SINGLY LINKED LIST

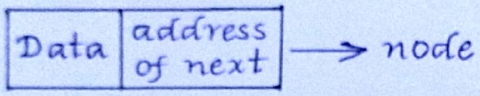
Definition:

A singly linked list is a linear dynamic data structure made up of nodes, where:

→ Each node contains:

* Data
* One pointer (next) that stores the address of the next node
* The last node points to NULL.

It is called a singly linked list because each node has only one link.



Structure of a node:

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

Memory representation of a singly linked list:

A singly linked list is represented using the linear Arrays in memory where each memory address stores two components.

1) Data value
2) The memory address of the next element.

→ This memory representation shows that the values need not be stored contiguously.

→ 100 points to memory address '20', where 200 is stored hence, 200 is to the next of 100.

→ Similarly, 200 next has a memory address of 30, where 300 is stored.

→ Similarly, 400 has its next as NULL, meaning it's the last node of this singly linked list.

→ It forms the singly linked list like

100 → 200 → 300 → 400 → NULL

	Address	Value	Next
head ←	10	100	20
	20	200	30
	30	300	40
	40	400	NULL

Characteristics of S.L.L:

1) Each node holds:

* one data value
* one reference to nextnode

2) The list has a head pointer:

* points to the 1st node
* used to access entire list

3) Sometimes we maintain a tail pointer:

* Helps in fast insertion at end
* Useful in Queue implementation.

4) Nodes are not stored in contiguous memory

* stored any where in heap memory
* Connected using pointers.

5) No direct access to elements

* Must traverse from head.

Insertion Operations

1. Insert at head position:

→ Adding a newnode at the beginning of the linked list.
→ The newnode becomes the first node.
→ The head is updated to newnode

Logic:

To insert at head:

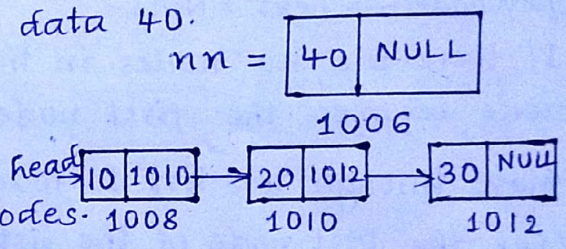
1. Create a newnode
2. Store data inside newnode
3. Connect newnode to a head
4. Then shift head to newnode

Algorithm for insert node at head:

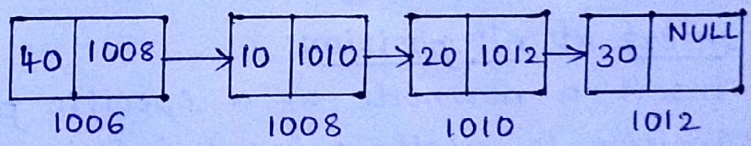
1. Create a new node using malloc(size of(struct Node));
   Newnode→data = given_int_value
   Newnode→next = NULL
2. If the linked list is empty(head==NULL), return the newnode as head.
3. Connect the next pointer of this newnode to the current head (Newnode→next=head)
4. Update the head pointer to point to the newnode (head=newnode)
5. Return the newhead

Tracing of an example:

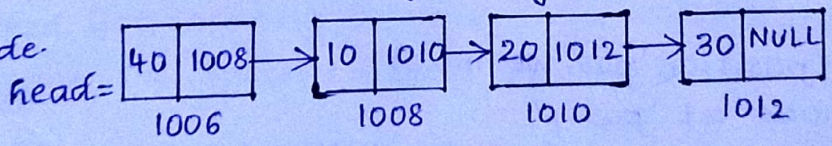
1. Allocate memory for a newnode with data 40.
   nn→data = 40
   nn→next = NULL



2. Check whether the list contains any nodes.
3. if head==NULL then assign head as newnode.
4. else link the newnode to the current first node of list.
5. Update head pointer so the newnode becomes first node.



6. The newnode is inserted at the beginning of the linked list & become the new head node.



2. Insert at tail position:

→ Adding a new node at the end of the linked list.
→ The newnode becomes the last node
→ Its next pointer will be NULL

Logic:

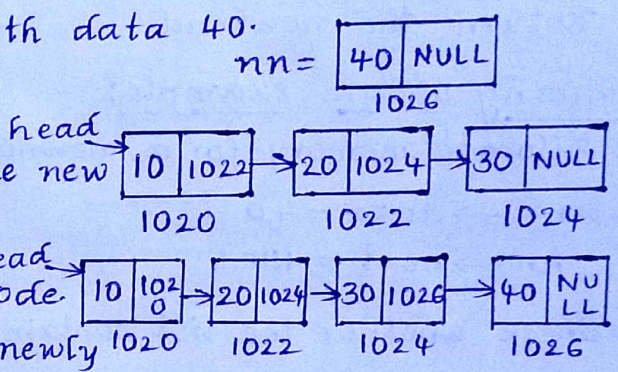
Unlike insert at head:
→ We cannot directly insert
→ we must traverse till the last node
because,
→ we need to find the node whose next is NULL
→ And then connect the newnode to it.

Algorithm for insert node at tail:

1. Create a newnode using malloc(sizeof(struct Node));
   newnode → data = given_int_value
   newnode → next = NULL
2. If the (head == NULL), return the newnode as head.
3. Traverse to the last node using a tempoyary point until
   current → next == NULL.
4. Connect last node's next pointer to the newnode, (current → next = newnode)
5. Return the original head

Tracing of an example:

1. Allocate memory for a new node with data 40.
   newnode → data = 40
   newnode → next = NULL
2. If there are no nodes in the list, the new node becomes the first node.
3. Move through the list to reach last node.
4. Link the last node of the list to the newly created node.
5. The head pointer remains the same because insertion is done at the end.



3. Insert at nth position:

→ Insert a newnode at a specific position (index) in the linked list.
→ Position is usally 0-based indexing.

Logic:

To insert at position 'pos', we must:
→ Move to node at 'pos-1'.
→ Store the address of node at position 'pos'
→ Adjust links carefully.

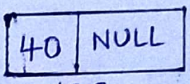
Algorithm for insert node at position:

1. Create a newnode using malloc(sizeof(struct Node));
2. If newnode == NULL, print "Memory allocation failed". Exit.
3. Newnode → data = given_int_value
4. Newnode → next = NULL.
5. If pos < 0, print "Invalid position", Return head.
6. if pos == 0
   newnode → next = head
   head = newnode, return head.

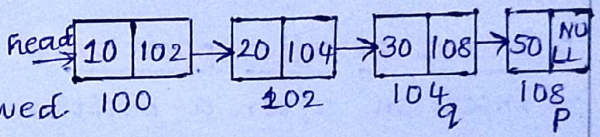
7. Set pointer p=head
8. for i=0 to pos-1
   if p==NULL, print "position exceeds list length", return head
   move p=p→next
9. set q=p→next
   p→next=newnode
   newnode→next=q
10. Return head

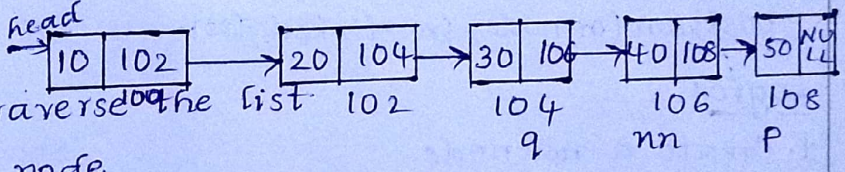
Tracing of an example:

pos=4.

1. Allocate memory for newnode with the value 40. nn= 

2. If newnode==NULL, memory not available.
3. Store the given value in the newnode.
4. if pos<0, negative position is not allowed
5. if pos==0, insert at head.
6. Temporary pointer p to traverse the list
7. Link newnode to the next node
8. Head remains same except when pos=0.





4. Insert at before position:

→ In a singly linked list, inserting before a given position means adding a newnode just before the specified node number in the list.

Logic:

To insert before a position, we must
1. Create a newnode
2. Move to the node before the target position.
3. Change links so the newnode connects properly.

* We must stop at 'pos-2' node to insert before position.

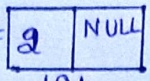
Algorithm for insert node at before position:

1. Create a newnode using malloc(sizeof(struct Node));
2. if pos==0, insert at head; newnode→next=head, return newnode.
3. Traverse to (pos-1) the node immediately before using temporary pointer P
4. set q=p→next

5. Insert before pos: p→next=newnode; newnode→next=q.

6. Return the head.

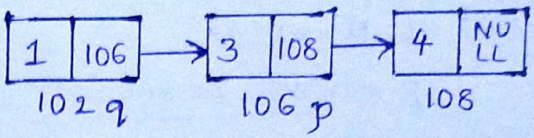
Tracing of an example:

1. Allocate memory for newnode nn= 

pos=2

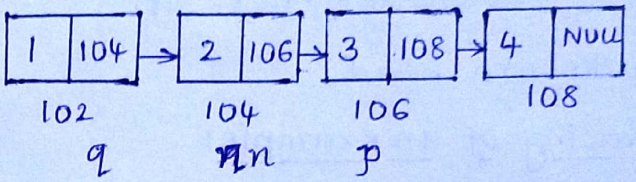
2. if pos==0, insertion at beginning of list.

3. Locate the node immediately before the insertion point.



4. q points to node at (pos-1)th position.

5. Link previous node to newnode, & newnode to currentnode at pos.



6. Return the start to the list

5. Insert at after position:

→ Insertion after a position means adding a newnode after a specified position (or node) in linked list.

Logic:

1. Create a newnode
2. Store the data in the newnode
3. Traverse the list until the given position.
4. Adjust the pointers:

→ newnode should point to the next of the current node

→ current node should point to the new node.

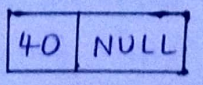
pointer logic: if

- temp → node at given position.
- newnode → next = newnode

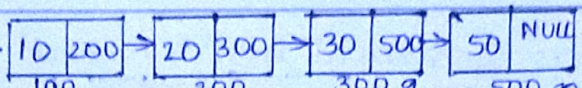
Algorithm for insert node at after position:

1. Create a newnode using malloc(sizeof(struct Node));
2. if pos==0, insert after head, head→next=newnode, return head.
3. Traverse to 'pos' node using temporary pointer p.
4. set q=p→next
5. Insert after p=p→next=newnode, newnode→next=q
6. Return the head.

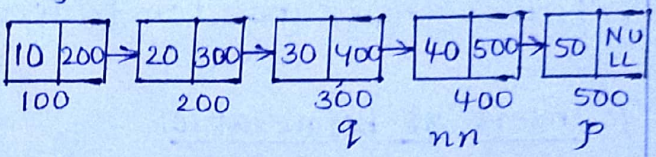
Tracing of an example:

1. Allocate memory for newnode nn= 

2. if pos==0, insert after head.



3. Iterate with pointer 'p' to reach the node before the desired insertion position.
4. Temporarily store the pointer to the node which come after the node.
5. Link the newnode into list between node p & q.
6. Return the head pointer to the list.



6. Insert at after value:

→ Insertion after a value means adding a newnode immediately after the node that contains a specified value in the linked list.

Logic:

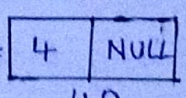
1. Start from the head node.
2. Move through the list until the given value is found.
3. Create a newnode.
4. Store the newdata in the node.
5. Set newnode→next = temp→next
6. Set temp→next == newnode. This connects the newnode after the required value

Algorithm for insert node after value:

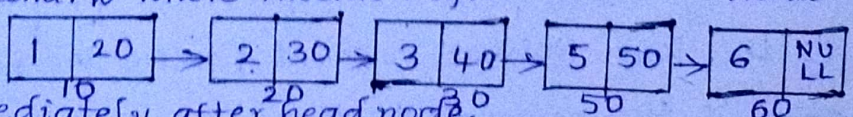
1. Create a newnode using malloc(sizeof(struct Node));
2. if list empty, return NULL.
3. if head→data == key, insert after head; head→next=newnode, return head
4. Traverse with two pointers.
5. Insert after key node: p→next = newnode, newnode→next=p→next
6. if key not found, do nothing or insert at tail.
7. Return head.

Tracing of an example:

1. Allocate the memory to newnode with value 4. nn=

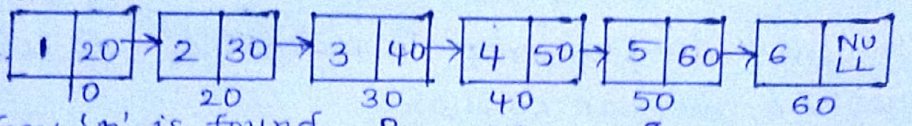


2. Handle an empty list scenario where insertion after the key is impossible.



3. Handle the insertion immediately after head node.

4. Set up pointers for normal list traversal to find insertion point



5. Perform insertion once key 'p' is found. p nn. q

6. Handle scenario where the specified key is not present in the list.

7. Return head pointer to list.

7. Insert at beforevalue:

→ The operation means adding a newnode before a node that contains a specific value in a singly linked list.

Logic:

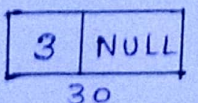
Since a singly linked list only has a pointer to the next node, we can't directly move backwards so to insert before value, we must:

1. Traverse the list from the head
2. Find the node whose next node contains the given value
3. Create a newnode.
4. Adjust the links so that the newnode comes before the target value.

Algorithm for insert node before value:

1. Create a newnode using malloc(sizeof(struct Node));
2. if list empty or head → data == key, insert at head: newnode → next = head return newnode.
3. Traverse with two pointers: p(current) & q(previous)
4. Insert before key node, q → next = newnode, newnode → next = p.
5. if key not found, insert at tail.
6. Return the head.

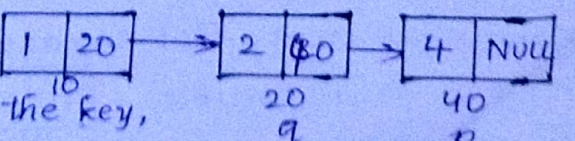
Tracing of an example:

1. Allocate memory to newnode 

= nn

value = 4.

2. Handling insertion into an empty list.
3. Inserting newnode between q & p.



4. if the loop finishes without finding the key, append the newnode to the end.

5. Ensure the function returns the correct starting point of list