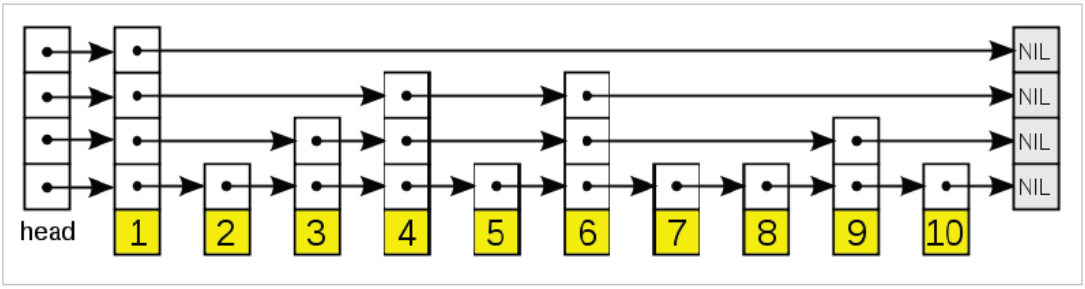
10. What is skip list? Explain its working principle.

A skip list is a data structure that is used for storing a sorted list of items with a help of hierarchy of linked lists that connect increasingly sparse subsequences of the items. A skip list allows the process of item look up in efficient manner. The skip list data structure skips over many of the items of the full list in one step, that's why it is known as skip list.



In short, skip lists are a linked list like structure which allows for fast search. It consists of a base list holding the elements, together with a tower of lists maintaining a linked hierarchy of subsequences, each skipping over fewer elements.

Description

A skip list is built in layers:

- The bottom layer is an ordinary ordered linked list.
- Each higher layer acts as an "express lane" for the lists below, where an element in layer i appears in layer i+1 with some fixed probability p (two commonly-used values for p are 1/2 or 1/4).

On average, each element appears in 1/(1-p) lists, and the tallest element (usually a special head element at the front of the skip list) appears log1/pn in lists.

A search for a target element begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target.

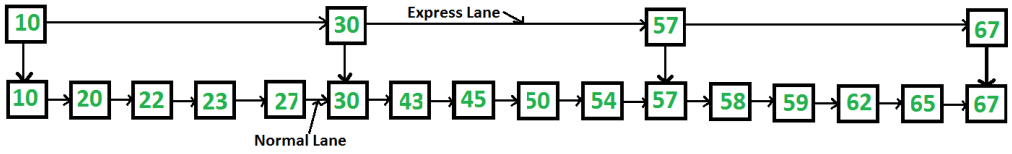
If the current element is equal to the target, it has been found.

If the current element is greater than the target, the procedure is repeated after returning to the previous element and log1/pn/p which is 0(logn) where p is a constant. By choosing different values of p, it is possible to trade search costs against storage costs.

Working

We create multiple layers so that we can skip some nodes.

See the following example list with 16 nodes and two layers. The upper layer works as an express lane which connects only main outer stations, and the lower layer works as a normal lane which connects every station. Suppose we want to search for 50, we start from first node of "express lane" and keep moving on "express lane" till we find a node whose next is greater than 50. Once we find such a node (30 is the node in following example) on "express lane", we move to "normal lane" using pointer from this node, and linearly search for 50 on "normal lane". In following example, we start from 30 on "normal lane" and with linear search, we find 50.



11. Illustrate the basic operations on Skip-list.

Following are the Operations Performed on Skip list:-

1. Insertion Operation : To Insert any element in a list
2. Search Operation : To Search any element in a list
3. Deletion Operation : To Delete any element from a list

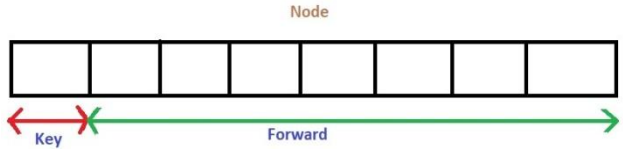
Insertion Operation

Before we start inserting the elements in the skip list we need to *decide the nodes level*. Each element in the list is represented by a node, the level of the node is chosen randomly while insertion in the list. Level does not depend on the number of elements in the node. The level for node is decided by the following *algorithm* –

```
randomLevel()
lvl := 1
//random() that returns a random value in [0...1)
while random() < p and lvl < MaxLevel do
lvl := lvl + 1
return lvl
```

MaxLevel is the upper bound on number of levels in the skip list. It can be determined as – $L(N) = \log_{p/2}(N)$. Above algorithm assure that random level will never be greater than MaxLevel. Here p is the fraction of the nodes with level i pointers also having level i+1 pointers and N is the number of nodes in the list.

Node Structure :- Each node carries a key and a forward array carrying pointers to nodes of a different level. A level i node carries i forward pointers indexed through 0 to i.



We will start from highest level in the list and compare key of next node of the current node with the key to be inserted. Basic idea is:

1. Key of next node is less than key to be inserted then we keep on moving forward on the same level
2. Key of next node is greater than the key to be inserted then we store the pointer to current node i at update[i] and move one level down and continue our search.

Pseudocode For Insertion

```
Insert(list, searchKey)
local update[0...MaxLevel+1]
x := list -> header
for i := list -> level downto 0 do
    while x -> forward[i] -> key  forward[i]
update[i] := x
x := x -> forward[0]
lvl := randomLevel()
if lvl > list -> level then
for i := list -> level + 1 to lvl do
    update[i] := list -> header
    list -> level := lvl
x := makeNode(lvl, searchKey, value)
```

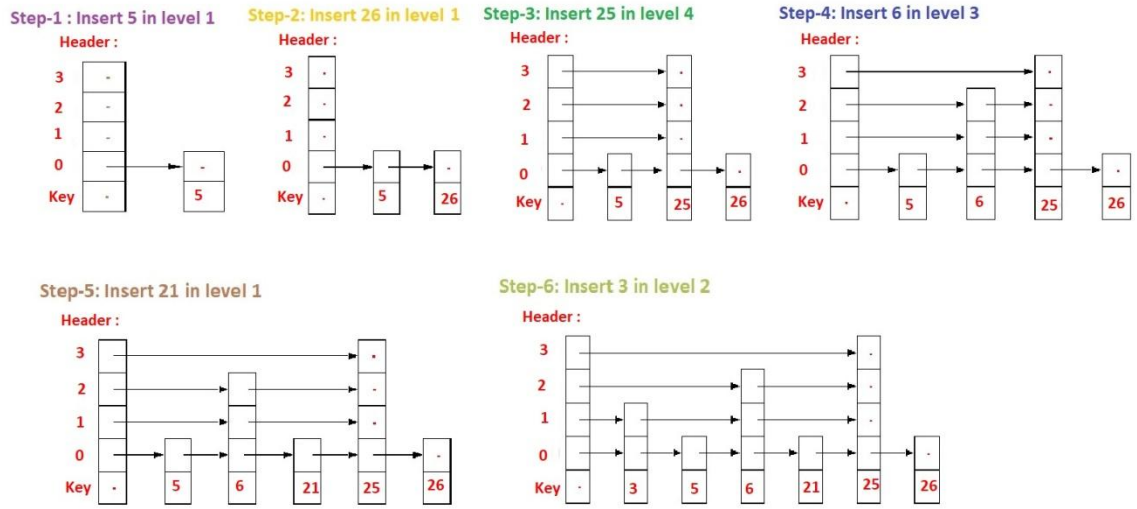
```
for i := 0 to level do
    x -> forward[i] := update[i] -> forward[i]
update[i] -> forward[i] := x
```

- Here update[i] holds the pointer to node at level i from which we moved down to level i-1 and pointer of node left to insertion position at level 0.

Example

Starting with an empty Skip list with MAXLEVEL 4, Suppose we want to insert these following keys with their "Randomly Generated Levels":

5 with level 1, 26 with level 1, 25 with level 4, 6 with level 3, 21 with level 1, 3 with level 2, 22 with level 2 .



Now, at Step-7 where we need to insert 22 in level 2 we need to go through some following details:

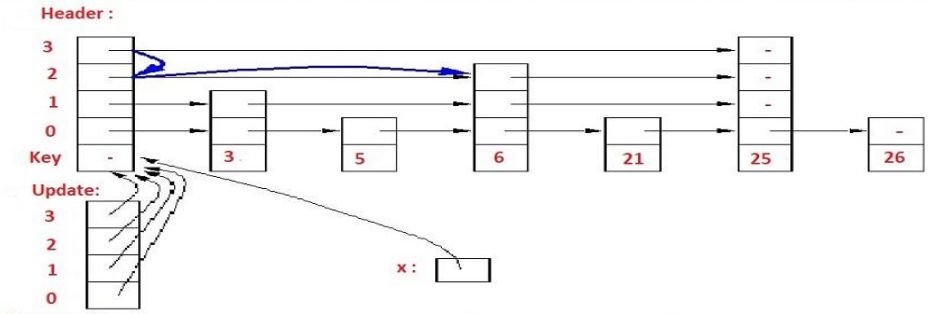
The insert algorithm maintains two local variables(besides the skip list header):

- X, a pointer which points to a node whose forward pointers point to nodes whose key we are currently comparing to the key we want to insert this lets us quickly compare keys, and follow forward pointers
- update, an array of node pointers which point to nodes whose forward pointers may need to be updated to point to the newly inserted node , if the new node is inserted in the list just before the node X points to this lets us quickly update all the pointers necessary to splice in the new node

6 keys have been inserted, and see how key 22 is inserted in a new node with level 2

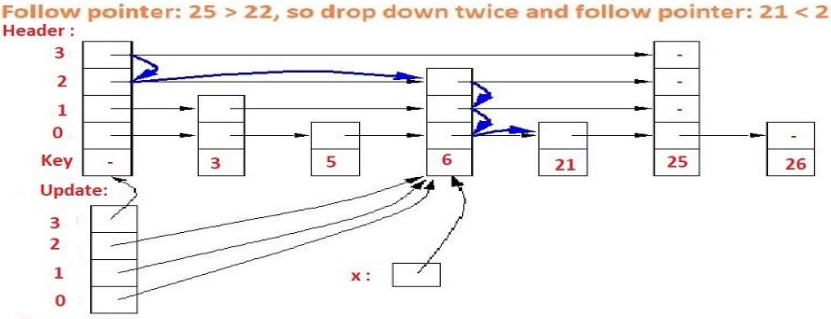
Inserting 22 in level 2: frame 1

Follow top-level pointer: 25 > 22, so drop down and follow pointer: 6 < 22, so update



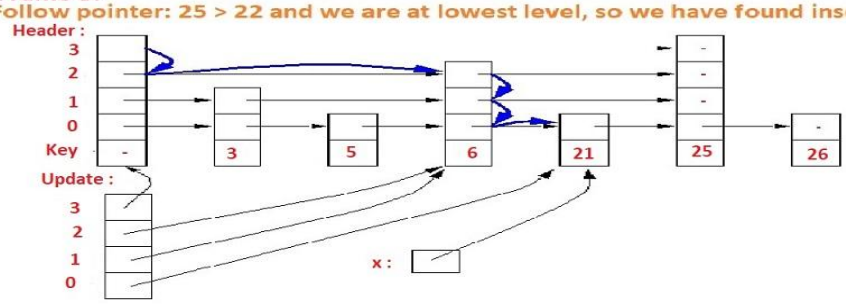
Frame 2:

Follow pointer: 25 > 22, so drop down twice and follow pointer: 21 < 22, so update

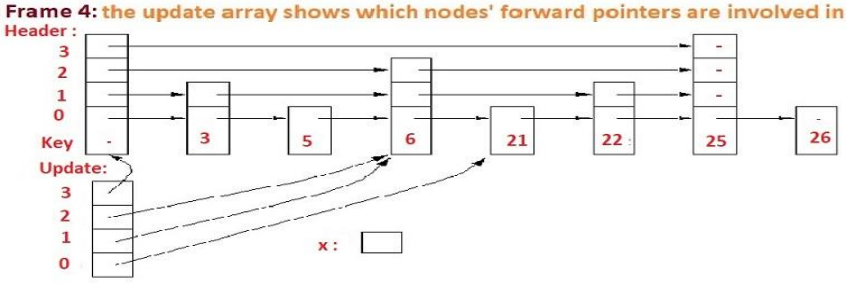


Frame 3:

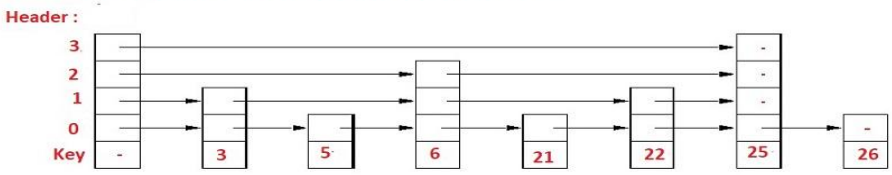
Follow pointer: 25 > 22 and we are at lowest level, so we have found insert point



Frame 4: the update array shows which nodes' forward pointers are involved in the insert



Here is the skip list after all keys have been inserted :



Searching Operation

Searching an element is very similar to approach for searching a spot for inserting an element in Skip list. The basic idea is if –

1. Key of next node is less than search key then we keep on moving forward on the same level.
2. Key of next node is greater than the key to be inserted then we store the pointer to current node i at update[i] and move one level down and continue our search.

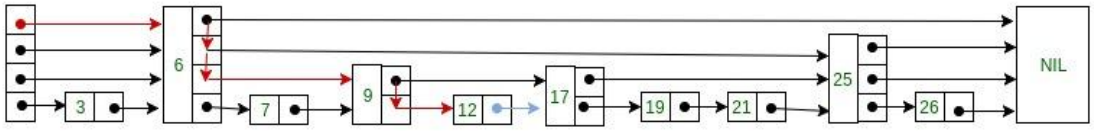
- At the lowest level (0), if the element next to the rightmost element (update[0]) has key equal to the search key, then we have found key otherwise failure.

Pseudocode For Searching:

```
Search(list, searchKey)
    x := list -> header
    -- loop invariant: x -> key  level downto 0 do
        while x -> forward[i] -> key  forward[i]
    x := x -> forward[0]
    if x -> key = searchKey then return x -> value
    else return failure
```

Example of Searching

Consider this example where we want to search for key 17- Now to search for key in a skip list we will follow the steps of our Pseudocode. Here, the idea is that we will compare the key values of every node with our search key (ie 17). if the key of next node is greater than our key 17 then we keep on moving on that same level otherwise we store the pointer to current node i at update[i] and move one level down and continue our search. Here, we will stop at which the key of next node is 19 (ie 17 < 19) and store pointer of that node.



Deletion Operation

Deletion of an element k is preceded by locating element in the Skip list using above mentioned search algorithm. Once the element is located, rearrangement of pointers is done to remove element from list just like we do in singly linked list. We start from lowest level and do rearrangement until element next to update[i] is not k. After deletion of element there could be levels with no elements, so we will remove these levels as well by decrementing the level of Skip list.

Pseudocode For Deletion

```
Delete(list, searchKey)
local update[0..MaxLevel+1]
x := list -> header
for i := list -> level downto 0 do
  while x -> forward[i] -> key  forward[i]
    update[i] := x
x := x -> forward[0]
if x -> key = searchKey then
  for i := 0 to list -> level do
      if update[i] -> forward[i] ≠ x then break
```

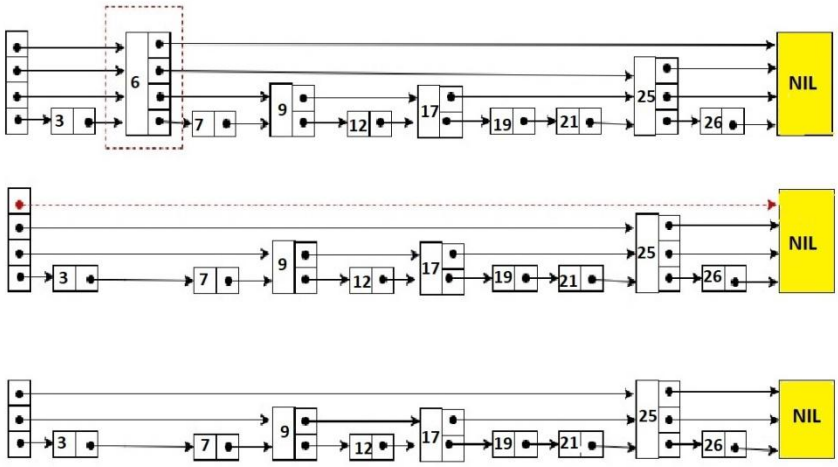
```
    update[i] -> forward[i] := x -> forward[i]
  free(x)
  while list -> level > 0 and list -> header -> forward[list -> level] = NIL do
    list -> level := list -> level – 1
```

Example of deletion

Consider this example where we want to delete element 6 –

Deletion of an element 6 is preceded by locating this element 6 in the Skip list using above mentioned search algorithm. Once 6 is located, rearrangement of pointers is done to remove 6 from list just like we do in singly linked list.

First we must find the preceding node ie 3 and identify the nodes whose pointers may need to be reset. Next we must reset the pointers to the targeted node "around" it (ie 7), and finally we deallocate the targeted node ie 3.



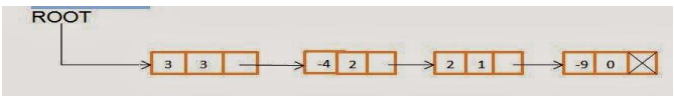
- Here at level 3, there is no element (arrow in red) after deleting element 6. So we will decrement level of skip list by 1.

12. How polynomial can be represented using linked list?

The linked list can be used to represent a polynomial of any degree. Simply the information field is changed according to the number of variables used in the polynomial. If a single variable is used in the polynomial the information field of the node contains two parts: one for coefficient of variable and the other for degree of variable. Let us consider an example to represent a polynomial using linked list as follows:

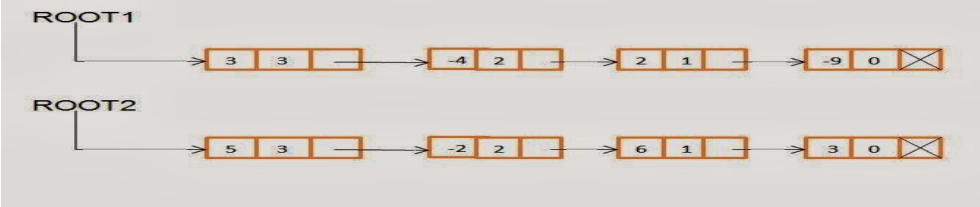
Polynomial: $3x^3-4x^2+2x-9$

Linked List:

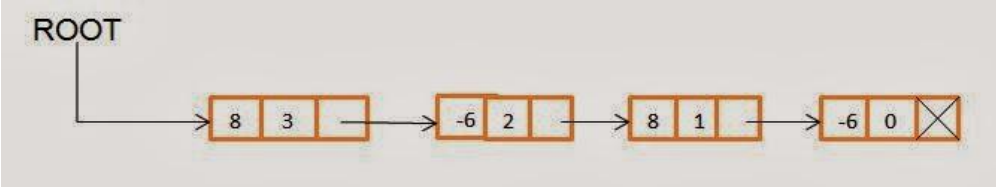


In the above linked list, the external pointer ‘ROOT’ point to the first node of the linked list. The first node of the linked list contains the information about the variable with the highest degree. The first node points to the next node with next lowest degree of the variable.

Representation of a polynomial using the linked list is beneficial when the operations on the polynomial like addition and subtractions are performed. The resulting polynomial can also be traversed very easily to display the polynomial.



The above two linked lists represent the polynomials, $3x^3-4x^2+2x-9$ and $5x^3-2x^2+6x+3$ respectively. If both the polynomials are added then the resulting linked will be:



The linked list pointer ROOT gives the representation for polynomial, $8x^3-6x^2+8x-6$.

C program to read and display a polynomial using Linked List:

```
#inculde<stdio.h>
#inculde<stdlib.h>
#inculde<conio.h>
struct node
{
int coeft;
int degree;
struct node *link;
};
typedef struct node poly;
main()
{
poly *root,*temp,*new;
int hdegree, coeft;
root=NULL;
clrscr();
printf("Enter the highest degree of polynomial:")
scanf("%d",&hdegree);
while(hdegree>=0)
{
printf("Enter coefficient of variable with degree %d",hdegree);
scanf("%d",&coeft);
if(coeft!=0)
```

```
{
new=(poly*)malloc(sizeof(poly));
if(new= =NULL)
{
printf("Memory allocation error…"); exit(0);
}
new->coeft=coeft;
new->degree=hdegree;
new->link=NULL;
if(root= =NULL)
{
root=new;
temp=root;
}
else
{
temp->link=new;
temp=new;
}
}
hdegree--;
}
clrscr();
printf("\n The Polynomial is:\n\n");
temp=root;
while(temp!=NULL)
{
if(temp->coeft>0)
 printf("+%dx%d",temp->coeft,temp->degree);
else
  printf("%dx%d",temp->coeft,temp->degree);
temp=temp->link;
}
 getch();
}
```