ECE 220 Computer Systems & Programming

Lecture 19: Linked Lists

ECE ILLINOIS

ILLINOIS

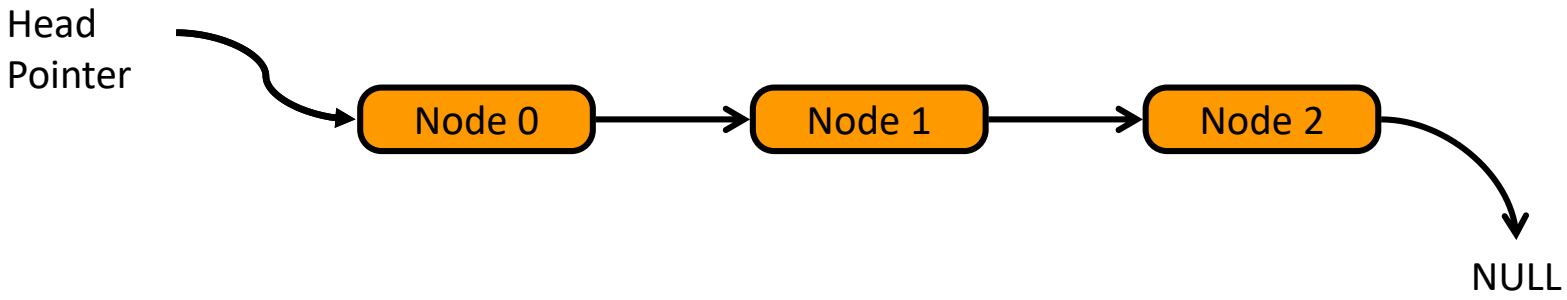
Outline

- Linked List Data Structure
- Chapter 19.5
- Key concepts
  - Search/Add/Delete operations on a linked list
  - Linked Lists vs Arrays

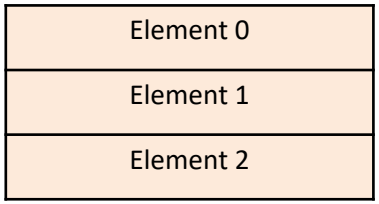
The Linked List Data Structure

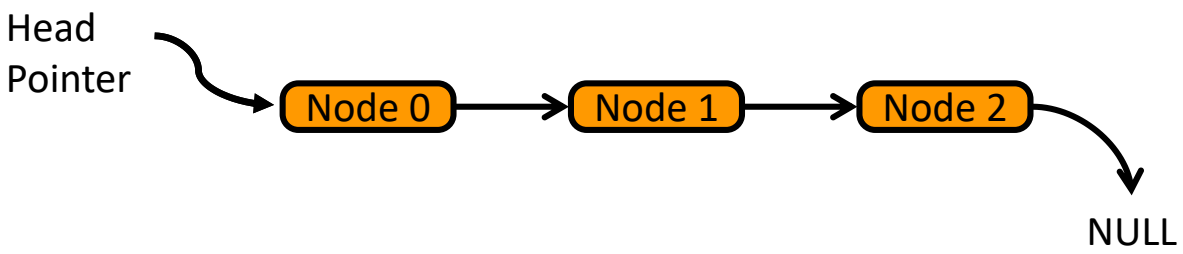
A linked list is an ordered collection of nodes, each of which contains some data, connected using pointers.

- Each node points to the next node in the list.
- The first node in the list is called the head
- The last node in the list is called the tail



Array vs Linked List

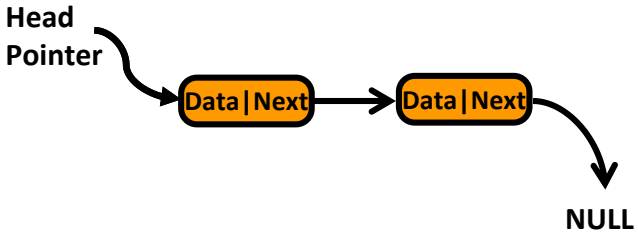




	Array	Linked List
Memory Allocation	Static/Dynamic	Dynamic
Memory Structure	Contiguous	Not necessary consecutive
Order of Access	Random	Sequential
Insertion/Deletion	Create/delete space, then shift all successive elements	Change pointer address

Example: Student Record

```
typedef struct studentStruct Record;
struct studentStruct
{
   char Name[100];
   int UIN;
   float GPA;
   Record *next;
};
```



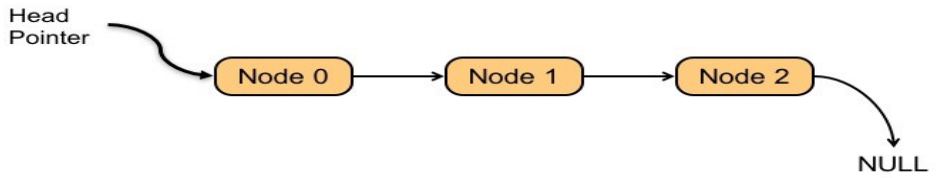
We have a list of 200 student records sorted by UIN

1. Find a particular student record by UIN
2. Add a new student record to the list at the correct place
3. Delete a student record from the list

Create a Simple Linklist:

```c
int main()
{
    Record *head = (Record *)malloc(sizeof(Record));

    head->UIN = 12345;

    int i;
    Record *current = head;

    for(i=1;i<5;i++)
    {
        current->next = (Record *)malloc(sizeof(Record));
        current->next->UIN = i*2+12345;
        current = current->next;
    }
    current->next = NULL;

    current = head;
    for(i=0;i<5;i++)
    {
        printf("Node %d: UIN: %d\n", i, current->UIN);
        current = current->next;
    }
```

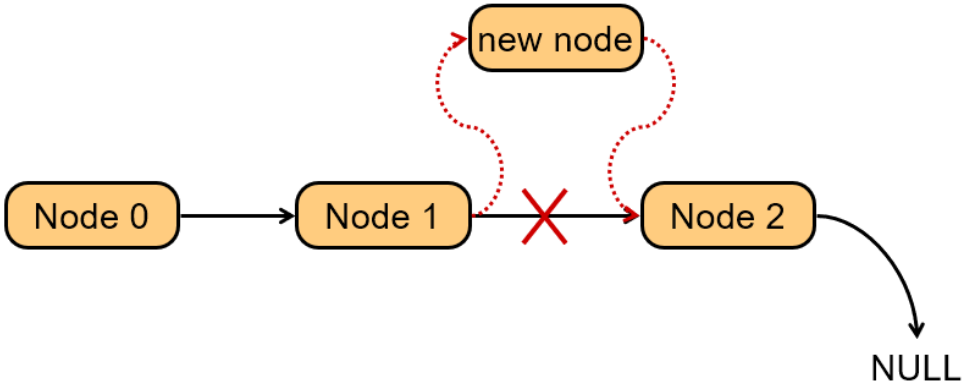
Find a Student Record by UIN



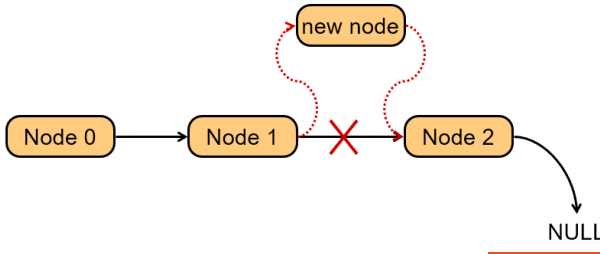

```
//print "Student Record Found" if UIN is found, return a pointer to this record
//otherwise print "Record Not Found", return NULL
Record *find_node(Record *head, int UIN)
{
    Record *current = head;

    //keep traversing the list while 1)not at the end of the list AND
    //                               2)current record's UIN < UIN we are looking for

}
```

Adding a Node

Create a new node with the proper info.
Find the node (if any) with a greater UIN.
“Splice” the new node into the list:

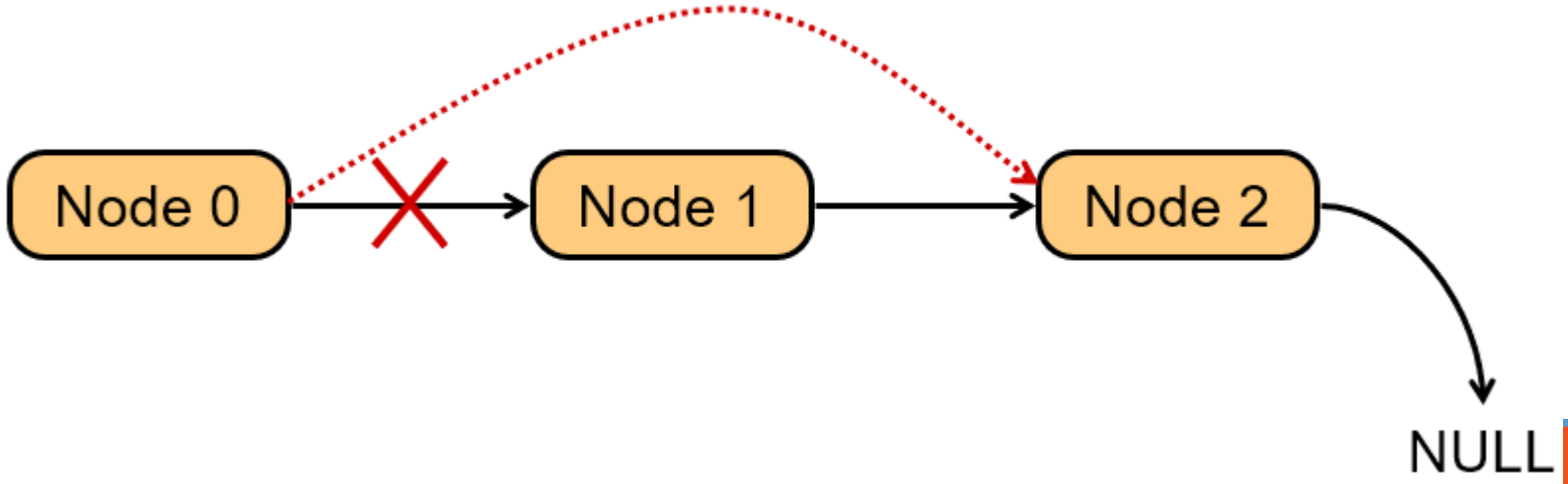


```
void add_node(Record **list, int new_UIN)
{
    Record *current = *list;
    Record *prev = *list;
    Record *temp = (Record *)malloc(sizeof(Record)); //allocate memory for the new node
    //initialize UIN for the new node

    //keep traversing the list until we reach the end
    while (       )
    {
        //the first instance when new_UIN is smaller than current record's UIN
        //we want to insert new node in front of the current node

            //if the current node is the head, update head pointer

            else //for everything else, update previous node's next pointer

            return; //exit function
        }
        //we've reach the last note and its UIN is still smaller than new node's UIN
        //new node will have to be inserted at the tail

        {

        }
        prev = current;
        current = current ->next;
    }
}
```



Deleting a Node

Find the node that points to the desired node.
Redirect that node's pointer to the next node (or NULL).
Free the deleted node's memory.



```
void remove_node(Record **list, int old_UIN)
{
    Record *prev;
    Record *current = *list;

    while(current != NULL) //find the record with matching UIN
    {


    }
                    //if record is not found, return out
        return;

                    //if record is the first node, update head pointer
    {

    }
    else //record found is in the middle of the list
    {

    }

    free(current);
}
```