Lecture 10
Linked List Operations

In this lecture

- Concept of a linked list revisited
- Types of Linked Lists
- Designing a node of a Linked List
- Operations on Linked Lists
    - Appending a node to a Linked List
    - Prepending a node to a linked list
    - Inserting a node in order to a linked list
    - Deleting a node from a linked list
    - Reversing a Linked List
- Doubly Linked Lists
- Further Readings
- Exercises

Concept of a linked list revisited

Static arrays are structures whose size is fixed at compile time and therefore cannot be extended or reduced to fit the data set. A dynamic array can be extended by doubling the size but there is overhead associated with the operation of copying old data and freeing the memory associated with the old data structure. One potential problem of using arrays for storing data is that arrays require a contiguous block of memory which may not be available, if the requested contiguous block is too large. However the advantages of using arrays are that each element in the array can be accessed very efficiently using an index. However, for applications that can be better managed without using contiguous memory we define a concept called "linked lists".

A **linked list** is a collection of objects linked together by references from one object to another object. By convention these objects are named as **nodes**. So the basic linked list is collection of nodes where each node contains one or more data fields AND a reference to the next node. The last node points to a NULL reference to indicate the end of the list.

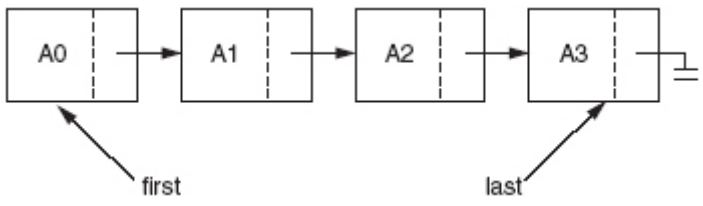


image source: Weiss Data Structures

The entry point into a linked list is always the first or head of the list. It should be noted that head is NOT a separate node, but a reference to the first Node in the list. If the list is empty, then the head has the value NULL. Unlike Arrays, nodes cannot be accessed by an index since memory allocated for each individual node may not be contiguous. We must begin from the head of the list and traverse the list sequentially to access the nodes in the list. Insertions of new nodes and deletion of existing nodes are fairly easy to handle and will be discussed in the next lesson. Recall that array insertions or deletions may require adjustment of the array (overhead), but insertions and deletions in linked lists can be performed very efficiently.

Types of Linked Lists

Linked lists are widely used in many applications because of the flexibility it provides. Unlike arrays that are dynamically assigned, linked lists do not require memory from a contiguous block. This makes it very appealing to store data in a linked list, when the data set is large or device (eg: PDA) has limited memory. One of the disadvantages of linked lists is that they are not random accessed like arrays. To find information in a linked list one must start from the head of the list and traverse the list sequentially until it finds (or not find) the node. Another advantage of linked lists over arrays is that when a node is inserted or deleted, there is no need to "adjust" the array.

There are few different types of linked lists. A **singly linked list** as described above provides access to the list from the head node. Traversal is allowed only one way and there is no going back. A **doubly linked list** is a list that has two references, one to the next node and another to previous node. Doubly linked list also starts from head node, but provide access both ways. That is one can traverse forward or backward from any node. A **multilinked list** (see figures 1 & 2) is a more general linked list with multiple links from nodes. For examples, we can define a Node that has two references, age pointer and a name pointer. With this structure it is possible to maintain a single list, where if we follow the name pointer we can traverse the list in alphabetical order of names and if we traverse the age pointer, we can traverse the list sorted by ages. This type of node organization may be useful for maintaining a customer list in a bank where same list can be traversed in any order (name, age, or any other criteria) based on the need.

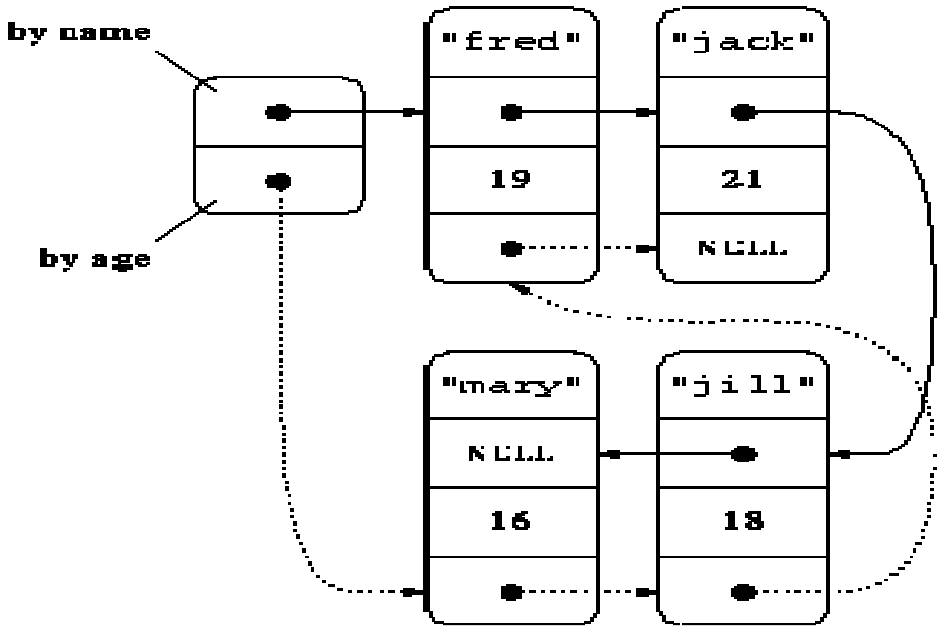


Figure 1 – Linked List with two pointers

Another example of multilinked list is a structure that represents a sparse matrix as shown below. Sparse matrices are widely used in applications. A sparse matrix is a large table of data where most are undefined. For example, we may have a 10^6 x 10^6 table of integers. If we just allocate an array to hold this table, we would require 4 x 10^6 x 10^6 bytes of memory. This is a very large chunk of contiguous memory that most computers don't have. Using a table where most entries are undefined to store this matrix is a bad idea. Therefore we come up with a data structure where only store the defined values. A typical record in the structure below consists of row and column index of the entry, value of the entry, a pointer to next row and a pointer to next column.

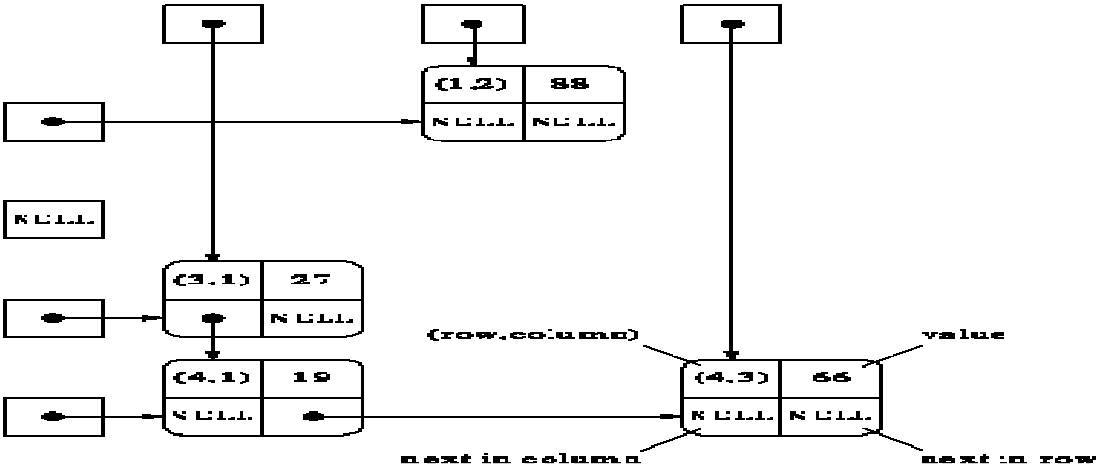


Figure 2 – A sparse matrix representation

Another important type of a linked list is called a **circular linked list** where last node of the list points back to the first node (or the head) of the list.

Designing the Node of a Linked List

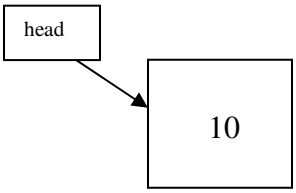
Linked list is a collection of linked nodes. A node is a struct with at least a data field and a reference to a node of the same type. A node is called a **self-referential** object, since it contains a pointer to a variable that refers to a variable of the same type. For example, a struct Node that contains an int data field and a pointer to another node can be defined as follows.

```
typedef struct node {
    int data;
    struct node* next;
} node;

node* head = NULL;
```

Creating the first node

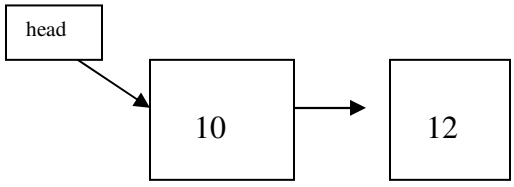
Memory must be allocated for one node and assigned to head as follows.

```
head = (node*) malloc(sizeof(node));

head➔data = 10;

head➔next = NULL;
```



Adding the second node and linking

```
node* nextnode = malloc(sizeof(node));
nextnode➔data = 12;
nextnode➔next = NULL;
head➔next = nextnode;
```



Operations on Linked Lists: We studied the fundamentals of linked lists in previous lesson. In this lesson, we will focus on some of the basic operations on linked lists. In a well-designed list data structure, you should be able to manipulate its elements without knowing anything about its data. Unlike arrays, the entry point into any linked list is the head of the list. One important thing to understand is that ***head*** of the list is a nothing but ***a reference to the first node*** in the list. Therefore for an empty list, the value of head = NULL. We also learned that any linked list ends with null pointer. We must take great care in manipulating linked lists since any misguided link in the middle will make the entire list inaccessible and error prone. In C, we have to be extra careful since memory allocations and de-allocations must be done manually. We shall discuss here, some of the basic operations that can be performed on a linked list.

1. Traversing a linked list.
2. Append a new node (to the end) of a list
3. Prepend a new node (to the beginning) of the list
4. Inserting a new node to a specific position on the list
5. Deleting a node from the list
6. Updating a node in the list

Traversing a Linked List

The idea here is to step through the list from beginning to the end. Traversing a linked list is important for many applications. For examples, we may want to print the list or search for a specific node in the list. Or we may want to perform an advanced operation on the list as we traverse the list. The algorithm for traversing a list is fairly trivial.

a. Start with the head of the list. Access the content of the head node if it is not null.
b. Then go to the next node(if exists) and access the node information
c. Continue until no more nodes (that is, you have reached the null node)

Code Example: The following function traverses through a linked list of integers and return a String containing all the data (all integers separated by a comma) in the list.

```
int toString(node* head, char** s){

}
```

The function takes a pointer to the head of the list and the address of a string. Function returns 0 if successful, 1 otherwise.

Appending a new Node to the end of a Linked List

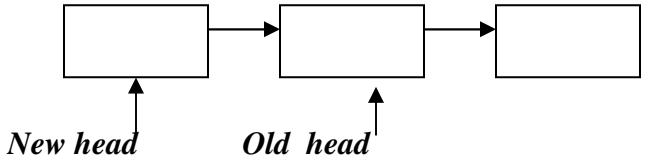
Sometimes we must append (or insert to end) new nodes to the list. Since we only have information about the head of the list(unless you maintain a last pointer), we need to traverse the list until we find the last node. Then we insert new node to the end of the list. Note that we have to consider special cases such as list being empty.

Exercise: Complete the following function

int append(node head, node* N); // appends a node N to the list**

Prepending a new Node to the beginning of a new Linked List

Sometimes we must prepend(or insert to beginning) a new node to the list. Prepending a node to a list is easy since there is no need to find the end of the list. If list is empty, we make new node the head of the list. Otherwise, we connect new node to the current head of the list and make new node the head of the list.

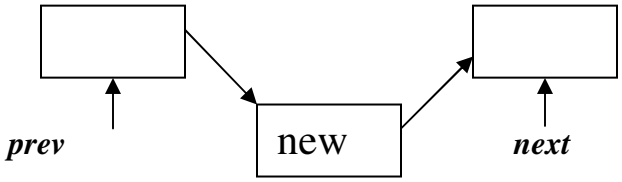


Exercise: Complete the following method

int append(node head, node* N); // append node N and return 0 if successful**

Inserting a new Node to a Sorted Linked List

Sometimes we must insert (to some specific location) new nodes to a list. It is important that we traverse the list to find the place to insert the node. In the traversal process, we must maintain a reference to previous and next nodes of the list. Then we will insert a new node between previous and next.

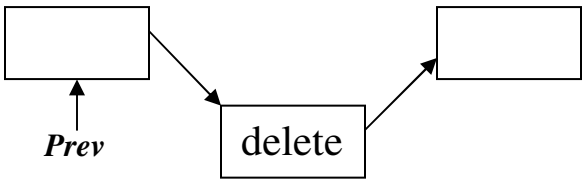


Exercise: Complete the following method

int insert(node head, node* N); // insert node N to a sorted list**

Deleting a Node from a Linked List

Sometimes we must delete a specific node from the list (if exists). It is important that we traverse the list to find the node to delete. In the traversal process, we must maintain a reference to previous node of the list. We need to worry about de-allocating memory associated with the deleted node, since C has no automatic garbage collection.



Exercise: Complete the following method

int delete(node head, node* N); // delete the node N (if exists)**

In this lesson we learned how to traverse, append, prepend, insert and delete nodes from a linked list. These operations are quite useful in many practical applications. Consider Microsoft powerpoint. We can think of powerpoint presentation as a list, whose nodes are the individual slides. In the process of creating and managing slides we can use the concepts such as inserting and deleting learned in this lesson. Therefore a data structure such as a linked list can be used to implement applications like powerpoint.

Doubly Linked Lists

A doubly linked list is a list that contains links to next and previous nodes. Unlike singly linked lists where traversal is only one way, doubly linked lists allow traversals in both ways. A generic doubly linked list node can be designed as:

```
typedef struct node {
    void* data;
    struct node* next;
    struct node* prev;
} node;
```

node* head = (node*) malloc(sizeof(node));

The design of the node allows flexibility of storing any data type as the linked list data. For example,

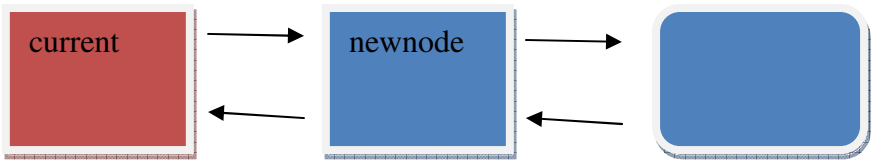
head➔ data = malloc(sizeof(int)); head➔ data = 12;

or

head➔ data = malloc(strlen("guna")+1); strcpy(head➔ data, "guna");

Inserting to a Doubly Linked Lists

Suppose a new node, **newnode** needs to be inserted after the node current



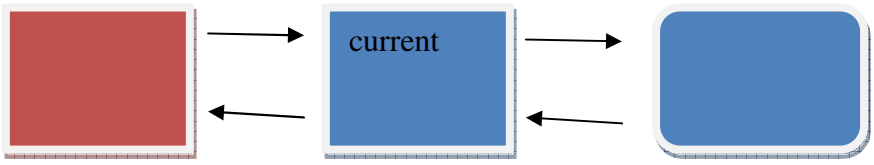
The following code can then be written

newnode➔ next = current➔next; current➔next = newnode;

newnode➔prev = current; (current➔next)➔prev = newnode;

Deleting a Node from a Doubly Linked Lists

Suppose a new node, **current** needs to be deleted



The following code can then be written

node* N = current➔prev

N➔ next = current➔next;

(N➔next)➔prev = N;

free(current);

Doubly linked lists (DLL) are also widely used in many applications that deals with dynamic memory allocation and deallocation. Although an additional pointer is used to allow traversal in both ways, DLL's are ideal for applications that requires frequent insertions and deletions from a list.

Further Readings

[1] http://www.cs.cmu.edu/~thoffman/S09-15123/Chapter-4/Chapter-4.html#CHAP_4.2

Exercises

For exercises 10.1-10.3, assume the following definition of node struct.

```
typedef struct node {
    int data;
    struct node* next;
} node;
```

10.1 Write a function that reverses a singly linked list by reversing the links

int reverse(node head);**

10.2 Write a function that sorts an existing linked list of ints using insertion sort. Do not create new nodes.

int sort(node head);**

10.3 Write a function that takes a regular linked list and makes it a circular linked list. Do not create new nodes.

int makecircular(node* head);

10.4 Explain pros of cons of using a linked list versus dynamic array.

10.5 Design the node structs needed to implement data structures shown in figure 1 and figure 2. How many bytes are needed for each of the structures?

10.6 Why is it important to pass the address of a node* (i.e. node**) instead of a node* to a function as defined in 10.1 and 10.2?