Linked List as ADT

Linked list is one of the basic structures used to implement an ADT list.
We can use linked list to create linear and non-linear structures. All the elements in a linked list have either zero, one or more successors.

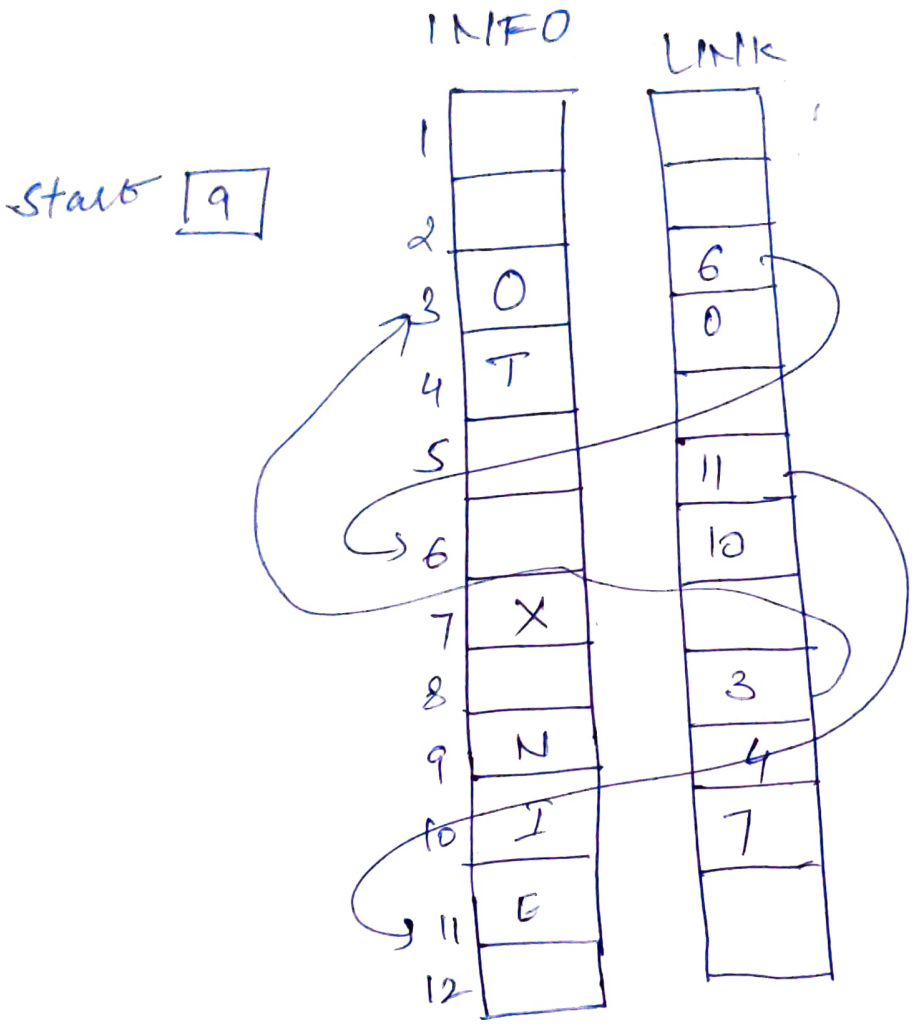
When compared to an array linked lists have a major advantage of easy insertion and deletion of data's.
There is no need for shifting the elements present in the linked list to accomodate a new element or to delete an element.

On the other hand, since there is no physical sequence for the elements we are restricted to use sequential searches instead of using a binary search.

Representation of linked lists in Memory

Let list be a linked list. Then list will be maintained in memory, unless otherwise specified as follows: First of all, list requires two linear arrays — INFO and LINK such that INFO[K] and LINK[K] contain the information part and the next pointer field of a node of list. List also requires a variable named START - which contains the location of the

beginning of the list. and a next pointer sentinel denoted by NULL which indicates the end of the list. Since the subscript of the arrays INFO and LINK will usually be positive.



Traversing a Linked List

Let list be a linked list in memory stored in linear arrays INFO and LINK with START pointing to the first element and NULL indicating the end of LIST.

Suppose we want to traverse LIST in order to process each node exactly once.

Our traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed.

Accordingly, LINK[PTR] points to the next node to be processed.

PTR := LINK[PTR]

Algorithm: Traversing a linked list

1) set Ptr = start [initializes pointer Ptr]
2) Repeat step 3 and 4 while Ptr != NULL
3) Apply Process to INFO[ptr]
4) set Ptr = Link[ptr] [ptr now points to the next node]
   [End of step 2 loop]
5) Exit.

Searching a Linked List

Let LIST be a linked list in memory. If ITEM is actually a key value and we are searching through a file for the record containing ITEM, then ITEM can appear only once in LIST.

① LIST is Sorted

SEARCH (INFO, LINK, START, ITEM, LOC)

LIST is a linked list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST or sets LOC = NULL.

1. Set ptr = START
2. Repeat step 3 while ptr ≠ NULL
3. If ITEM = INFO[ptr], then:
   Set LOC = ptr and Exit

   Else

   Set ptr = LINK[ptr] [ptr now points to the next node]
   [End of If Structure]

   [End of Step 2 loop]
4. [Search is unsuccessful] Set LOC = NULL
5. Exit.

The complexity of this algorithm is same (3) as that of linear search algorithm

The worst case running time is proportional to the number n of element in LIST, and the average case running time is approximately proportional to n/2.

List is Sorted

Suppose the data in LIST are sorted.

SEARCHSORTEDLIST(INFO, LINK, START, ITEM, LOC)

LIST is a sorted list in memory. This algorithm finds the location (LOC) of the node where ITEM first appears in LIST or sets LOC = NULL.

1. Set ptr = START
2. Repeat step 3 while ptr ≠ NULL
3. If ITEM > INFO[ptr] then:
   Set ptr = link[ptr]
   Elseif ITEM = INFO[ptr] then:
   Set LOC = ptr and Exit [Search is Successful]
   Else
   Set LOC = NULL and Exit

[ITEM now exceeds INFO[PTR]]
[End of if structure]
[End of Step 2 loop]

4. Set LOC = NULL
5. exit

* The Complexity of this algorithm is still the same as that of other linear search algorithm.
ie the worst case running time is proportional to the number n of element in LIST, and the average-case running time is approximately proportional to $n/2$.

NOTE:
WITH a sorted array we can apply binary search whose running time is proportional to $\log_2 n$.
on the other hand, a binary search algorithm can not be applied to a sorted linked list since there is no way of indexing the middle element in a list. [Biggest disadvantage]

ALGORITHM : The addition of two polynomial using linked list

This algorithm adds two polynomials. Let ph1, ph2 and ph3 represent the pointers of the three linked list. Each node can contain two integers exp and coff.

Assuming that the two linked lists contain some relevant data about the two polynomials in advance.

we have a function append to insert a new node at the end of the given list.

```
m1 = ph1;
m2 = ph2;
```

malloc is called to create a new node p3 which builds a third list.

```
m3 = ph3;
/* we traverse the lists till one list gets
   exhausted */
while ((m1 != NULL) || (m2 != NULL))
{
  while(m1 -> exp > m2 -> exp)
  {
    m3 -> exp = m1 -> exp;
    m3 -> coff = m1 -> coff;
    append(m3, ph3);
```

```
/* now move to the next term in list 1 */
    m1 = m1 -> next;
}
/* If m2 exponent turns out to be higher, make
p3 same as p2 and append to final list */
while (m1 -> exp < m2 -> exp)
{
    m3 -> exp = m2 -> exp;
    m3 -> coff = m2 -> coff;
    append (m3, ph3);
    m2 = m2 -> next;
}
/* If both exponents are same, we must add
the coefficients to get the term for the final list */
while (m1 -> exp = m2 -> exp)
{   m3 -> exp = m1 -> exp;
    m3 -> coff = m1 -> coff + m2 -> coff;
    append (m3, ph3);
    m1 = m1 -> next;
    m2 = m2 -> next;
}
}
```

/* If list2 get exhausted and if there are terms remaining only in list1, these remaining terms should be appended to the end of list3. However, we do not have to do it terms as p1 is already pointing to remaining terms, we need to append the pointer m1 to ph3 */

```
if (m1 != NULL)
  append (m1, ph3);
  else
   append(m2, ph3);
```