International Islamic University Chittagong
Department of Computer Science & Engineering
Spring - 2021

Course Code: CSE-2321
Course Title: Data Structures

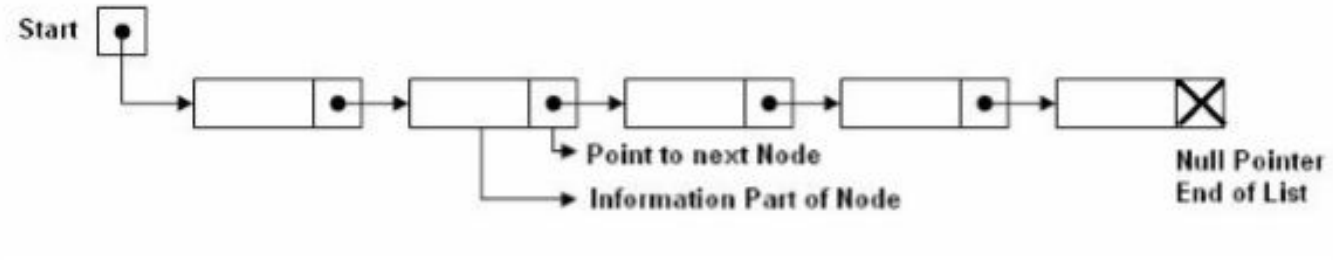
Mohammed Shamsul Alam
Professor, Dept. of CSE, IIUC

Lecture – 10

Linked List

Linked list

- A ***linked list,*** or ***one-way list,*** is a linear collection of data elements, called ***nodes***, where the linear order is maintained by means of ***links*** or ***pointers***.
- In linked list, each node is divided into two parts: the first part contains the information of the element, and the second part, called the *link field* or next *pointer field*, contains the address of the next node in the list.
- The pointer of the **last node** contains a special value, called ***null*** pointer, which is any invalid address. The null pointer, denoted by x in the diagram, signals the end of the list.
- There is a special pointer ***START*** - which contains the address of **first node** in the list.
- **START = NULL** means list has no nodes. Such a list is called the *null list* or *empty list*.
- The following Fig shows a schematic diagram of a linked list with five nodes.



Linked list

- Linked lists were developed in the year **1955-56 by Allen Newell, Cliff Shaw, and Herbert Simon at RAND Corporation** as a primary data structure. It was developed for their Information Processing Language (IPL). IPL was used by the authors to develop several early artificial intelligent programs, including Logic Theory Machine, the General Problem Solver, and a computer chess program.

The drawbacks of using arrays are:

1. Once the elements are stored, it becomes difficult to insert or delete an element at any position in a list.
2. Arrays have fixed size. Hence, if the memory allocated is too large than the actual data, unused portion of the memory is wasted. If the allocated memory is less, it will result in loss of data due to inadequate memory.

The advantage of using linked list is its ability to dynamically shrink and expand in size. This allows you to insert or delete elements efficiently at any position in the list.

Figure 5.4 pictures a linked list in memory where each node ... single character. We can obtain the actual list of characters, or, in other words, the string, as follows:

START = 9, so INFO[9] = N is the first character.
LINK[9] = 3, so INFO[3] = O is the second character.
LINK[3] = 6, so INFO[6] = □ (blank) is the third character.
LINK[6] = 11, so INFO[11] = E is the fourth character.
LINK[11] = 7, so INFO[7] = X is the fifth character.
LINK[7] = 10, so INFO[10] = I is the sixth character.
LINK[10] = 4, so INFO[4] = T is the seventh character.
LINK[4] = 0, the NULL value, so the list has ended.

In other words, NO EXIT is the character string.

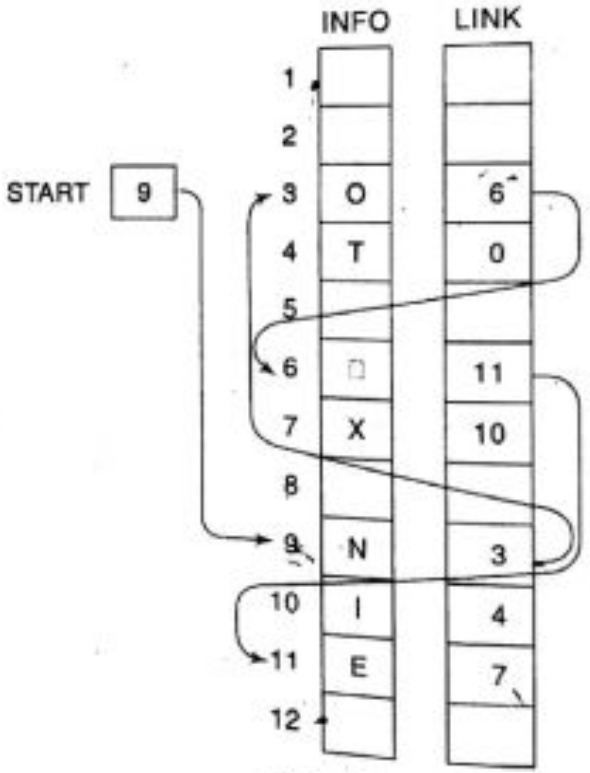
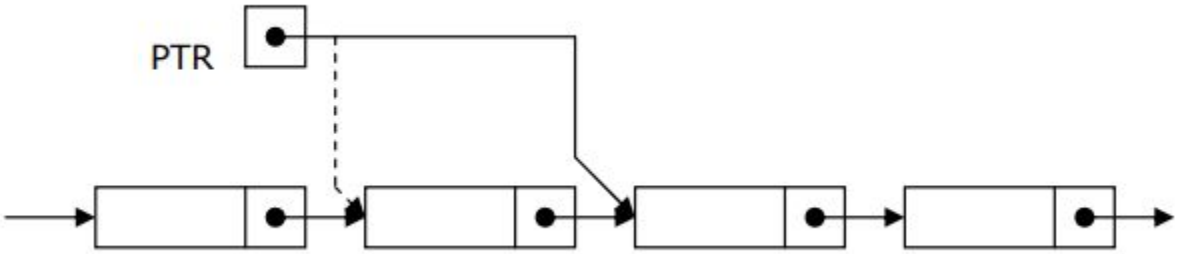


Fig. 5.4

Traversing a Linked List

Algorithm: (Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of list. The variable PTR point to the node currently being processed.

1. Set PTR=START. [Initializes pointer PTR.]
2. Repeat Steps 3 and 4 while PTR!=NULL.
3. Apply PROCESS to INFO[PTR]. // PTR-> INFO
4. Set PTR = LINK [PTR] [PTR now points to the next node.]
   // PTR= PTR-> NEXT
   [End of Step 2 loop.]
5. Exit.



Traversing a Linked List

Example 5.6

The following procedure prints the information at each node of a linked list. Since the procedure must traverse the list, it will be very similar to Algorithm 5.1.

Procedure: PRINT(INFO, LINK, START)
This procedure prints the information at each node of the list.

1. Set PTR := START.
2. Repeat Steps 3 and 4 while PTR ≠ NULL:
3. Write: INFO[PTR].
4. Set PTR := LINK[PTR]. [Updates pointer.]
   [End of Step 2 loop.]
5. Return.

Example 5.7

The following procedure finds the number NUM of elements in a linked list.

Procedure: COUNT(INFO, LINK, START, NUM)

1. Set NUM : = 0. [Initializes counter.]
2. Set PTR : = START. [Initializes pointer.]
3. Repeat Steps 4 and 5 while PTR ≠ NULL.
4. Set NUM : = NUM + 1. [Increases NUM by 1.]
5. Set PTR : = LINK[PTR]. [Updates pointer.]
   [End of Step 3 loop.]
6. Return.

Searching a Linked List

Algorithm: SEARCH(INFO, NEXT, HEAD, ITEM, PREV, CURR, SCAN)
LIST is a linked list in the memory. This algorithm finds the location LOC of the node where ITEM first appear in LIST, otherwise sets LOC=NULL.

1. Set PTR=START.
2. Repeat Step 3 and 4 while PTR≠NULL:
3. if ITEM = INFO [PTR] then: // ITEM = PTR->INFO
   Set LOC=PTR, and Exit. [Search is successful.]
   Else:
   Set PTR=LNK [PTR] // PTR = PTR->NEXT
   [End of Step 2 loop.]
4. Set LOC=NULL [Search is unsuccessful.]
5. Exit.

Searching a Sorted Linked List

Suppose the data in LIST are sorted. Again we search for ITEM in LIST by traversing the list using a pointer variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one, of LIST. Now, however, we can stop once ITEM exceeds INFO[PTR]. The algorithm follows.

Algorithm 5.3: SRCHSL(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]. [PTR now points to next node.]
   Else if 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 algorithms; that is, the worst-case running time is proportional to the number n of elements in LIST, and the average-case running time is approximately proportional to $n/2$.

- Binary search algorithm can not be applied to a sorted linked list, since there is no way of indexing the middle element in the list.

Array vs Linked list

.

	Array	Linked List
Strength	• Random Access (Fast Search Time) • Less memory needed per element • Better cache locality	• Fast Insertion/Deletion Time • Dynamic Size • Efficient memory allocation/utilization
Weakness	• Slow Insertion/Deletion Time • Fixed Size • Inefficient memory allocation/utilization	• Slow Search Time • More memory needed per node as additional storage required for pointers