Unit 3 – Linked List

Dynamic Data Structure

Data structure is a way of storing and organising data efficiently such that the required operations on them can be performed is efficient with respect to time as well as memory. Simply, Data Structure is used to reduce complexity (mostly the time complexity) of the code.

Data structures can be two types:

1. Static Data Structure

2. Dynamic Data Structure

Static Data structure

In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.

Dynamic Data Structure

In Dynamic data structure the size of the structure in not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.

Static Data Structure VS Dynamic Data Structure

Static Data structure has fixed memory size whereas in Dynamic Data Structure, the size can be randomly updated during run time which may be considered efficient with respect to memory complexity of the code. Static Data Structure provides easier access to elements with respect to dynamic data structure. Unlike static data structures, dynamic data structures are flexible.

Properties

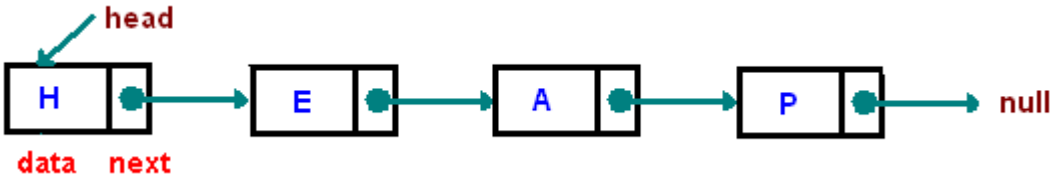
In static allocation, the memory allocation is at the time of compilation and that cannot alter its size at the time of the program execution. So, this will arise the problem of **memory shortage and memory wastage.**

In dynamic allocation, the memory allocation will be at the time of execution. This will eliminate the problem of memory shortage and memory wastage. So, the dynamic allocation will permit the **allocation, De-allocation, and reallocation** of memory at the time of the execution

Introduction to Linked Lists

Linked List is a very commonly used linear data structure which consists of group of nodes in a sequence.

Each node holds its own data and the address of the next node hence forming a chain like structure.



Each element (we will call it a node) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

Advantages

1. A linked list is a dynamic data structure.
2. The number of nodes in a list is not fixed and can grow and shrink on demand.
3. Any application which has to deal with an unknown number of objects will need to use a linked list.
4. Insertion and deletion are easier and efficient.

Disadvantages

Compare with an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

The linked list uses more memory compare with an array because extra bytes are required to store a reference to the next node.

Data structure specification

```
struct node
      {
        int info;
        struct  node *next;
       }
Typedef struct node NODE;
NODE *start;
```

Operations on Linked List

The basic operations are

1. Creation
2. Insertion
3. Deletion
4. Searching
5. Traversing
6. Display
7. Concatenation

Types of Linked Lists

Following are the types of linked list

Singly Linked List.

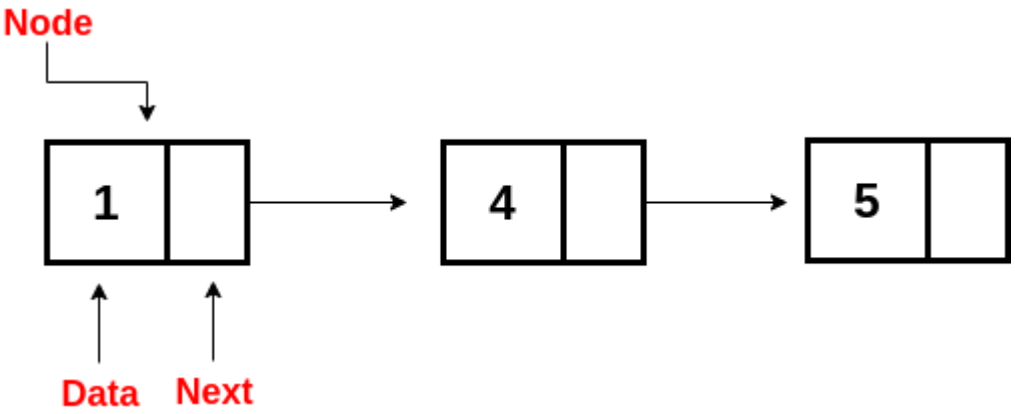
Doubly Linked List.

Circular Linked List.

Singly Linked List

A Singly-linked list is a collection of nodes linked together in a sequential way where each node of the singly linked list contains a data field and an address field that contains the reference of the next node.

The structure of the node in the Singly Linked List is



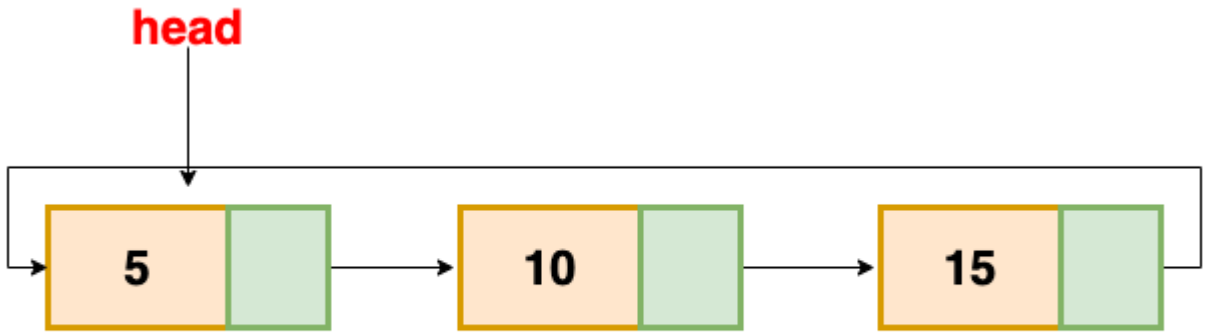
Doubly Linked List

A Doubly Linked List contains an extra memory to store the address of the previous node, together with the address of the next node and data which are there in the singly linked list. So, here we are storing the address of the next as well as the previous nodes.

Doubly Linked List

Circular Linked List

A circular linked list is either a singly or doubly linked list in which there are no NULL values. Here, we can implement the Circular Linked List by making the use of Singly or Doubly Linked List. In the case of a singly linked list, the next of the last node contains the address of the first node and in case of a doubly-linked list, the next of last node contains the address of the first node and prev of the first node contains the address of the last node.

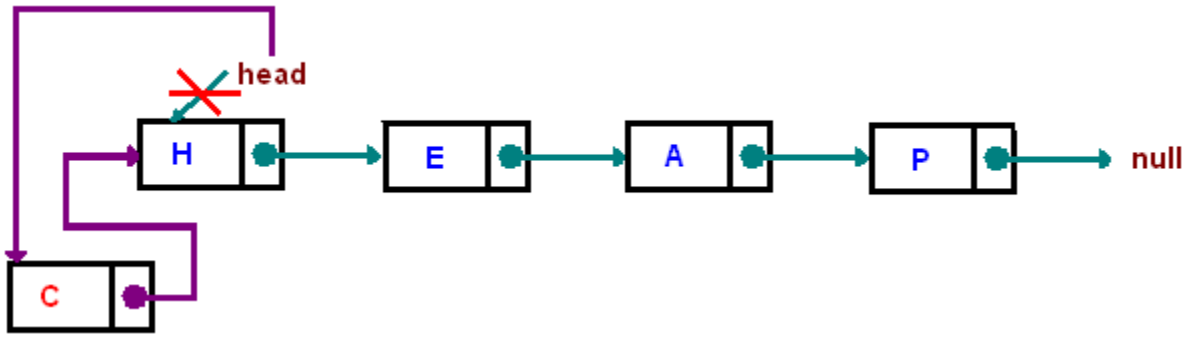


Circular Linked List

Linked list creation Program?

Linked list Node insertion

At the beginning

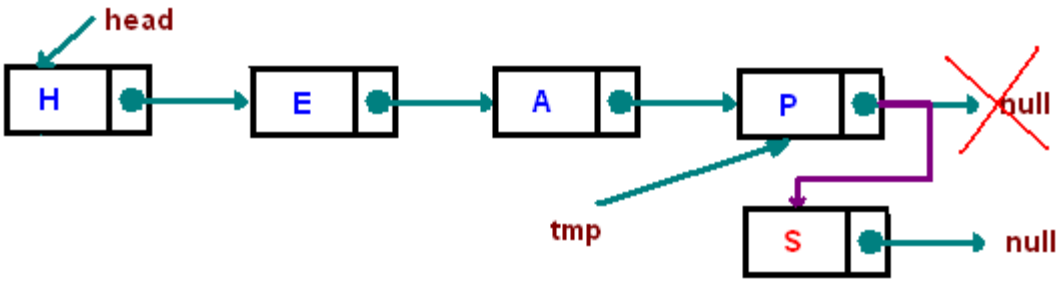


Algorithm

```
Sl_insertion_first(head, newvalue)
Begin
    newnode=new NODE // Dynamically create a new node for insertion
                                // NODE – user defined data type for representing a node
    newnode->info=newvalue
    newnode->next=NULL
    IF (head=NULL) THEN
        head=newnode
    ELSE
      Begin
          newnode->next=head
          head=newnode
      end

end
```

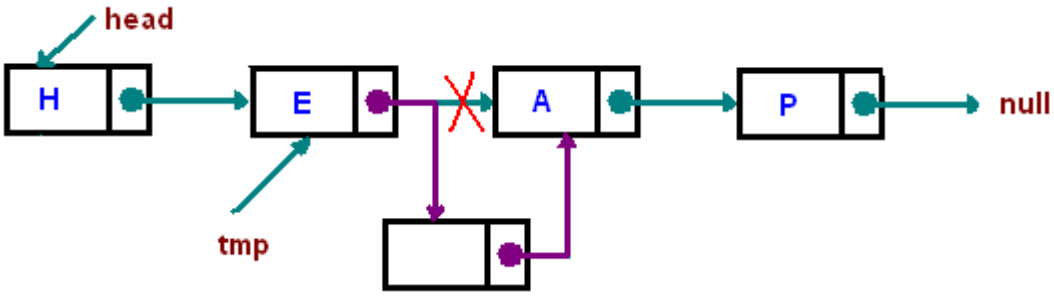
Insertion as a last node



Algorithm

```
Sl_insertion_last(head,newvalue)
Begin
    newnode=new NODE
    newnode->info=newvalue
    newnode->next=NULL
IF(head=NULL) Then
    head=newnode
Else
   Begin
     temp=head
      While(temp->next!=NULL) Then
           temp=temp->next
     temp->next=newnode
    end
End
```

Insertion Between Two Nodes



Algorithm

```
Sl_insertion_between(head,newvalue,pos)// pos-Position where new node is to be inserted
Begin
   newnode=new NODE
    newnode->info=newvalue
    newnode->next=NULL
IF(head=NULL) Then
    head=newnode
Else
   Begin
      Temp=head
       i=1
       While(i<pos)
         Begin
             prev=temp
```

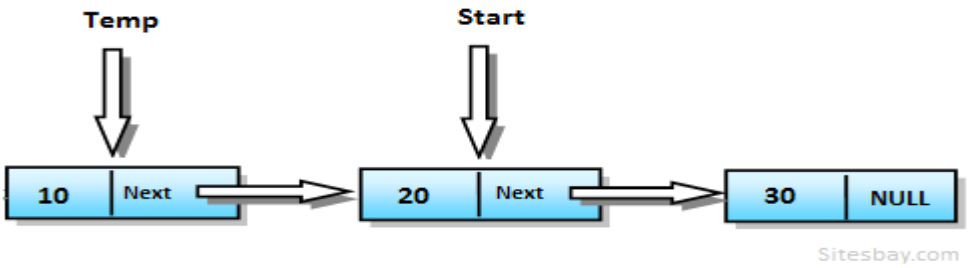
```
            temp=temp->next
            i=i+1;
        end
    newnode->next=temp
    prev-next=newnode
  end
End
```

Node Deletion in Singly Linked List

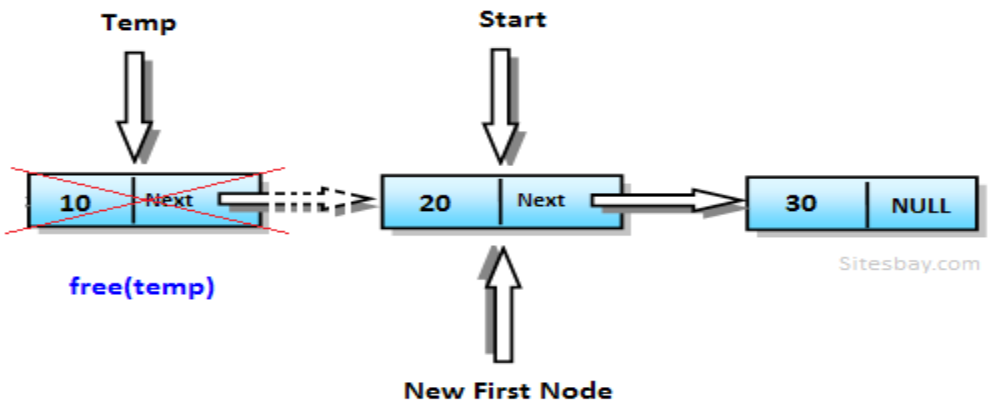
I First Node Deletion

1. **Store Current Start in Another Temporary Pointer**

2. **Move Start Pointer One position Ahead**



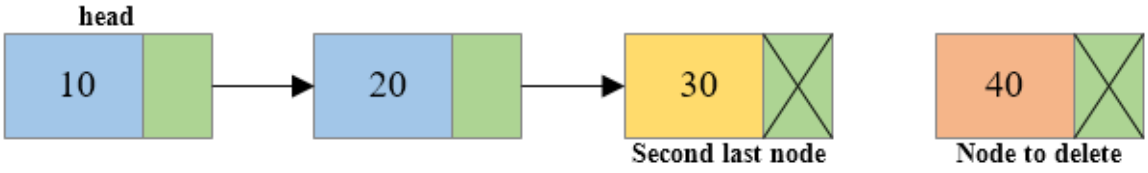
3. **Delete temp i.e Previous Starting Node as we have Update the link to the beginning**



Algorithm

```
Sl-deletion-first(head)
Begin
    If (head->next=NULL)
       temp=head
       head=NULL
   Else
       temp=head
      head=head->next
  delete(temp)
End
```

II Last Node Deletion

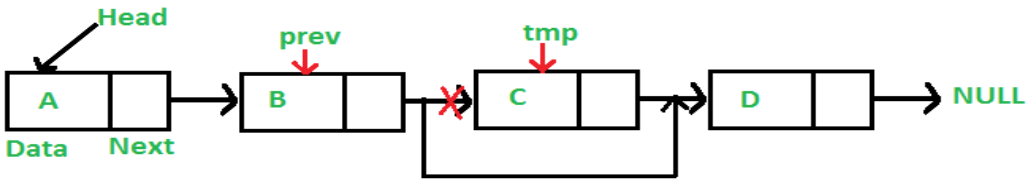


Algorithm

```
Sl_deletion_last(head)
 Begin
    If (head->next=NULL)
       temp=head
       head=NULL
    Else
```

```
    Begin
    temp=head
    While(temp->next!=NULL)
        prev=temp
        temp=temp->next
    prev->next=NULL
 delete(temp)
End
```

Delete a Node In Between Two Nodes



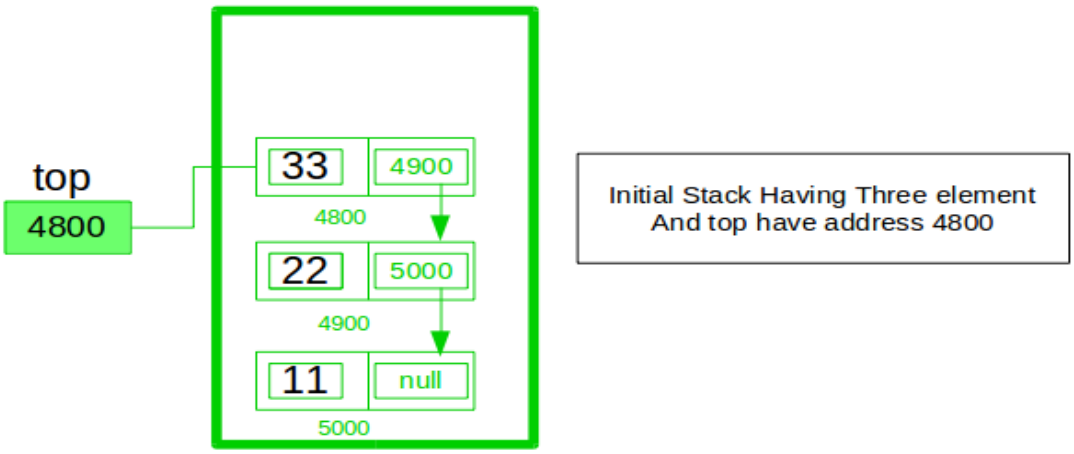
Algorithm

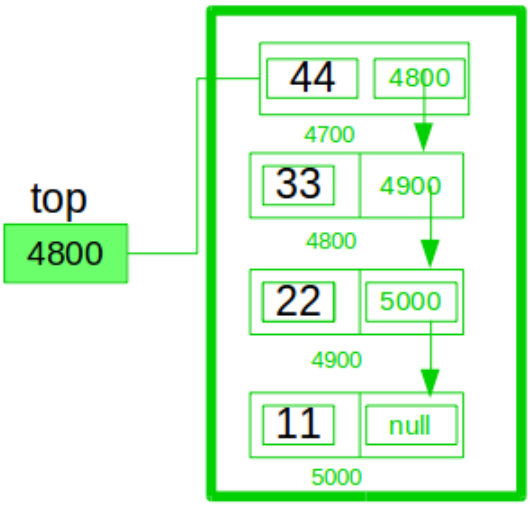
```
Sl_deletion_between(head, pos)
 Begin
    If (head->next=NULL)
       temp=head
       head=NULL
   Else
     Begin
      i=1, temp=head
      While(i<pos)
          prev=temp
          temp->temp->next
          i=i+1
     prev->next=temp->next
     temp->next=NULL
    end
  delete(temp)
```

End

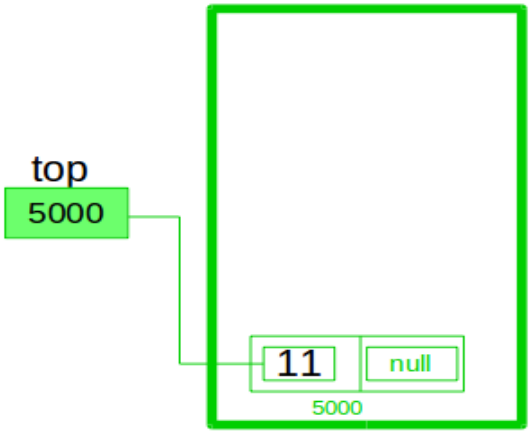
Linked Stacks (Linked list representation of stacks)

Implement a stack using single linked list concept. All the single linked list operations perform based on Stack operations LIFO(last in first out) and with the help of that knowledge we are going to implement a stack using single linked list. using single linked lists so how to implement here it is linked list means what we are storing the information in the form of nodes and we need to follow the stack rules and we need to implement using single linked list nodes so what are the rules we need to follow in the implementation of a stack a simple rule that is last in first out and all the operations we should perform so with the help of a top variable only with the help of top variables are how to insert the elements let's see





First create a temp node and assign 44
Into data field and top in link field
And in last assign temp assign into top




```
Pop three  element
    temp = top;
     top = top->link;
     temp->link = NULL;
     free(temp);
```

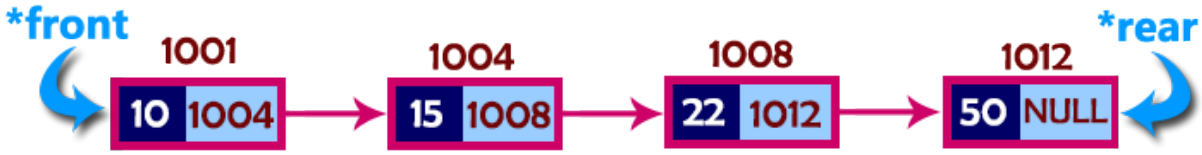
Linked List Implementation of Queues

The major problem with the queue implemented using an array is, it will work for an only fixed number of data values. That is the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data

(No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.

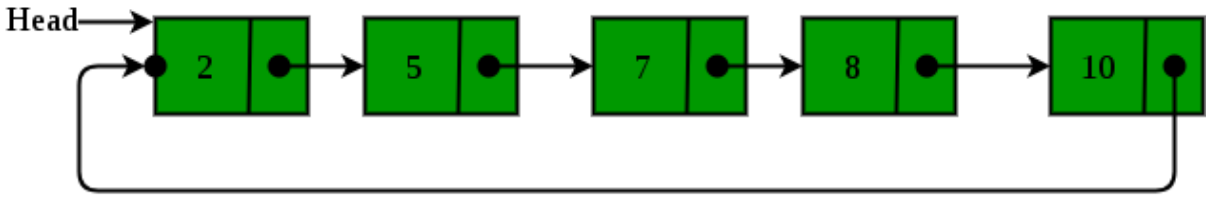
Example



In above example, the last inserted node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

Circular Linked Lists

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Advantages of Circular Linked Lists:

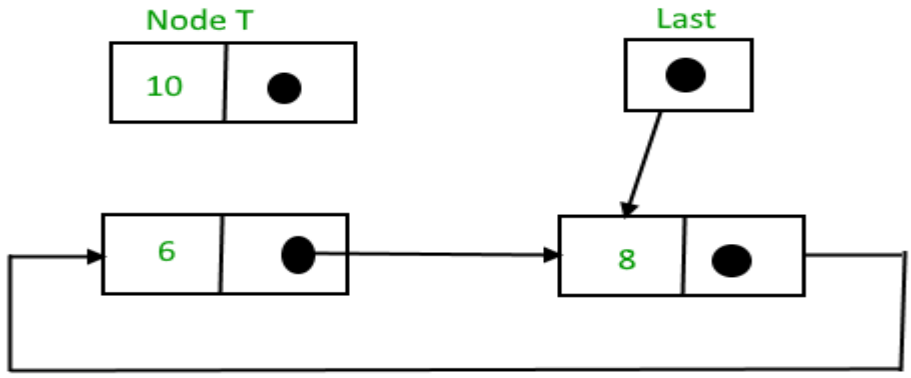
1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

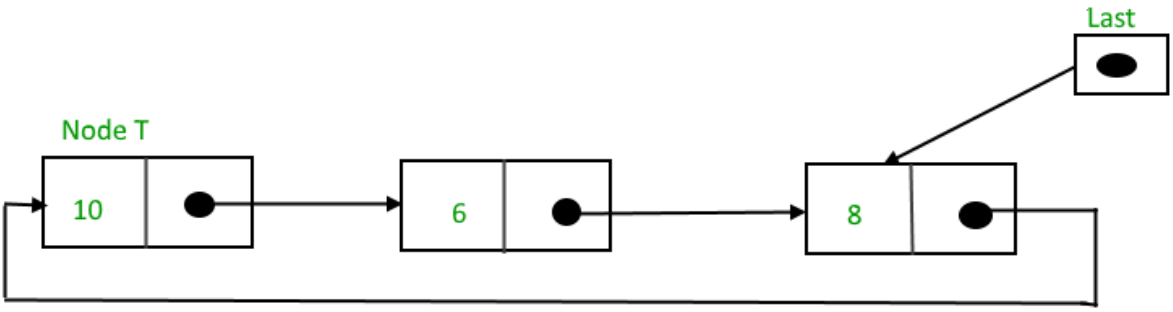
3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

Program to create a circular singly linked list

Node Insertion in Singly Circular Linked List



After Insertion



Algorithms

Case I – As a first Node

```
Clist_insert_first (Head)
Begin
    nnode =new NODE
    nnode=newvalue     // newvalue is the new info
    nnode->next=NULL
    If (Head=NULL) Then
          Head=nnode
          Head->next=Head
    Else
        temp=Head
        while (temp->next !=Head)
              temp=temp->next
        nnode->next=head
        temp->next=nnode
        head=node
  endif
END
```

Case II – Insert as a Last Node

```
Clist_insert_last (Head)
Begin
    nnode =new NODE
    nnode=newvalue
    nnode->next=NULL
    If (Head=NULL) Then
          Head=nnode
          Head->next=Head
   Else
             temp=Head
        while (temp->next !=Head)
                temp=temp->next
        nnode->next=temp->next
        temp->next=nnode
    endif
END
```

Node Deletion from Circular Singly Linked List

Case I – Delete the first node

```
Cllist_delete_first( Head)
  Begin
        If ( Head->next = Head)
           temp=Head
           Head=NULL
            Delete(temp)
        Else
           temp=Head
```

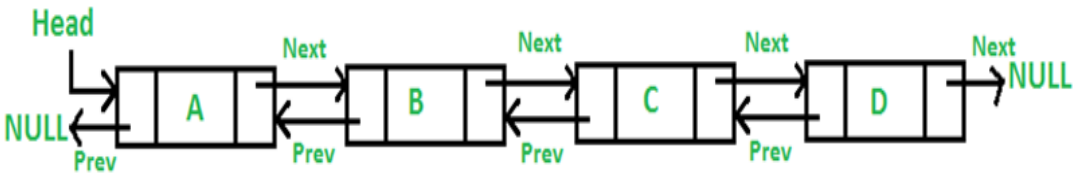
```
        while (temp->next !=Head)
                temp=temp->next
          temp->next = Head->next
           t1=Head
           Head=Head->next
           Delete(t1)
       Endif
END
```

Case -II- Deleting the last node

```
Cllist_delete_first( Head)
  Begin
       If ( Head->next = Head)
          temp=Head
          Head=NULL
          Delete(temp)
      Else
           temp=Head
          while (temp->next !=Head)
                prev=temp
                temp=temp->next
           prev->next=temp->next
           temp->next=NULL
           delete(temp)
       Endif
END
```

Doubly Linked List (Two Way List)

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



Next − Each link of a linked list contains a link to the next node called Next.

Prev − Each link of a linked list contains a link to the previous node called Prev.

Doubly Linked List Node Insertion Algorithms

Case I : As a First Node

```
Dl_insertion_first(head, newvalue)
Begin
    newnode=new NODE // Dynamically create a new node for insertion
                          // NODE – user defined data type for representing a node
    newnode->info=newvalue
    newnode->left=NULL
    newnode->right=NULL
    IF (head=NULL) THEN
        head=newnode
    ELSE
      Begin
          newnode->right=head
          head->left=newnode
          head=newnode
```

```
      end
end
```

Case II : As a Last Node

```
Dl_insertion_last(head,newvalue)
Begin
    newnode=new NODE
    newnode->info=newvalue
    newnode->left=NULL
    newnode->right=NULL
IF(head=NULL) Then
    head=newnode
Else
   Begin
      temp=head
       While(temp->right!=NULL) Then
              temp=temp->right
      temp->right=newnode
      newnode->left=temp
     end
End
```

Case III : Insertion at a Specified Position

```
Dl_insertion_between(head,newvalue,pos)// pos-Position where new node is
```

```
                                                    to be inserted
Begin
   newnode=new NODE
    newnode->info=newvalue
    newnode->right=NULL
    newnode->left=NULL
IF(head=NULL) Then
    head=newnode
Else
   Begin
      Temp=head
       i=1
       While(i<pos)
         Begin
             prev=temp
              temp=temp->right
              i=i+1;
         end
      newnode->right=temp
      temp->left=newnode
      prev->right=newnode
      newnode->left=prev
  end
End
```

Node Deletion in Doubly Linked List

Case I : Delete the Head Node

```
Sl-deletion-first(head)
Begin
     If (head->right=NULL)
        temp=head
        head=NULL
    Else
       temp=head
      head=head->right
      head->left=NULL
      temp-right=NULL
  delete(temp)
End
```

Case II: Delete the last Node

```
Sl_deletion_last(head)
 Begin
     If (head->right=NULL)
        temp=head
        head=NULL
     Else
       Begin
       temp=head
       While(temp->right!=NULL)
            prev=temp
            temp=temp->right
       prev->right=NULL
      temp->left=NULL
  delete(temp)
```

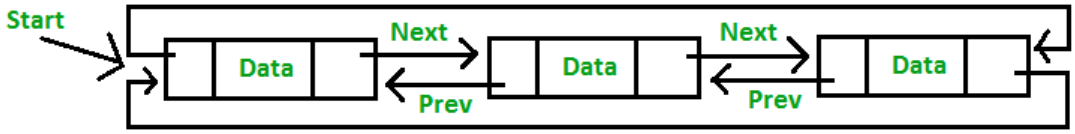
End

Case III : Delete the node in a specified Position

```
Sl_deletion_between(head, pos)
 Begin
     If (head->right=NULL)
        temp=head
        head=NULL
   Else
      Begin
       i=1, temp=head
       While(i<pos)
           prev=temp
           temp=temp->right
           i=i+1
      prev->right=temp->right
      (temp->right)->left=prev
      temp->right=NULL
      temp->left=NULL
      end
  delete(temp)
End
```

Circular Doubly Linked List

Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by previous and next pointer and the last node points to first node by next pointer and also the first node points to last node by previous pointer.



Basic Memory Management Concepts

There are two types of memories first is the logical memory and second is the physical memory. The memory which is temporary such as RAM is also known as the temporary memory, and the memory which is permanent such as the hard disk is also known as the physical memory of the system.

When we want to execute any programs then that programs must be brought from the physical memory into the logical memory. So that we use the concept of memory management.

The Operating system translates the physical address into the logical address, and if he wants to operate, then he must translate the physical address into the logical address. This is the also known as binding. Means when a physical address is mapped or convert into the logical address, and then this is called as the binding.

Contiguous allocation

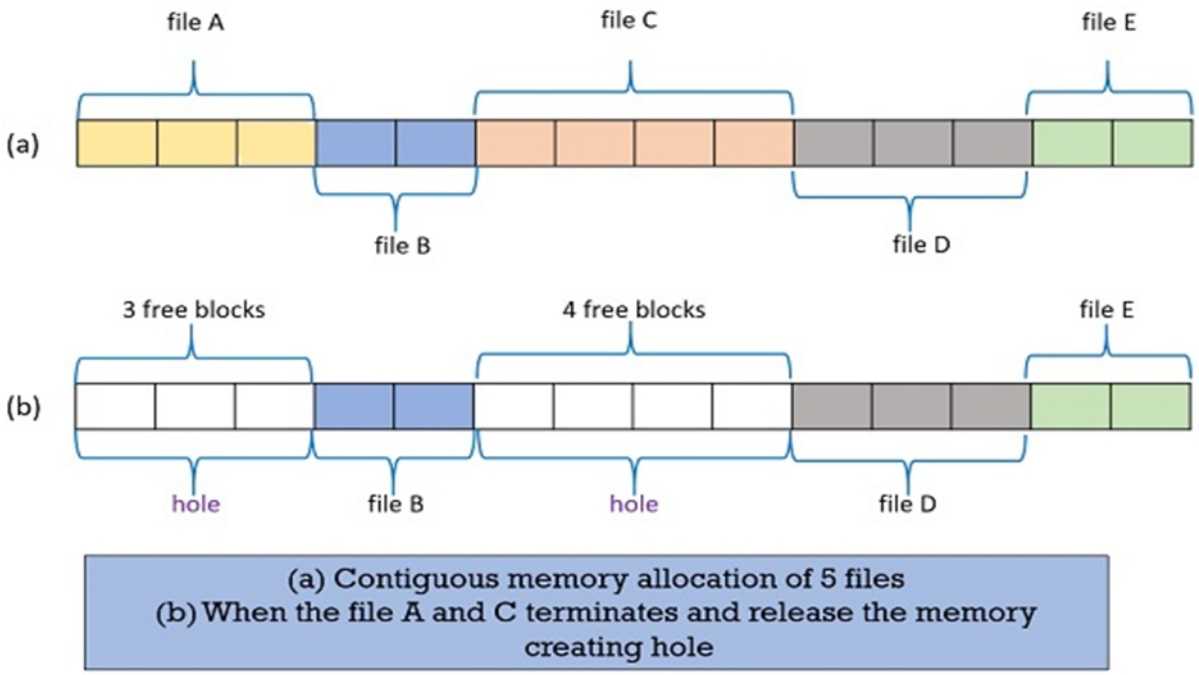
The contiguous allocation means the memory spaces are divided into the small and equal size and the different process uses the various partitions for running their applications process. Moreover, when a request has found, then the process will allocate the space. Moreover, in this, the adjacent spaces are provided to every process. Means all the process will reside in the memory of the computer and when a process will request for the memory then this available memory or free memory will be allotted to him.

However, there will be a problem when the memory which is required by the process is not enough for him or when the size of memory is less which is required for the process. So, this problem is also known as the internal fragmentation. The main reason for the internal fragmentation is, all the memory is divided into the fixed and continuous sizes. So that if a process requires large memory, then that process will not be fit into the small area.

The second problem also occurs in the continues allocation. This is also known as the external fragmentation. In this when the memory is not enough after combing the multiple

parts of a single memory. In this when the required memory is high after combining the various areas of memory, then this is called external fragmentation. So, there are the following problems arise when we use the contiguous memory allocation.

1) Wasted Memory: the wasted memory is that memory which is unused and which can't be given to the process. When the various process comes which require memory which is not available this is called as the wasted memory.
2) Time Complexity: there is also wastage of time for allocating and de-allocating the memory spaces to the process.
3) Memory Access: there must be some operations those are performed for providing the memory to the processes.

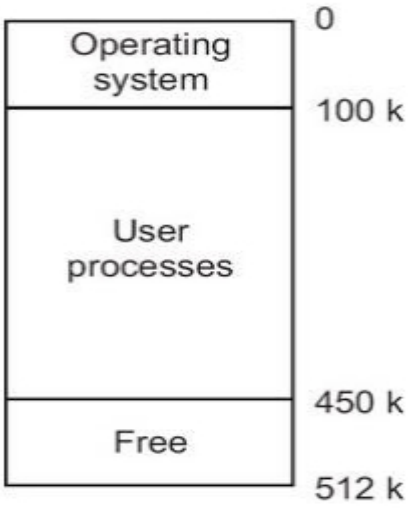


(a) Contiguous memory allocation of 5 files
(b) When the file A and C terminates and release the memory creating hole

Partition Allocation: when there is more than one partition freely available to accommodate a process's request, a partition must be selected. To choose a particular partition, a partition allocation method is needed. A partition allocation method is considered better if it avoids internal fragmentation.
When it is time to load a process into main memory and if there is more than one free block of memory of sufficient size then the OS decide which free block to allocate.

Single Partition Allocation

In this scheme Operating system is residing in low memory and user processes are executing in higher memory.



Advantages

It is simple.

It is easy to understand and use.

Disadvantages

It leads to poor utilization of processor and memory.

Users job is limited to the size of available memory.

Multiple-partition Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed sized partitions. There are two variations of this.

Fixed Equal-size Partitions

It divides the main memory into equal number of fixed sized partitions, operating system occupies some fixed portion and remaining portion of main memory is available for user processes.

Advantages

Any process whose size is less than or equal to the partition size can be loaded into any available partition.

It supports multiprogramming.

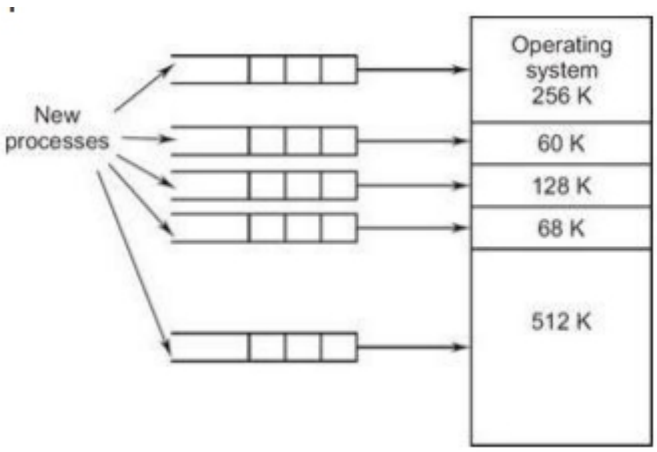
Disadvantages

If a program is too big to fit into a partition use overlay technique.

Memory use is inefficient, i.e., block of data loaded into memory may be smaller than the partition. It is known as internal fragmentation.

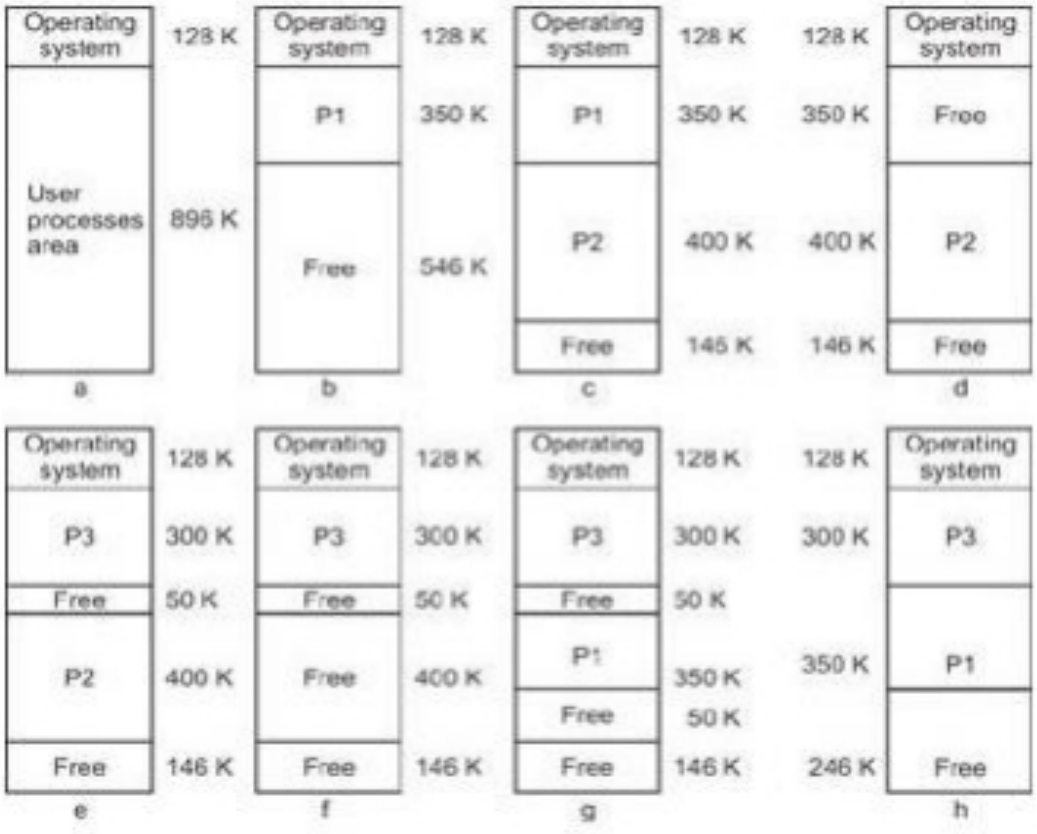
Fixed Variable Size Partitions

By using fixed variable size partitions we can overcome the disadvantages present in fixed equal size partitioning.



Dynamic Partitioning

Even though when we overcome some of the difficulties in variable sized fixed partitioning, dynamic partitioning require more sophisticated memory management techniques. The partitions used are of variable length. That is when a process is brought into main memory, it allocates exactly as much memory as it requires. Each partition may contain exactly one process. Thus the degree of multiprogramming is bound by the number of partitions. In this method when a partition is free a process is selected from the input queue and is loaded into the free partition. When the process terminates the partition becomes available for another process.



Memory allocation (partition allocation) Strategies

Best-fit:- It chooses the block, that is closest in size to the given request from the beginning to the ending free blocks. This strategy produces the smallest leftover hole.

First-fit:- It begins to scan memory from the beginning and chooses the first available block which is large enough. Searching can start either at the beginning of the set of blocks or where the previous first-fit search ended. We can stop searching as soon as we find a free block that is large enough.

Worst-fit:- It allocates the largest block. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

last-fit:- It begins to scan memory from the location of the last placement and chooses the next available block. may be required more frequently with next-fit algorithm. Best-fit is the worst performer, even though it is to minimize the wastage space. Because it consumes the lot of processor time for searching the block which is close to its size

Garbage Collection

In this method, nodes no longer in use remain allocated and undetected until all available storage has been allocated. A subsequent request for allocation cannot be satisfied until nodes that had been allocated but are no longer in use are removed.

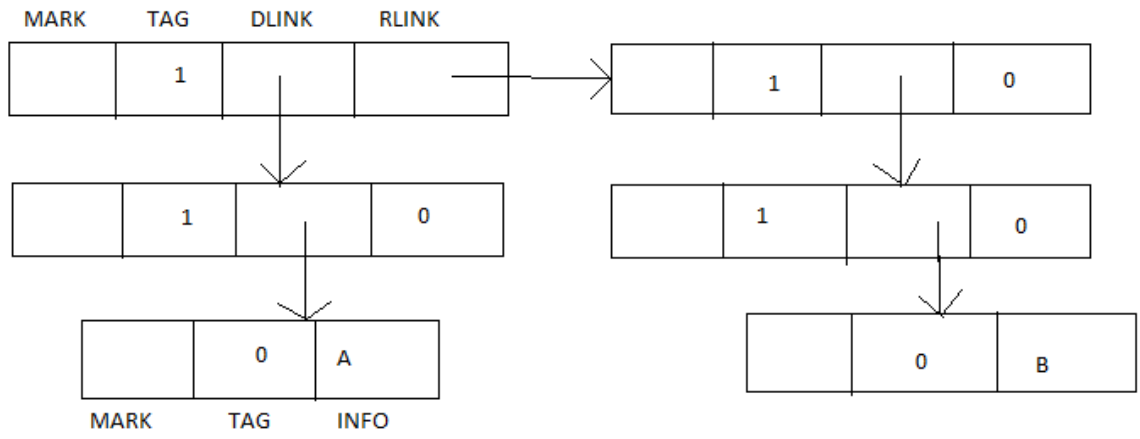
That is garbage collection is the process of collecting all unused nodes and returning them to available space. This process is carried out in essentially two phases.

In the first phase, known as the marking phase, all nodes in use are marked.

In the second phase all unmarked nodes are returned to the available space

In the second Phase, when variable size nodes are in use, it is desirable to compact so that all free nodes from a continuous block of memory. In this case the second phase is referred to as memory compaction.

Marking



In this case, we need a MARK bit in each node. The marking algorithm mark all directly accessible nodes (nodes accessible through pointer variable) and also all indirectly accessible nodes (nodes accessible through link field of nodes).

Each node regardless of its usage will have a bit field mark field, MARK, as well as a one-bit tag field, TAG. The tag bit of a node will be zero if it contains atomic information (**atomic nodes**). The tag bit is one otherwise. A node with tag of one has two link fields DLINK & RLINK is called **list nodes**.

For performing the marking algorithm, it is required to unmark all nodes (Initial condition)

That is MARK(i)=0 for all nodes i.

The marking procedure is a two-stage algorithm:

1. DRIVER Algorithm
2. MARK1 Algorithm

```
Function DRIVER()
Begin
  FOR  i-1 to n do // Assume that n nodes are available
       MARK(i)=0
  FOR each pointer variable pointer X with MARK(X)=0 do
      MARK(X)=1
      IF TAG(X)=1 THEN CALL MARK1(X)
  END
Function MARK1(X)
Begin
   Initialize the stack
   P=X
   LOOP
      LOOP
          Q=RLINK(P)
          IF TAG(Q)=1 AND MARK(Q)=0 THEN
                    CALL PUSH(Q)
          MARK(Q)=1
          P=DLINK(P)
          IF MARK(P)=1 OR TAG(P)=0 THEN  EXIT
          MARK(P)=1
       FOREVER
    IF stack  is empty then RETURN
    CALL POP(P)
 FOREVER
END
```

This marking algorithm MARK1(X) will start from the list node X and mark all nodes can be reached from X via a sequence of RLINK's and DLINK's. While examining any node, we will have a choice as to whether to move to the DLINK or RLINK. The MARK1 will

move to the DLINK but will at the same time place the RLINK on a stack in case that RLINK is a list node not yet marked. The use of this stack will enable us to return at a later pointer to the RLINK and examine all paths from there.