managing dynamic data structures, enabling a program to allocate space for structures whose sizes are determined only at runtime. The essential tools for dynamic memory allocation are the functions malloc and free, along with the sizeof operator.

When using malloc, memory is allocated, and a void pointer to the allocated space is returned; if the allocation request cannot be fulfilled, malloc returns NULL. Typically, malloc is employed in conjunction with the sizeof operator to allocate memory based on the size of a particular data type, providing flexibility and adaptability in memory management during program execution.

```
struct Node {
        int data;
        struct Node *next;
        };
        void main() {
        struct Node newnode = (struct Node*)malloc(sizeof(struct Node));
}
```

evaluates sizeof(struct Node) to determine the size in bytes of a structure of type struct Node,

allocates a new area in memory of that number of bytes and stores a pointer to the allocated memory in variable newnode.

The free function deallocates memory. The memory is returned to the system so that it can be reallocated in the future. To free memory dynamically allocated by the preceding malloc call, use the statement free(newnode)

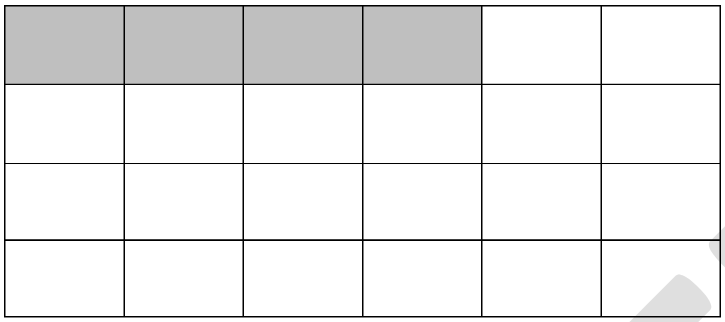
Memory Leaks

Memory leaks occur when a program forgets to clean up unused memory, causing a gradual buildup that can eventually slow down the program or even make it crash. It's important to free up memory that's no longer needed to prevent these issues and keep the program running smoothly.

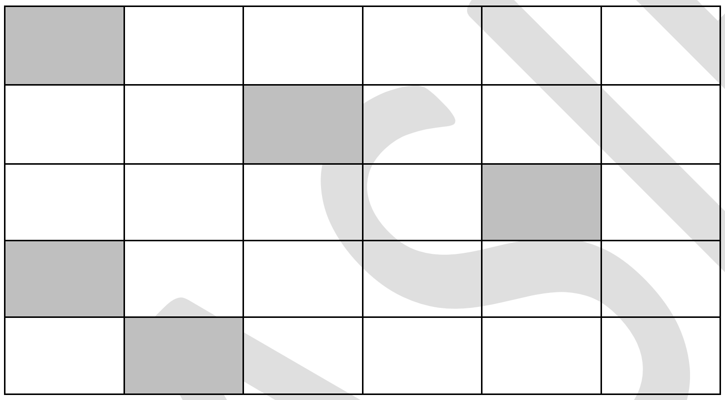
Linked List

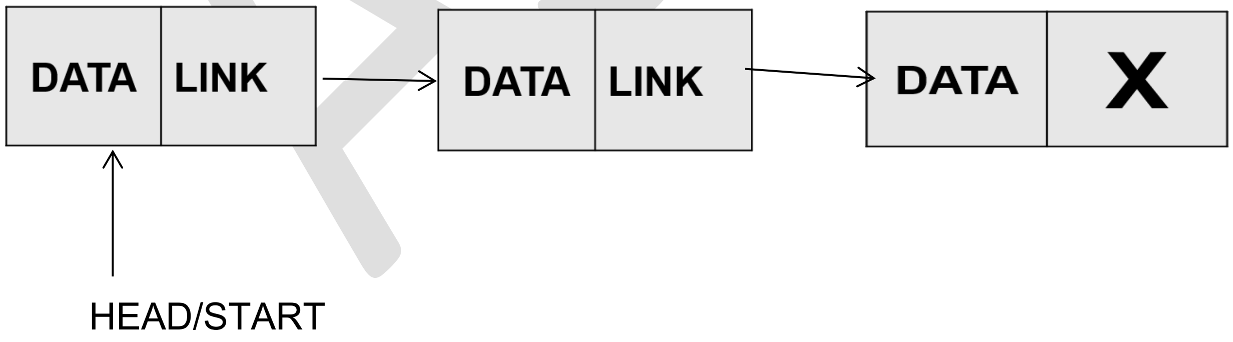
A linked list is a linear data structure that consists of nodes, each containing data and a reference to the next node in the sequence. Unlike arrays, linked lists allow dynamic memory allocation, enabling efficient insertion and deletion of elements at any position. The flexibility and dynamic nature of linked lists make them useful for various applications, particularly when the size of the data structure is unpredictable or needs to change frequently.

Array's need Contiguous Block of Memory



Linked List: Dynamic Memory Allocation. Memory can grow dynamically





A Linked list is an ordered collection of finite, homogeneous data elements called nodes where linear order is maintained by means of links or pointers.

Nodes: Linked list consist chain of elements, in which each element is referred to as a node. Node consist of two parts:

- Data: Refers to the information held by the node
- Link: Hold the address of the next node in the list

Head/ Start: Contains a pointer to the first data node in the list or a null pointer if the list is empty.

Advantages of Linked Lists:

1. Dynamic Size:
   Linked lists allow for dynamic memory allocation, enabling the structure to grow or shrink as needed during program execution.
2. Efficient Insertions and Deletions:Inserting or deleting elements within a linked list is more efficient than in arrays, especially when dealing with large datasets, as it only requires adjusting pointers.
3. No Wasted Memory: Unlike arrays, linked lists do not have a fixed size, preventing wasted memory space, as memory is allocated only when needed.

Disadvantages of Linked Lists:

1. Sequential Access:
   Linked lists have slower sequential access compared to arrays since elements are not stored in contiguous memory locations.
2. Extra Memory Usage:
   Each element in a linked list requires additional memory for storing the data and the reference to the next node, which can lead to higher overall memory usage compared to arrays.
3. Complexity and Overhead:
   Managing pointers in linked lists adds complexity to the code, and there's an overhead associated with maintaining these references, especially in doubly-linked lists where each node has references to both the next and previous nodes.

Classification of linked List

1. Single Linked List
2. Circular Linked List
3. Double linked List

Linked list / simple linked list / singly linked list

A node in a single linked list can be represented by a self-referential structure, which is a data structure that contains a pointer to itself. This means that a node can store its own data, as well as a reference to the next node in a linked list or tree. This allows nodes to be easily connected to each other, and it makes it easy to traverse data structures that are made up of nodes.

Here is an example of a node in a single linked list self-referential structure for a node in a linked list:

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

This structure has two members: data and next. The data member stores the node's data, and the next member stores a pointer to the next node in the linked list.

To create a new node, we would use the following code:

```
struct node *newnode = malloc(sizeof(struct node));
newnode->data = 10;
newnode->next = NULL;
```

This code creates a new node with the value 10, and it sets the next pointer to NULL, which indicates that the node is the end of the linked list.

We can then add the new node to the linked list by updating the next pointer of the previous node to point to the new node. For example, if the linked list already contains one node, we would update the next pointer of the first node to point to the new node as follows:

```
first_node->next = newnode;
```

This will create a linked list with two nodes, with the new node at the end.

Operations on a single Linked List

1. Insertion

2. Traversing
3. Deletion
4. Copy
5. Merging
6. Searching
7. Reversing
8. Sorting

Simple Linked List – Insertion

A linked list is a dynamic data structure where elements, represented as nodes, are connected by references. Inserting elements into a linked list can be achieved through different methods. Elements can be inserted at various positions in a linked list:

- The various positions are:
- Insert at front (as a first element)
- Insert at end (as a last element)
- Insert at particular position

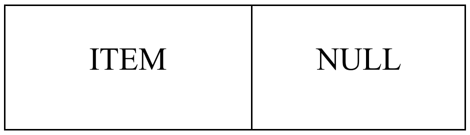
Before any insertion, it is crucial to dynamically allocate memory for the new node, commonly done using the malloc() function in C or equivalent memory allocation functions in other languages. The provided C code exemplifies these concepts, demonstrating functions for inserting at the front and at the end of a linked list.

Simple Linked List – Insertion at End

Inserting a node at the end of a linked list is achieved by checking if the list is not null. If the list is not empty, the new node is inserted at the end. However, if the list is initially empty, the operation results in the creation of a new node, which is then initialized as the header of the list. This process ensures the formation of a list with the new node becoming the starting point when the list is empty. The insertion procedure is as follows

1. Initially the list is empty, so the pointer HEAD points to NULL
   HEAD = NULL
2. Now we create space for the node using malloc function and store data in that node and its next address link part will be set to NULL

```
Newnode=malloc(node)
Newnode→data=item
Newnode→nextaddr=NULL
```



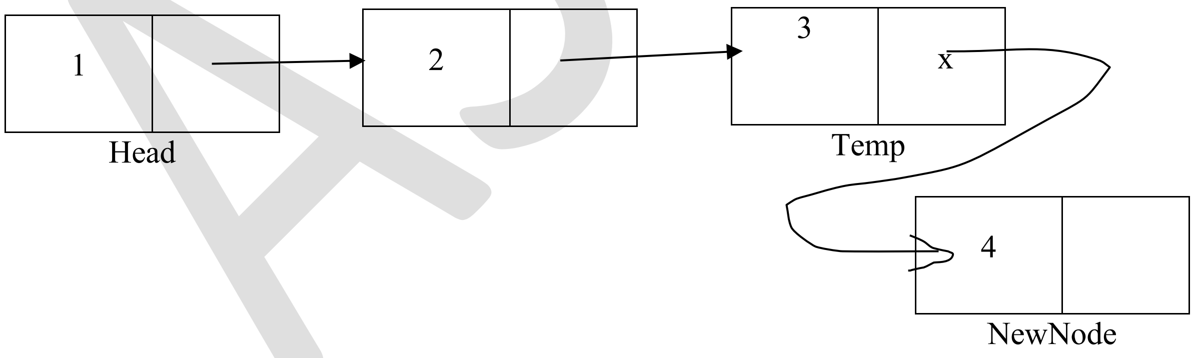
Newnode

3. Now we need to check if the list is empty then there is only one node that is our Newnode so that we can point head to the Newnode

```
If(head==NULL) then
        Head=Newnode
```

4. Else if there are other nodes already in the list then we need to find the last node and get a pointer called temp and point it to the last node. Last node will be the node with nextaddr having value NULL. We can use a loop to traverse the list till we find the last node whose nextaddr is NULL. Then we can simply point the temp node to the newnode making it the new last node.

```
Temp=head
While(Temp→nextaddr!=NULL)
        Temp=Temp.nextaddr
EndWhile
Temp.nextaddr=Newnode
```



Algorithm Simple Linked List – Insertion at End

Algorithm -INSERT AT END or CREATE LIST if list is null (header, item)

Input: head is the pointer to the header node and item is the element to be inserted.

Output: A node with element item is inserted at end of the list.

Data Structure: Linked list

```
1. Start
2. Read item;
3. newnode=malloc(node);
4. newnode.data=item;
5. newnode.next=NULL
6. if(head==NULL) then                    // list empty
       a. HEAD=newnode
       b. Exit
7. Else
       a. temp=HEAD
       b. while(temp.next!=NULL)
              i. temp=temp.next
       c. end while
       d. temp.next=newnode;
8. Endif
9. Stop
```

Simple Linked List – Insertion at Beginning of the list

Inserting a node at the beginning of a linked list involves checking if the list is not null. If the list is not empty, a new node is added at the front. However, if the list is initially empty, the operation results in the creation of a new node, and this new node is designated as the HEAD, establishing the starting point for the list. This ensures that the list is properly initialized with the new node becoming the initial element when the list is empty. The insertion procedure is as follows:

1. Initially the list is empty, so the pointer HEAD points to NULL

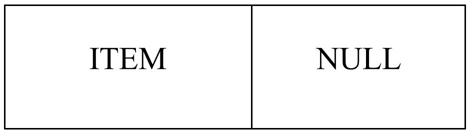
   HEAD = NULL

2. Now we create space for the node using malloc function and store data in that node and its next address link part will be set to NULL

   Newnode=malloc(node)

   Newnode→data=item

   Newnode→nextaddr=NULL



Newnode

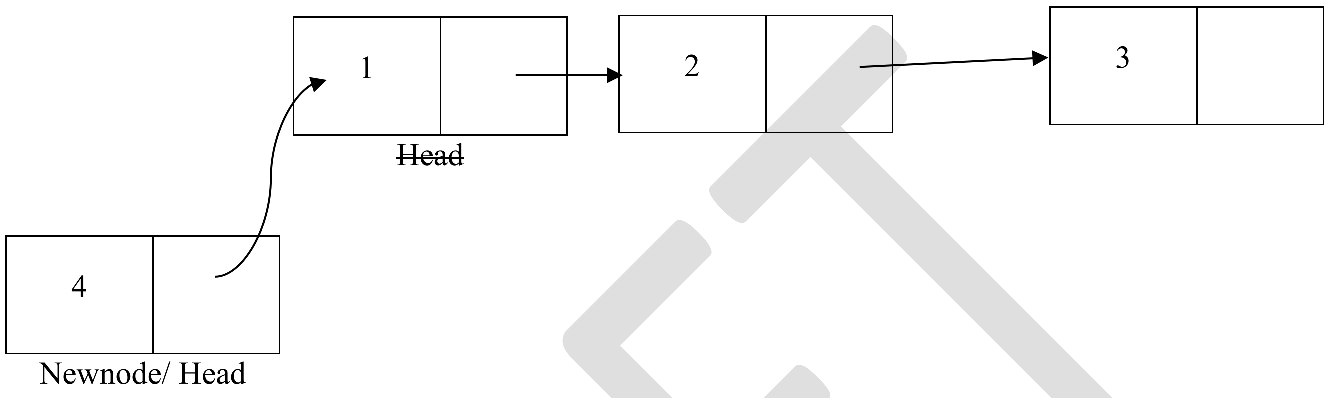
3. Now we need to check if the list is empty then there is only one node that is our Newnode so that we can point head to the Newnode

   If(head==NULL) then

Head=Newnode

4. If the list is not empty and there are elements in the list insertion at beginning of the list is a O(1) process which involves two steps

```
Newnode.next=head;
head= Newnode
```



Algorithm Insertion at Front

Algorithm -INSERT AT FRONT (head, item)

Input: HEAD is the pointer to the header node and item is the element to be inserted.

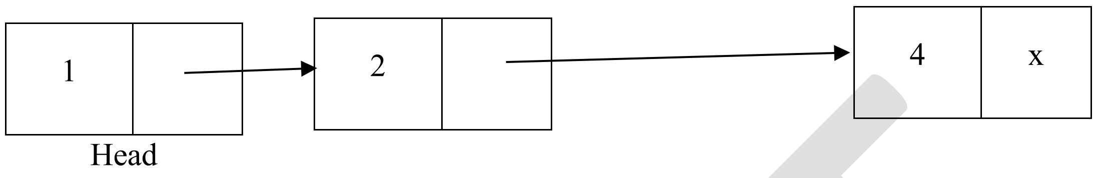
Output: A node with element item is inserted at front of the list if list is not null.

Data Structure: Linked list

```
1. Start
2. Read item;
3. newnode=malloc(node);
4. newnode.data=item;
5. newnode.next=NULL
6. if(head==NULL) then    // list empty
    a. HEAD=newnode
    b. Exit
7. Else
    a. newnode.next=head;
    b. head= newnode
8. Endif
9. Stop
```

Simple Linked List – Insertion at Particular position

Inserting an element at a particular position in a linked list involves the flexibility to add it at any desired location within the list. For instance, to insert the element 3 after element 2, the linked list is traversed to find the node representing element 2. Subsequently, a new node containing the element 3 is created, and its pointers are adjusted to seamlessly integrate it into the list after element 2. This dynamic insertion process allows for the customization of the linked list, accommodating elements at specific positions as needed. Let’s consider we need to insert a node with value 3 after node with value 2 as in below figure, this involves the following procedure:



1. Initially the list is empty, so the pointer HEAD points to NULL
   HEAD = NULL
2. Now we create space for the node using malloc function and store data in that node and its next address link part will be set to NULL
   Newnode=malloc(node)
   Newnode→data=item
   Newnode→nextaddr=NULL

Newnode

3. Now we need to check if the list is empty then there is only one node that is our Newnode so that we can point head to the Newnode
   If(head==NULL) then

   Head=Newnode
4. If the list is not empty we need to traverse the list to find the location of the desired value key for that use a temporary variable temp and point it initially to head and then traverse the list.

```
temp=head
While( temp!=NULL and temp.data!=key)
        temp=temp.nextaddr
```

5. When the above loop exit we will either find the desired position to insert or loop terminate when temp becomes NULL. Then we can point the newnodes nextaddr to where currently temp was pointing to
   newnode.nextaddr = TEMP1.nextaddr

Algorithm- Insertion at particular position

Algorithm: INSERT AT ANY (header, item, key)

Input: head is the pointer to the header node and item is the element to be inserted and key being the data one node after which the item to be inserted.

Output: A node with element item is inserted at given position if

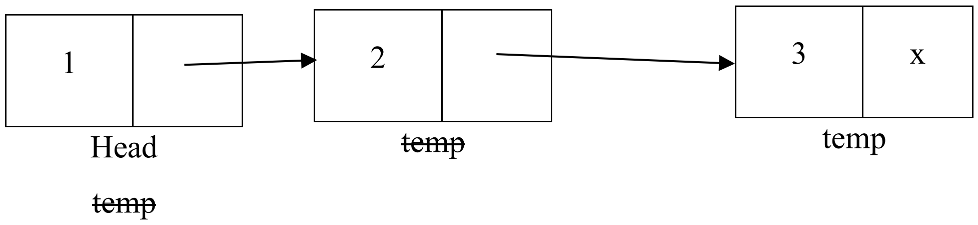
the list is not null.

Data Structure: Linked list

```
1. Start
2. Read item;
3. newnode=malloc(node);
4. newnode.data=item;
5. newnode.next=NULL
6. if(head==NULL) then                    // list empty
     a. PRINT LIST EMPTY
     b. Exit
7. Else
     a. temp=HEAD
     b. while(temp.next!=NULL && temp.data!=key)
          i. temp=temp.next
     c. end while
     d. if(temp==NULL)
          i. PRINT("KEY NOT FOUND")
         ii. EXIT
     e. Endif
     f. newnode.next=temp.next;
     g. temp.next=newnode;
8. Stop
```

Traversing a linked list

Traversing a linked list involves systematically visiting each node in the list, starting from the initial node and progressing through the subsequent nodes until reac3hing the last node. This process allows for the examination or manipulation of data within each node, facilitating operations such as searching, printing, or modifying elements as needed throughout the linked list structure.



Algorithm TRAVERSE (header)

Input: Header is the pointer to the first node if not null.

Data Structure: A single linked list.

Steps:

```
1. If (head=NULL) then
        1. Print("List is empty")
        2. Exit
2. Else
        1. Temp=head
        2. While(temp!=NULL)
                1. Print(temp.data)
                2. Temp=temp.nextaddr
        3. EndWhile
3. Stop
```

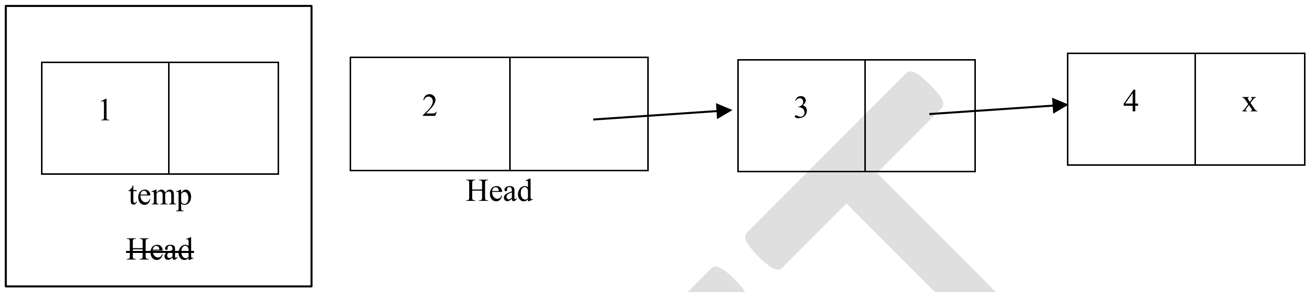
Linked List Deletion

Linked list deletion involves removing nodes from various positions within the list, and there are different cases to consider:

1. Deletion from the Front of the List:
    a. The first node (head) is removed.
    b. The next node becomes the new head.
    c. The memory occupied by the deleted node is freed using the free() function.
2. Deletion from the End of the List:
    a. The last node is identified and removed.
    b. The previous node becomes the new last node.
    c. The memory occupied by the deleted node is freed using free().
3. Deletion from Any Position of the List:
    a. The node at the specified position is located.
    b. The pointers of the preceding and succeeding nodes are adjusted to skip the node to be deleted.
    c. The memory occupied by the deleted node is freed using free().

The free() function is crucial in the deletion process as it releases the memory allocated by malloc() (or a similar memory allocation function) back to the system's memory pool. This ensures that the memory used by the deleted node is made available for future use, preventing memory leaks.

Case 1: Deletion from the FRONT of the list



Here we do not need to access the first node anymore. So, will move the HEAD pointer to the next node. And then ensure we use free operator to deallocate the memory area.

Algorithm DELETE FRONT(header)

Input: Header is the pointer to the header node of the linked list

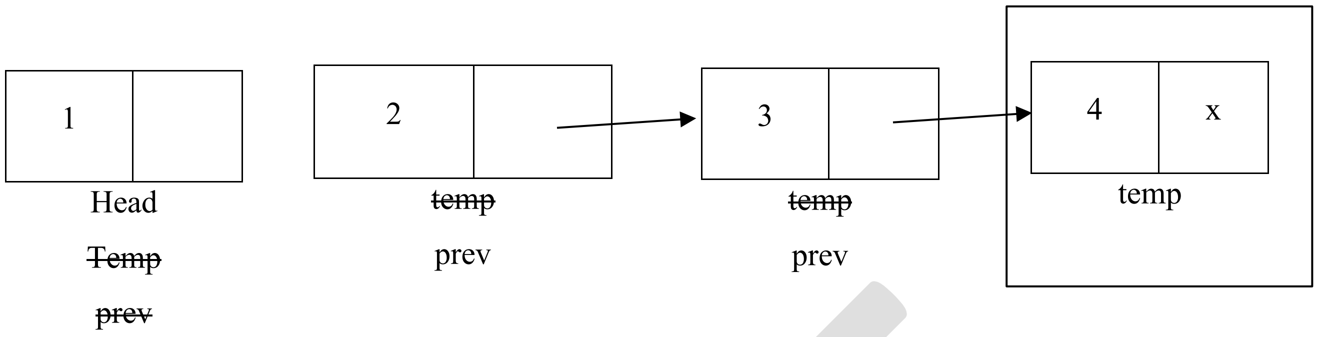
Output: A single linked list eliminating the node at the front.

Data Structure: Linked list

Steps:

```
1. If(head=NULL)
        1. Printf("List empty")
        2. Exit
2. EndIf
3. Temp=head
4. Head=head.nextaddr
5. FREE (temp)
6. Stop
```

CASE 2: Deletion from the END of the list



Here we use two pointers called 'prev' and 'temp' for identifying thee last node and the node just before it which will be the new last node after deletion. A loop can be used to find the last node whose 'nextaddr' is null. There is one special case here which is when there is only one single node called head then only thing we have to do is assign head as NULL. And finally we should free the node to deallocate space.

Algorithm: DELETE LAST (head)

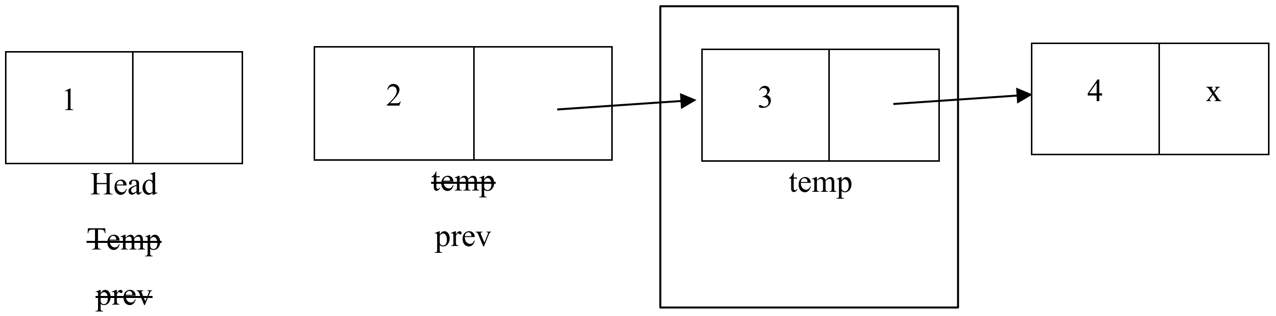
Input: Head is the pointer to the header node of the linked list

Output: A single linked list eliminating the node at the end.

Data Structure: Linked list

```
1. If(head==NULL)
     a. Print("List Empty")
     b. Exit
2. Endif
3. Temp=Head
4. if(Head->next==NULL)// only one element
     a. Head=NULL;
5. Else
     a. While(Temp-> next!=NULL)
          i. Prev=Temp;
          ii. Temp=Temp->next
     b. End While
     c. Prev->next=NULL
6. Free(Temp)
7. Endif
8. Stop
```

CASE 3: Deletion from MIDDLE of the list



This deletion algorithm, operating on a linked list, checks for the presence of the key in each node. If the key is found, the node is deleted, and memory is freed. The algorithm uses a 'Flag' variable to determine if the key is found, and it employs 'Temp' and 'Prev' pointers to navigate the list. If the key is in the header, the header is adjusted accordingly. If the key is not found, a message is printed. The algorithm ensures proper deletion from the middle, handling different scenarios effectively.

Algorithm: DELETE MIDDLE (header, key)

Input: Header is the pointer to the header node of the linked list, key is the data content of the node to be deleted.

Output: A single linked list except the node with content as key.

Data Structure: Linked list.

Steps

```
1. Flag=0
2. If(head==NULL)
      a. Print("List Empty")
      b. Exit
3. Else if(head->data==key)
      a. Item=head->data
      b. Temp=head
      c. head=head->link
      d. free(temp)
      e. Exit
4. Else
      a. Temp=head
      b. While(temp->link!=NULL)
            i. Prev=temp
           ii. Temp=temp.link
          iii. If(temp->data==key)
                  1. Flag=1
                  2. Exit While
           iv. Endif
```

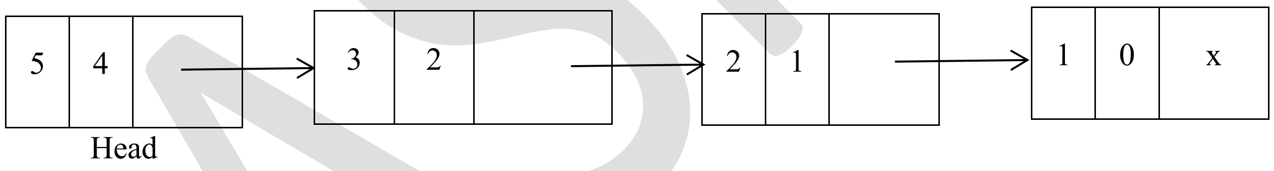
```
        c. End while
        d. If(flag==0)
              i. Print("key not found")
             ii. Exit
        e. Else
              i. Item=temp->data
             ii. Prev->link=temp->link
            iii. Free(temp)
        f. Endif
    5. Endif
    6. Stop
```

Representing a polynomial using Linked list and its operations

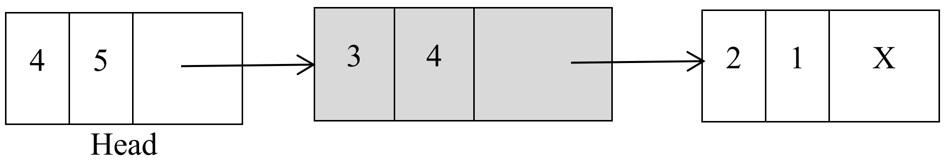
We have already seen how to represent a polynomial using an array and now let's see how to use linked list for the same. For this we can use a single linked list with two data part for storing coefficient and exponent along with the link part pointing to the next term in the polynomial. Each term in the polynomial will be in a single node holding three pieces of information:

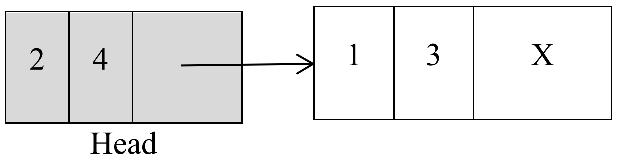
1. Coefficient
2. Exponent
3. Address of the next node in sequence

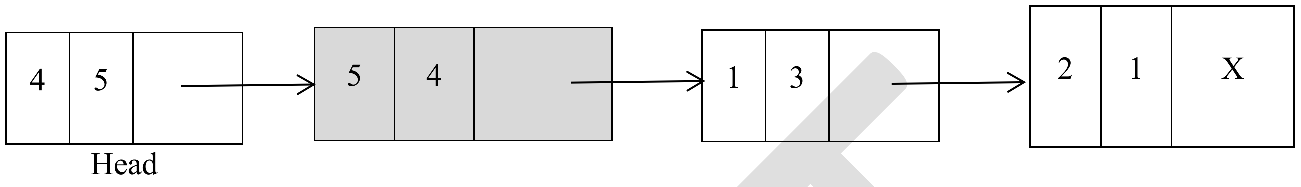
For example, let's take a polynomial like $5x^4+3x^2+2x+1$ it can be represented in a linked list with four nodes each node representing a term in the polynomial



Now let's see how polynomial addition using linked list would work. Lets take two polynomials for the same $4x^5+3x^4+2x$ and $2x^4+x^3$







In this algorithm designed for polynomial addition, the two input polynomials are efficiently represented as linked lists, with each node encapsulating both a coefficient and an exponent. The algorithm initiates with pointers P1current and P2current positioned at the heads of the input polynomial linked lists (p1head and p2head). A pivotal while loop is employed, persisting until both input polynomials have been exhaustively processed.

Within each iteration, a new node (Newnode) is dynamically created through memory allocation, serving as a container for the sum of corresponding terms or the term possessing the greater exponent. The algorithm diligently examines the exponents of the current nodes in both input polynomials, updating the new node accordingly. If both exponents are equal, the coefficients are added, and the result is stored in the new node. In cases where one exponent surpasses the other, the new node encapsulates the term with the greater exponent, and the respective pointer (P1current or P2current) advances to the subsequent node.

The construction of the result polynomial linked list is meticulously orchestrated through the use of pointers Rhead and Rcurrent. If the result polynomial is initially empty (Rhead is NULL), the new node instantaneously assumes the role of the head; otherwise, it is seamlessly linked to the existing result polynomial. This systematic process ensures the coherent assembly of the result polynomial, with the pointers facilitating an organized structure.

Algorithm: POLYNOMIAL_ADDITION (polyhead1, polyhead2)

Input: polyhead1is the pointer to the header node of the first polynomial and polyhead2 is the pointer to the header node of the second polynomial

Output: A resultant polynomial with sum of the two, input polynomial

Data Structure: Linked list.

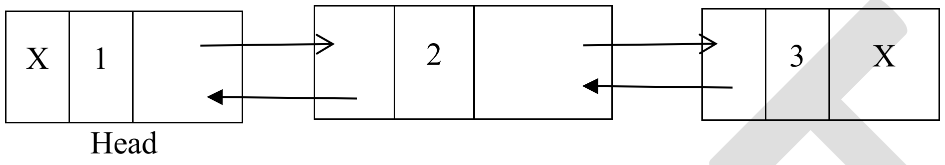
Steps

```
1. P1current=p1head
2. P2current=p2head
3. Rhead=NULL
4. While(p1current!=NULL OR p2current!=NULL)
    1. Newnode=malloc (NODE)
    2. Newnode.link=NULL
    3. If (p1current!=NULL AND p2current!=NULL)
        1. If(p1current.exp=p2current.exp)
            1. Newnode.coeff=p1current.coeff+p2current.coeff
            2. Newnode.exp=p1current.exp
            3. P1current=p1current.link
            4. P2current=p2current.link
        2. ElseIf(p1current.exp>p2current.exp)
            1. Newnode.coeff=p1current.coeff
            2. Newnode.exp=p1current.exp
            3. P1current=p1current.link
        3. ElseIf(p2current.exp>p1current.exp)
            1. Newnode.coeff=p2current.coeff
            2. Newnode.exp=p2current.exp
            3. P2current=p2current.link
        4. Else if (p1current!=NULL)
            1. Newnode.coeff=p1current.coeff
            2. Newnode.exp=p1current.exp
            3. P1current=p1current.link
        5. Else if (p2current!=NULL)
            1. Newnode.coeff=p2current.coeff
            2. Newnode.exp=p2current.exp
            3. P2current=p2current.link
        6. EndIf
        7. If(rhead=NULL)
            1. Rhead=newnode
            2. Rcurrent = rhead
        8. Else
            1. Rcurrent.link=newnode
            2. Rcurrent=newnode
        9. EndIf
5.EndWhile
```

Doubly Linked List

We have already seen the singly linked list which is a one way list and can only move in one direction now lets look into doubly linked list which has the capability to move in either direction from left to right or vice versa. Doubly Linked list consists of chain of elements, in which each element is referred to as a node. A node consists of three parts:

- Data: Refers to the information held by the node
- RightLink: Holds the address of the next node in the list
- LeftLink: Holds the address of the previous node in the list



Operations on doubly linked list

1. Doubly linked list insertion at the end

Algorithm: DoublyLinkedListInsertionAtEnd

Input:

- item: Data to be stored in the new node.

- head: Pointer to the head node of the doubly linked list.

Output:

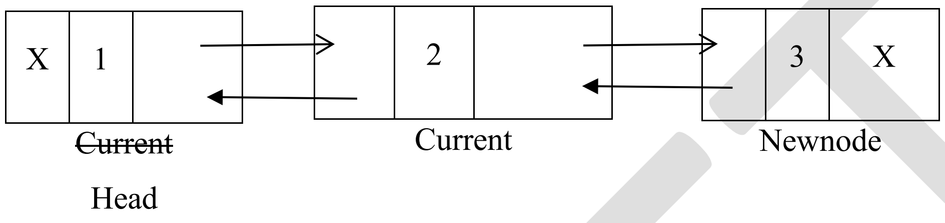
- Doubly linked list with the new node inserted at the end.

Steps:

```
1. Newnode = malloc(NODE)
2. Newnode.data = item
3. Newnode.leftaddr = NULL
4. Newnode.rightaddr = NULL
5. If(head == NULL)
   1. Head = Newnode
6. Else
   1. Current = head
   2. While(Current.rightaddr != NULL)
      1. Current = Current.rightaddr
   3. EndWhile
   4. Current.rightaddr = Newnode
```

```
    5. Newnode.leftaddr = Current
7. EndIf
8. Stop
```

This algorithm dynamically allocates memory for a new node, assigns the provided data to it, and sets its left and right pointers to NULL. If the linked list is empty, the new node becomes the head. Otherwise, it traverses the list to find the last node and then inserts the new node after it, adjusting pointers accordingly.



2. Doubly linked list insertion at the front

Algorithm: DoublyLinkedListInsertionAtFront

Input:

- item: Data content of the node to be inserted.

- head: Pointer to the head node of the doubly linked list.

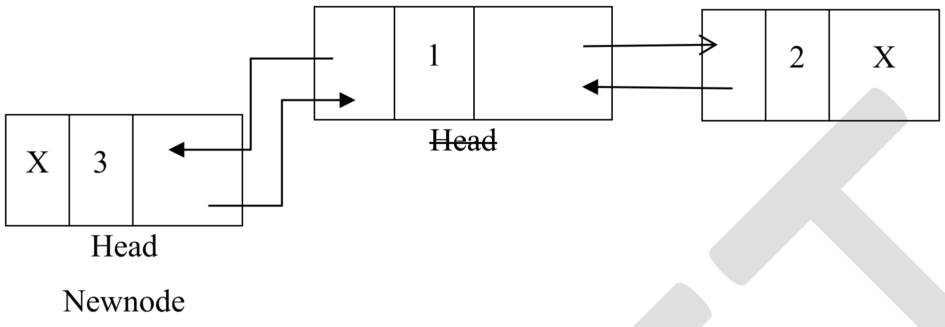
Output:

- Doubly linked list with the new node inserted at the front.

Steps:

```
1. Newnode = malloc(NODE)
2. Newnode.data = item
3. Newnode.leftaddr = NULL
4. Newnode.rightaddr = NULL
5. If(head == NULL)
    1. Head = Newnode
6. Else
    1. Newnode.rightaddr = head
    2. Head.leftaddr = Newnode
    3. Head = Newnode
7. EndIf
8. Stop
```

This algorithm dynamically allocates memory for a new node, assigns the provided data to it, and sets its left and right pointers to NULL. If the linked list is empty, the new node becomes the head. Otherwise, it adjusts pointers to insert the new node at the front of the list, updating the head accordingly.



This algorithm dynamically allocates memory for a new node, assigns the provided data to it, and sets its left and right pointers to NULL. If the linked list is empty, the new node becomes the head. Otherwise, it adjusts pointers to insert the new node at the front of the list, updating the head accordingly.

3. **Doubly linked list insertion at any position, say after 2**

Algorithm: DoublyLinkedListInsertionAfterPosition

Input:

- item: Data content of the node to be inserted.
- key: Data content of the node after which the new node will be inserted.
- head: Pointer to the head node of the doubly linked list.

Output:

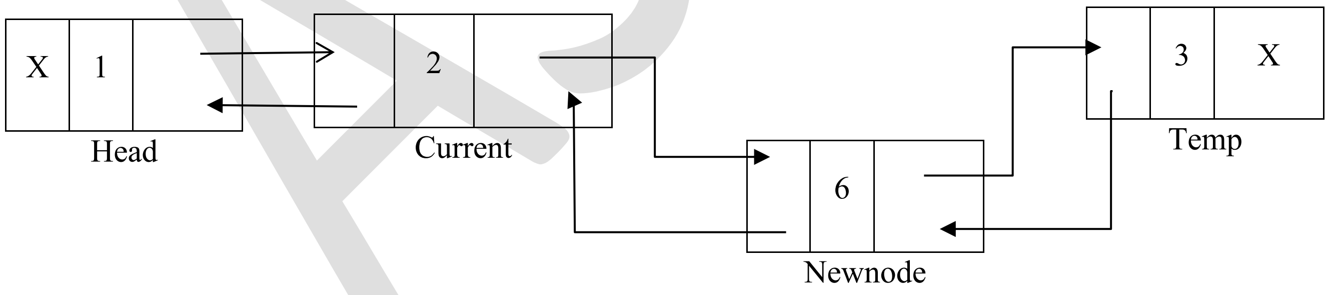
- Doubly linked list with the new node inserted after the specified position.

Steps:

```
1. Flag = 0
2. Newnode = malloc(NODE)
3. Newnode.data = item
4. Newnode.leftaddr = NULL
5. Newnode.rightaddr = NULL
```

```
6. If(head == NULL)
   1. Print("List is Empty..")
7. Else
   1. Current = head
   2. While(current != NULL)
      1. If(current.data == key)
         1. Flag = 1
         2. ExitWhile
      2. EndIf
      3. Current = current.rightaddr
   3. EndWhile
   4. If(flag == 1)
      1. If(current.rightaddr != NULL)
         1. Temp = current.rightaddr
         2. Temp.leftaddr = Newnode
         3. Newnode.rightaddr = Temp
      2. EndIf
      3. Current.rightaddr = Newnode
      4. Newnode.leftaddr = Current
   5. Else
      1. Print("Key not Found..")
   6. EndIf
8. EndIf
9. Stop
```

This algorithm inserts a new node with the specified item after the node containing the specified key. It traverses the list to find the node with the given key. If the key is found, the new node is inserted after it by adjusting the pointers accordingly. If the key is not found, a message is printed indicating that the key was not found.



If current nodes right pointer is NULL then it is same as insertion at end.

4. Doubly linked list deletion from front

Algorithm: DoublyLinkedListDeletionFront

Input:

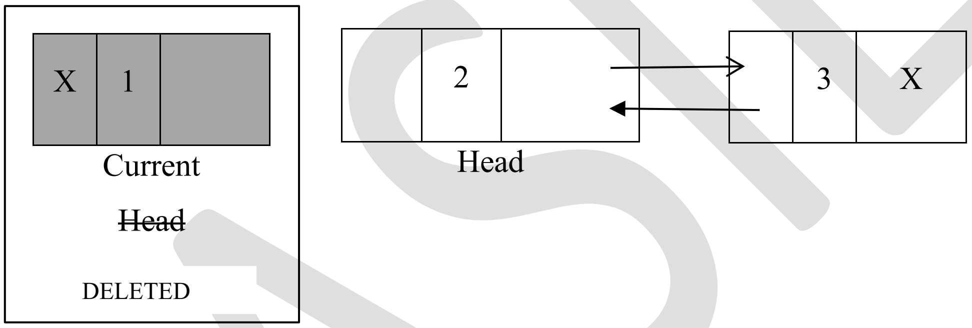
- head: Pointer to the head node of the doubly linked list

Output:

- List with the front node removed.

Steps:

```
1. If (head == NULL) then
    1. Print("List is Empty...")
2. Else If (head.rightaddr == NULL)
    1. Item = head.data
    2. FREE(head)
    3. Head = NULL
3. Else
    1. Current = head
    2. Item = current.data
    3. Head = head.rightaddr
    4. Head.leftaddr = NULL
    5. FREE(current)
4. EndIf
5. Stop
```



5. Doubly linked list deletion from end

Algorithm: DoublyLinkedListDeletionEnd

Input:

- head: Pointer to the head node of the doubly linked list.

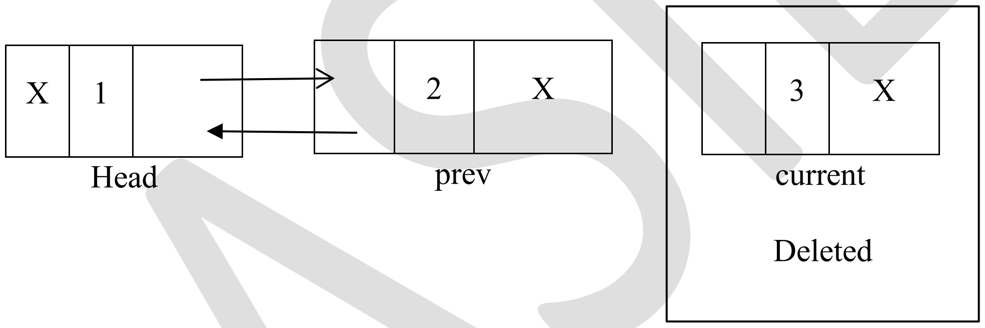
Output:

- List with the last node removed.

Steps:

```
1. If (head == NULL) then
   1. Print("List is Empty")
2. Else If (head.rightaddr == NULL)
   1. Item = head.data
   2. FREE(head)
   3. Head = NULL
3. Else
   1. Current = head
   2. While (current.rightaddr != NULL)
      1. Prev = current
      2. Current = current.rightaddr
   3. EndWhile
   4. Item = current.data
   5. Prev.rightaddr = NULL
   6. FREE(current)
4. Stop
```

This algorithm first checks if the linked list is empty. If it is, it prints a message indicating that the list is empty. If the list contains only one node, that node is removed, and the head pointer is updated to NULL. If the list contains more than one node, the algorithm traverses the list to find the last node, updates the previous node's pointer to NULL, and frees the memory occupied by the removed node.



6. Doubly linked list deletion from any position

Algorithm: DoublyLinkedListDeleteAny

Input:

- item: Data content of the node to be deleted.

- head: Pointer to the head node of the doubly linked list.

Output:

- Doubly linked list with the node containing the specified item removed.

Steps:

```
1. If (head == NULL)
   1. Print("List Empty..No Deletion")
2. Else
   1. If (head.data == item AND head.rightaddr == NULL)
      1. FREE(head)
      2. Head = NULL
   2. Else If (head.data == item AND head.rightaddr != NULL)
      1. Current = head
      2. head = head.rightaddr
      3. Head.leftaddr = NULL
      4. FREE(current)
   3. Else
      1. Current = head
      2. While (current != NULL AND current.data != item)
         1. Current = current.rightaddr
      3. EndWhile
      4. If (current != NULL)
         1. Prev = current.leftaddr
         2. Next = current.rightaddr
         3. If (current.rightaddr != NULL)
            1. Next.leftaddr = prev
         4. Prev.rightaddr = next
         5. FREE(current)
      5. Else
         1. Print("Item not Found")
   4. EndIf
3. EndIf
4. Stop
```

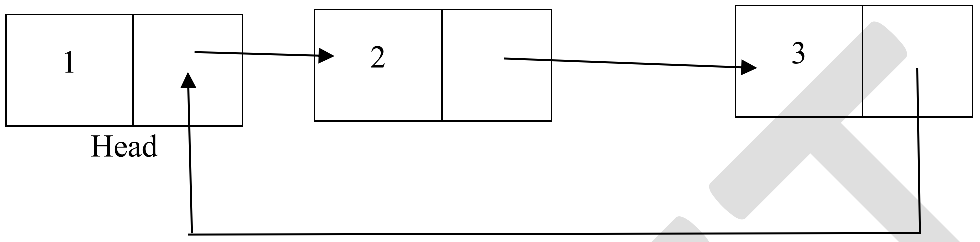
Circular Linked List

A circular linked list is a variant of a linked list where the last node's link field doesn't point to null, but rather holds the address of the first node, thus forming a circle or loop within the list. This means that traversal through the list can start from any node, and it will eventually reach back to the starting node.

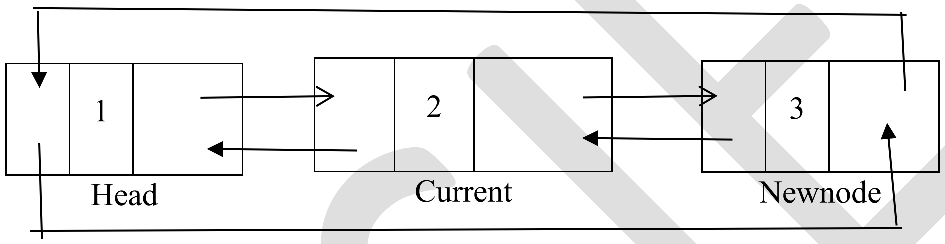
In contrast to a standard singly linked list, where the last node's link field is null, the circular linked list provides a continuous loop, allowing for more efficient operations like iterative traversals without the need to check for the end of the list. Additionally, circular linked lists are useful in applications like scheduling algorithms, where tasks need to cycle indefinitely.

This circular structure also simplifies certain operations, such as inserting or deleting nodes, as there is no need to handle special cases for the last node pointing to null. However, it's essential to manage pointers carefully to avoid infinite loops during traversal or unintended circular references. Overall, circular linked lists offer flexibility and convenience in certain scenarios due to their cyclic nature.

Below figure is of a singly circular linked list.



There can be doubly circular linked list also, it is represented in figure below.



Operations on circular linked list

1. Insertion in a circular linked list: A node can be added to circular linked list in three ways:
    a) Insertion in an empty list
    b) Insertion at the beginning of the list
    c) Insertion at the end of the list
    d) Insertion in between the nodes

a) Insertion in an empty list

When inserting into an empty circular linked list, the process is straightforward since there are no existing nodes to consider. Here's the algorithm for inserting into an empty circular linked list:

Algorithm: InsertionInEmptyCircularList

Input:

- data: Data to be stored in the new node.

- head: Pointer to the head node of the circular linked list.

Output:

- Circular linked list with a single node containing the provided data.

Steps:

```
1. Newnode=malloc(node)
2. Newnode.data=item
3. Newnode.link=NULL
4. if (last != NULL)
       i. return last;
5. last = Newnode;
6. Newnode ->next = last;
```

In this algorithm, when the circular linked list is initially empty, a new node is created with the provided data. Since it's the only node in the list, its link field points to itself, effectively forming a loop. The head pointer is then updated to point to this new node, establishing it as the starting point of the circular list.

b) Insertion at the beginning of the list

Algorithm: CircularLinkedListInsertionAtBeginning

Input:

- item: Data content of the node to be inserted.

- last: Pointer to the last node of the circular linked list (or NULL if the list is empty).

Output:

- Circular linked list with the new node inserted at the beginning.

Steps:

```
1. Allocate memory for the new node using malloc().
2. Set the data of the new node to the provided item.
3. If the last node is NULL (i.e., the list is empty):
   a. Set the link (or next) pointer of the new node to point to itself.
   b. Set the last pointer to point to the new node.
   c. Return the new node (as it is now the only node in the list).
4. Otherwise (if the list is not empty):
   a. Set the link (or next) pointer of the new node to point to the next node after the last node.
   b. Set the next pointer of the last node to point to the new node.
   c. Return the last node (unchanged).
```

c) Insertion at the end of the list

Algorithm: CircularLinkedListInsertionAtEnd

Input:
- item: Data content of the node to be inserted.
- last: Pointer to the last node of the circular linked list (or NULL if the list is empty).

Output:
- Circular linked list with the new node inserted at the end.

Steps:
```
1. Allocate memory for the new node using malloc().
2. Set the data of the new node to the provided item.
3. If the last node is NULL (i.e., the list is empty):
    a. Set the link (or next) pointer of the new node to point to itself.
    b. Set the last pointer to point to the new node.
    c. Return the new node (as it is now the only node in the list).
4. Otherwise (if the list is not empty):
    a. Set the link (or next) pointer of the new node to point to the next node after the last node.
    b. Set the next pointer of the new node to point to the first node in the list.
    c. Set the next pointer of the last node to point to the new node.
    d. Update the last pointer to point to the new node.
    e. Return the last node (unchanged).
```

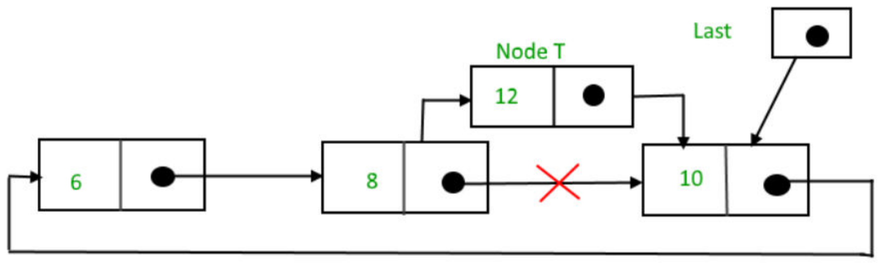
d) Insertion in between the nodes

Algorithm: CircularLinkedListInsertionBetweenNodes

Input:
- item: Data content of the node to be inserted.
- prevNode: Pointer to the node after which the new node will be inserted in the circular linked list.

Output:
- Circular linked list with the new node inserted between two nodes.

Steps:
```
1. Allocate memory for the new node using malloc().
2. Set the data of the new node to the provided item.
3. Set the link (or next) pointer of the new node to point to the next node after prevNode.
4. Set the next pointer of prevNode to point to the new node.
5. Return prevNode (unchanged) or the new node if needed.
```



Stack implementation using linked list

To implement a Last-In-First-Out (LIFO) stack using a linked list, where insertion and deletion are performed at one end (the top), you can follow these operations

- PUSH: Insertion at the end of the linked list.
- POP: Deletion from the end of the linked list.

Here's a step-by-step algorithm for implementing a stack using a linked list:

Algorithm: Stack PUSH using Linked List

Data Structure:

- Node: Structure to represent each node in the linked list.

Steps:

```
1. newnode = malloc(sizeof(struct node))
2. newnode >data=item
3. newnode >link=TOP
4. TOP=newnode
```

Algorithm: Stack POP using Linked List

Data Structure:

- Node: Structure to represent each node in the linked list.

Steps:

```
1. If TOP=null
        Print " Stack Empty"
        Exit
2. Else
        temp=TOP
        Item=temp->data
        TOP=temp->link
        FREE(temp)
```

Queue using Linked List

To implement a First-In-First-Out (FIFO) queue using a linked list, where insertion is performed at one end (rear) and deletion is performed at the other end (front), you can follow these operations:

ENQUEUE: Insertion at the rear (end) of the linked list.

DEQUEUE: Deletion from the front of the linked list.

Here's a step-by-step algorithm for implementing a queue using a linked list:

Algorithm: ENQUEUE using Linked List

Data Structure:

- Node: Structure to represent each node in the linked list.

Steps

```
1. newnode = malloc(sizeof(struct node));
2. if (newnode == NULL)
    ◦ return ERROR;
3. newnode->data = item;
4. newnode->link = NULL;
5. if (REAR == NULL)
    ◦ FRONT = REAR = newnode;
6. Else
    ◦ REAR->link = newnode
    ◦ REAR = newnode
```

Algorithm: DEQUEUE using Linked List

Data Structure:

- Node: Structure to represent each node in the linked list.

Steps

```
1. if (FRONT == NULL)
   return ERROR;
2. temp = FRONT;
3. item = temp->data;
4. FRONT = FRONT->link;
5. if (FRONT == NULL)
        REAR = NULL;
6. free(temp);
7. return item;
```

Feature	Arrays	Linked Lists
Memory Allocation	Contiguous memory allocation	Non-contiguous memory allocation
Size	Fixed size	Dynamic size
Insertion/Deletion	Costly (Shift elements if not at end)	Efficient (No shifting, just update pointers)
Access Time	Constant time (O(1))	Linear time (O(n))

Feature	Arrays	Linked Lists
Random Access	Efficient (Direct access using index)	Inefficient (Traversal required)
Search Time	Efficient (Binary search for sorted)	Linear time (O(n))
Memory Overhead	Minimal	Additional memory for pointers
Memory Wastage	Can have wasted memory if size is large	Minimal, as memory is allocated dynamically
Implementation Complexity	Simple	Moderate
Application	Suitable for fixed-size collections	Suitable for dynamic-size collections

These comparisons can vary depending on specific implementations and use cases, but they provide a general overview of the differences between arrays and linked lists.

Memory management techniques

Memory management techniques involve two primary approaches:

Static Memory Management:

- In static memory management, the memory required for various data structures in a program is allocated before the program starts its execution.
- Once memory is allocated, its size remains fixed throughout the program's execution.
- Memory allocated statically cannot be extended or returned to the memory block for use by other programs concurrently.
- Static memory management is suitable for scenarios where the memory requirements are known and fixed beforehand, and there's no need for dynamic allocation and deallocation during program execution.

Dynamic Memory Management:

- Dynamic memory management allows the user to allocate and deallocate memory as needed during the execution of programs.
- This approach is particularly useful in scenarios such as multiprogramming environments or single environments where multiple programs reside in memory simultaneously.
- In dynamic memory management, memory allocation and deallocation can be performed based on the program's requirements, which may vary during execution.

- Data structures such as linked lists are commonly used to implement dynamic memory management, as they provide flexibility in memory allocation and deallocation.
- Dynamic memory management enables efficient utilization of memory resources and helps optimize memory usage based on program requirements.

These two memory management techniques cater to different requirements and constraints, providing flexibility and efficiency in memory utilization based on the nature of the program and its memory needs.

The heap is a region of the main memory that is utilized for dynamic memory allocation upon request by a program. The responsibility of managing free memory blocks, assigning specific blocks to user programs as needed, and reclaiming memory from unused blocks is undertaken by a component of the operating system known as the memory manager.

A simple organization of memory involves maintaining a linked list of all memory blocks, which is updated whenever a block is allocated or deallocated. The blocks in such linked lists can be organized in various ways, typically based on block sizes or block addresses.

There are two common approaches for organizing memory blocks:

- Fixed Block Size:
    - In this approach, all memory blocks have the same predetermined size.
    - Each block can accommodate a fixed amount of data.
    - This simplifies memory management but may lead to inefficiencies if the required memory size varies significantly across allocations.
- Variable Block Size:
    - Here, memory blocks can have different sizes based on the specific requirements of the allocated data.
    - The size of each block is determined dynamically, allowing for more efficient memory utilization.
    - This approach offers flexibility but may require more complex memory management algorithms to handle variable block sizes effectively.

Both approaches have their advantages and trade-offs, and the choice between fixed and variable block sizes depends on factors such as the nature of the applications being run, memory usage patterns, and performance considerations.

Fixed Block Storage and Variable memory allocation

Fixed block storage, also known as fixed-size allocation, is a straightforward memory management approach where each memory block is of the same predetermined size. This method simplifies storage maintenance as it allows for uniformity in memory allocation and management.

On the other hand, variable memory allocation involves a memory management system that can handle requests for blocks of various sizes. Unlike fixed block storage, where all blocks have the same size, variable memory allocation allows programs to request memory blocks in a wide range of sizes based on their specific requirements.

In variable memory allocation:

The memory management system must be able to allocate memory blocks dynamically, adjusting the size of the allocated block to match the program's needs.

This flexibility enables efficient utilization of memory resources, as programs can request only the amount of memory they require, reducing wastage.

However, managing variable-sized blocks can be more complex compared to fixed block storage, requiring sophisticated algorithms for memory allocation and deallocation.

Overall, while fixed block storage simplifies memory management with uniform block sizes, variable memory allocation provides greater flexibility to accommodate the diverse memory requirements of programs. The choice between the two depends on factors such as the nature of the applications being run and the efficiency of memory utilization desired.

Memory de-allocation

Garbage collection is a memory management technique used to reclaim memory occupied by objects or nodes that are no longer in use by the program. In systems with manual memory management, such as languages like C and C++, developers are responsible for explicitly deallocating memory when it is no longer needed. However, in systems with garbage collection, this process is automated, reducing the likelihood of memory leaks and simplifying memory management for developers.

Here's how garbage collection typically works:

Detection of Unused Nodes:

Garbage collection involves identifying objects or nodes that are no longer accessible or referenced by the program. These objects are considered "garbage" because they cannot be reached or used by the program.

Reclamation of Memory:

Once unused nodes are identified, the memory they occupy can be reclaimed and returned to the available memory pool.

This reclaimed memory can then be used for new allocations by the program.

Triggering Garbage Collection:

Garbage collection is typically triggered when the system detects a shortage of available memory.

When a memory allocation request cannot be satisfied due to insufficient available memory, the garbage collector is invoked to reclaim memory from unused objects.

Search and Restore:

The garbage collector searches through all nodes or objects in the system, identifying those that are no longer accessible from any external pointer or reference. Once identified, these inaccessible nodes are marked as available for reuse, effectively restoring them to the available memory pool.

Garbage collection helps prevent memory leaks and improves memory efficiency by automatically reclaiming memory from unused objects. However, it comes with a performance overhead as the garbage collector needs to periodically traverse the entire object graph to identify and reclaim unused memory. Different garbage collection algorithms exist, each with its own trade-offs between memory efficiency and runtime performance.

Compaction

Compaction is a memory management technique that involves rearranging memory to consolidate used memory blocks or nodes at one end of the memory space, while relocating available (free) memory blocks to the other end. This process aims to reduce fragmentation and optimize memory utilization by creating larger contiguous blocks of free memory.

	INTERNAL FRAGMENTATION	**EXTERNAL FRAGMENTATION**
Basic	It occurs when fixed sized memory blocks are allocated to the processes	It occurs when variable size memory space are allocated to the processes dynamically.
Occurrence	When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.
Solution	The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Compaction, paging and segmentation.

Memory allocation schemes

Memory allocation schemes determine how memory is allocated to processes or programs in a computer system. The first fit allocation scheme assigns memory to the first available hole that is large enough to accommodate the requested memory size. In contrast, the best fit scheme selects the smallest hole that is sufficient for the memory request, aiming to minimize wasted memory. Conversely, the worst fit scheme allocates memory from the largest available hole, potentially leading to larger fragments of unused memory. Each scheme has its trade-offs; first fit is simple but may lead to fragmentation, best fit aims to minimize waste but can be inefficient due to frequent searches, while worst fit may result in larger holes of unused memory. The choice of allocation scheme depends on factors such as the system's memory management goals and the characteristics of the workload being executed.

First Fit:

- Assigns memory to the first available hole that is large enough to accommodate the requested memory size.
- Simple and efficient as it only requires scanning memory until a suitable hole is found.

- May lead to fragmentation over time as small gaps of unused memory can accumulate between allocated blocks.
- Example: If a process requests 50 KB of memory, the first fit algorithm would allocate memory from the first hole it encounters that is at least 50 KB in size.

Next Fit:

- Similar to first fit, but starts searching for free memory from the location where the previous allocation ended.
- Continues the search from the last allocated block onwards, rather than starting from the beginning of memory each time.
- Aims to reduce fragmentation by potentially filling in gaps more efficiently than first fit.
- Example: If a process requests 30 KB of memory after a previous allocation of 20 KB, the next fit algorithm would search for a hole starting from the location immediately after the previous allocation.

Best Fit:

- Selects the smallest hole that is sufficient to accommodate the requested memory size.
- Minimizes wasted memory by utilizing the smallest available hole.
- May result in inefficient memory usage due to frequent searches for suitable holes, especially in systems with a large number of small holes scattered throughout memory.
- Example: If a process requests 60 KB of memory, the best fit algorithm would search for the smallest available hole of at least 60 KB.

Worst Fit:

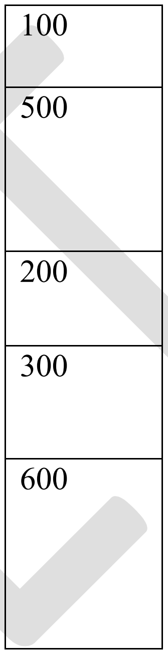
- Allocates memory from the largest available hole, even if it is larger than the requested memory size.
- Aims to minimize fragmentation by leaving larger holes for future allocations.
- May lead to inefficient use of memory as it can leave behind smaller holes that may not be suitable for future allocations.
- Example: If a process requests 40 KB of memory, the worst fit algorithm would search for the largest hole of at least 40 KB and allocate memory from there.

Problem

Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

First-fit:

212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition
(new partition 288K = 500K - 212K)
426K must wait



Best-fit:

212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition

Worst-fit:

212K is put in 600K partition
417K is put in 500K partition
112K is put in 300K partition
426K must wait
In this example, best-fit turns out to be the best.