# ECE 220 Computer Systems & Programming

Lecture 17: Linked Lists

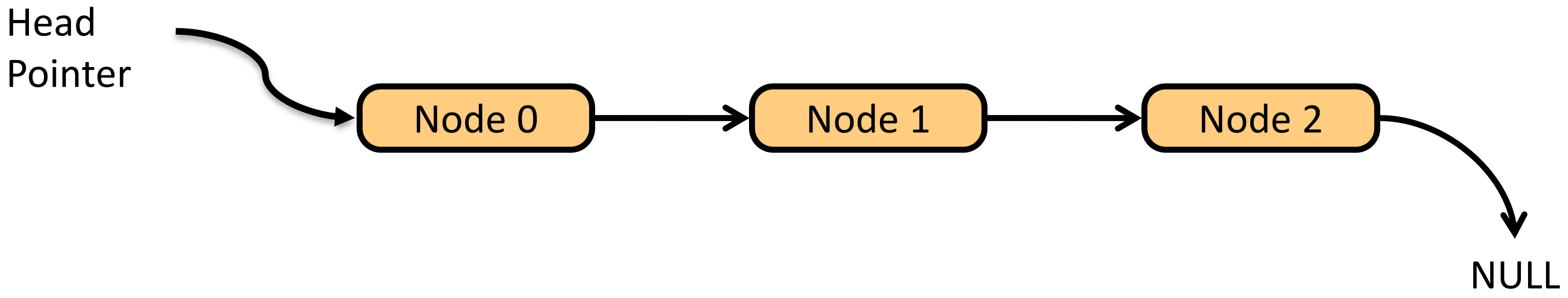
ECE ILLINOIS

ILLINOIS

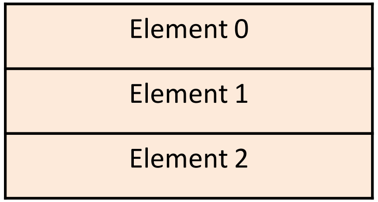
# The Linked List Data Structure

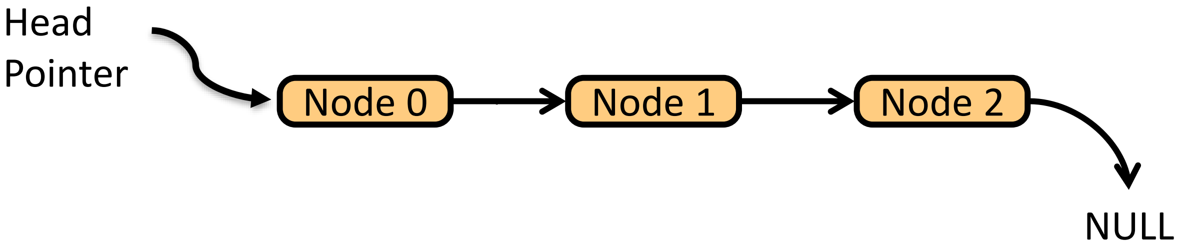
A linked list is an ordered collection of nodes, each of which contains some data, connected using pointers.

- 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

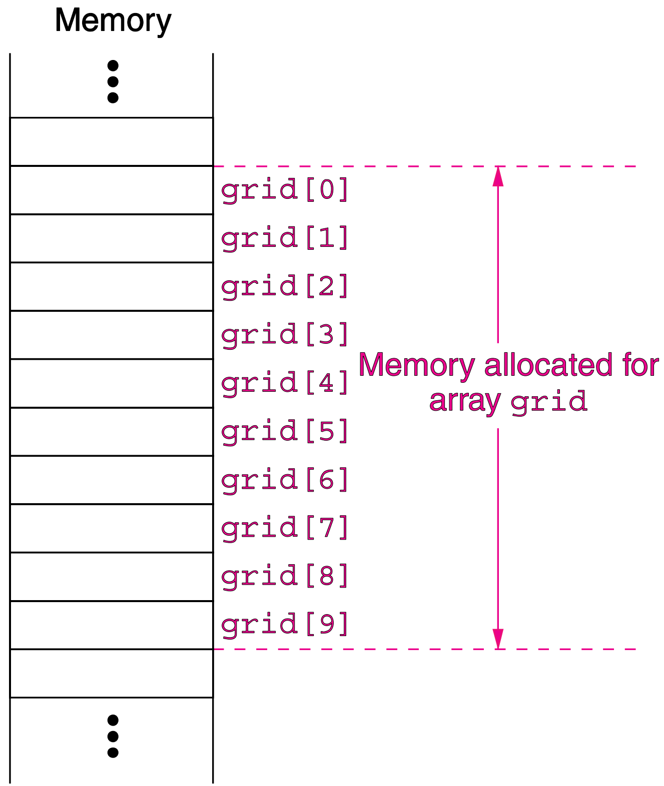


# Array vs Linked List

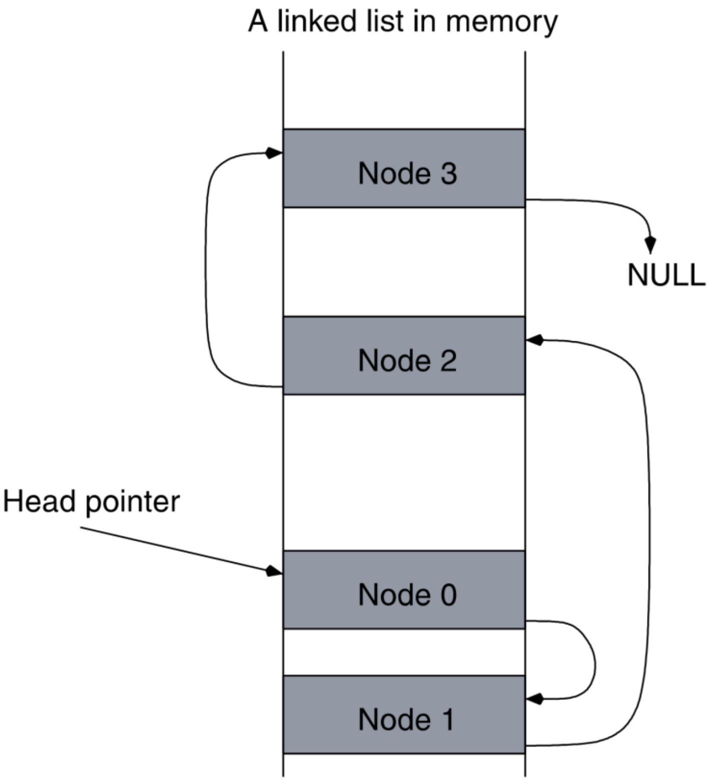



| | Array | Linked List |
|---|---|---|
| Memory Allocation | Static/Dynamic | Dynamic |
| Memory Structure | Contiguous | Not necessary consecutive |
| Order of Access | Random | Sequential |
| Insertion/Deletion | Create/delete space, then shift all successive elements | Change pointer address |



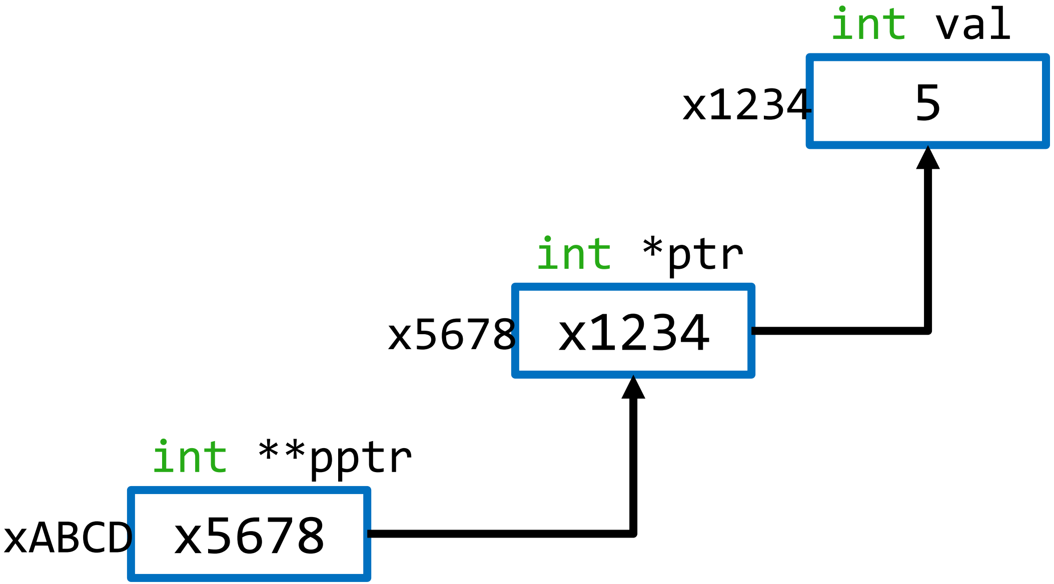

**Array**
(can be automatic or dynamic)




**Linked List**
(dynamic only)

# Review: Double Pointer

```
int val = 5;
int *ptr;

ptr = &val;

int **pptr;
pptr = &ptr;
```




```
printf("x%X\n", &pptr);
printf("x%X\n", pptr);
printf("x%X\n", *pptr);
printf("%d\n", **pptr);
```

# Example: Linkedlist and its runtime stack

```
typedef struct person_node Person;
struct person_node
{
   char name[20];
   Person *next;
};
```

```
int main()
{
   Person *theList = NULL;
   AddPerson(&theList, “Bob");
   AddPerson(&theList, "Bill");
}
```

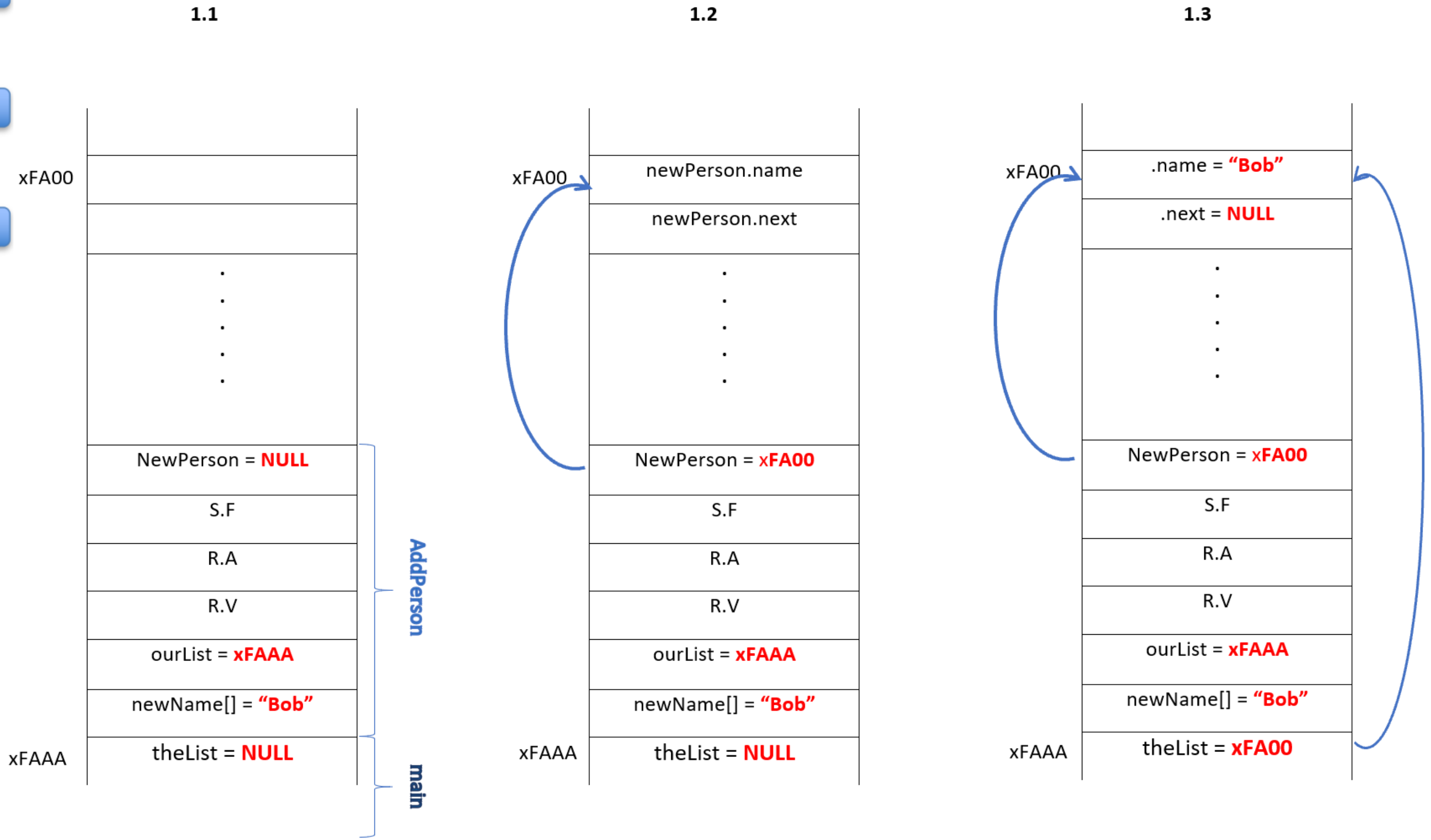
```
/* add to the linked list */
int AddPerson(Person **ourList, char newName[])
{
   Person *newPerson = NULL;

   newPerson = (Person *)malloc(sizeof(Person));
   if (newPerson == NULL)
      return 0;

   strcpy(newPerson->name, newName);
   newPerson->next = *ourList;

   *ourList = newPerson;

   return 1;
}
```

** All the addresses are hypothetical; they are used to help us visualize the memory layout and implementation of the linked list

```
/* add to the linked list */
int AddPerson(Person **ourList, char newName[])
{
   Person *newPerson = NULL;
                                                  1
   newPerson = (Person *)malloc(sizeof(Person));
   if (newPerson == NULL)
      return 0;

   strcpy(newPerson->name, newName);              2
   newPerson->next = *ourList;

   *ourList = newPerson;

   return 1;                                      3
}
```

```
typedef struct person_node Person;
struct person_node
{
   char name[20];
   Person *next;
};
```

```
int main()
{
   Person *theList = NULL;
   AddPerson(&theList, "Bob");
   AddPerson(&theList, "Bill");
}
```



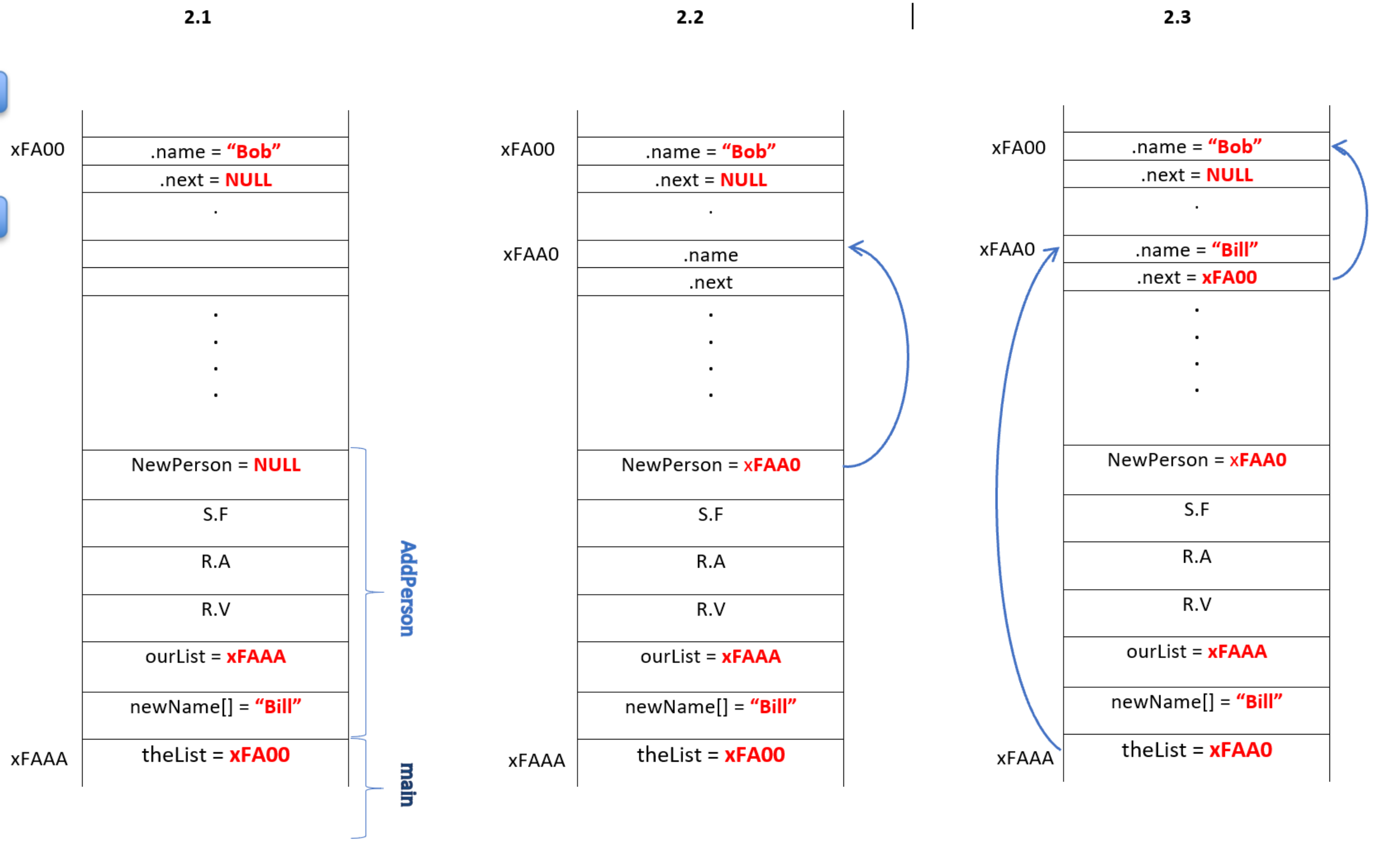

Runtime stack - **AddPerson(&theList, "Bob");**

```
/* add to the linked list */
int AddPerson(Person **ourList, char newName[])
{
  Person *newPerson = NULL;                              1

  newPerson = (Person *)malloc(sizeof(Person));
  if (newPerson == NULL)
    return 0;

  strcpy(newPerson->name, newName);                      2
  newPerson->next = *ourList;

  *ourList = newPerson;

  return 1;                                              3
}
```

```
typedef struct person_node Person;
struct person_node
{
   char name[20];
   Person *next;
};
```

```
int main()
{
  Person *theList = NULL;
  AddPerson(&theList, “Bob");
  AddPerson(&theList, "Bill");
}
```

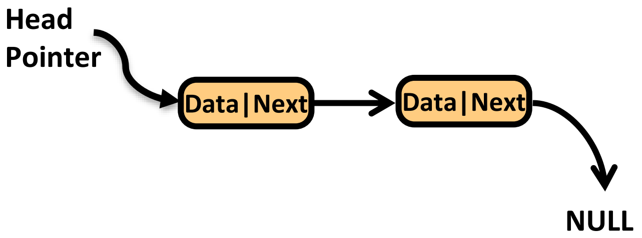
** All the addresses are hypothetical; they are used to help us visualize the memory layout and implementation of the linked list




Runtime stack - **AddPerson(&theList, “Bill");**

# Exercise: Student Record

```
typedef struct studentStruct
{
    char *Name;
    int UIN;
    float GPA;
   struct studentStruct *next;
}student;
```




1. Create a list of 5 students. The last student will take the head position and the first student will take the tail position. For Name, we will allocate space into the heap based on the given name length.
2. Add a new student to the tail position.
3. Remove a student record from the list.
4. Free up the memory space

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

typedef struct studentStruct{
        char *name;
        int UIN;
        float GPA;
        struct studentStruct *next;
}student;
```

```c
int main()
{
        student *headptr=NULL;

        //first student node
        student temp;
        temp.name = (char *)malloc(sizeof("abcd")+1);
        strcpy(temp.name, "abcd");
        temp.UIN=1112;
        temp.GPA=3.1;
        temp.next=NULL;
        insert_head(&headptr, &temp);


        //second student node
        temp.name = (char *)malloc(sizeof("bcde")+1);
        strcpy(temp.name, "bcde");
        temp.UIN=1113;
        temp.GPA=3.0;
        temp.next=NULL;
        insert_head(&headptr, &temp);
```

# void insert_head(student **head, student *data)

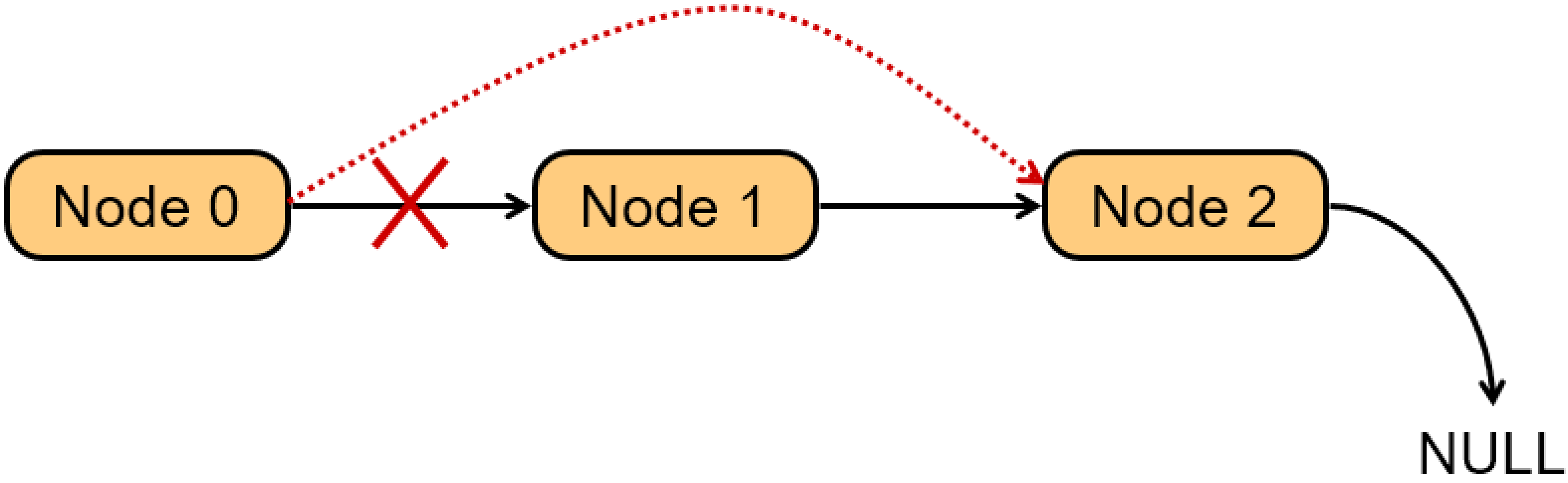
```
void insert_head(student **head, student *data)
{
        student *tmp=(student*)malloc(sizeof(student));
                *tmp=*data;
        tmp->next=*head;
        *head=tmp;
}
```

```c
//insert student node at the tail
temp.name = (char *)malloc(sizeof("LMNO")+1);
strcpy(temp.name, "LMNO");
temp.UIN=2227;
temp.GPA=2.1;
temp.next=NULL;

insert_tail(&headptr, &temp);
```

```c
void insert_tail(student **head, student *data)
{
        while(*head)
        {
                head=&((*head)->next);
        }

        student *tmp=(student*)malloc(sizeof(student));
                *tmp=*data;

        *head=tmp;
}
```

# 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.**



int remove_student(student **head, int uin)

```c
int remove_student(student **head, int uin)
{
        student *previous;
        student *current;
        current=*head;
        while(current)
        {
                if(current->UIN==uin)
                        break;
                previous=current;
                current=current->next;
        }

        if(current==NULL)
                return 0;

        if(current==*head)
        {
                *head=current->next;
        }
        else
        {
        previous->next=current->next;
        }
        free(current->name);
        free(current);
        return 1;

}
```

# Free up the memory allocations: void delete_record(student **head)

```
void delete_record(student **head)
{

student *tmp;

while(*(head)!=NULL)
{
free((*head)->name);
tmp=(*head)->next;
free(*head);
*(head)=tmp;
}
}
```