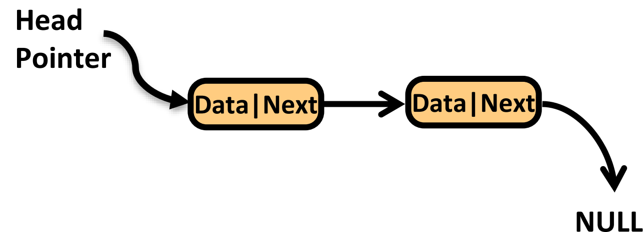
ECE 220 Computer Systems & Programming

Lecture 18: **Problem Solving with Linked List**

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. Add a new student before a known student
4. Add a new student after a known student
5. Remove a student record from the list.
6. 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);
```

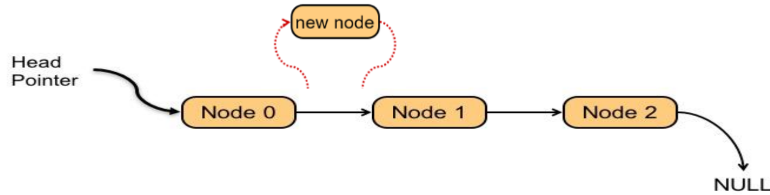
void insert_head(student **head, student *data)

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

```
//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);
```

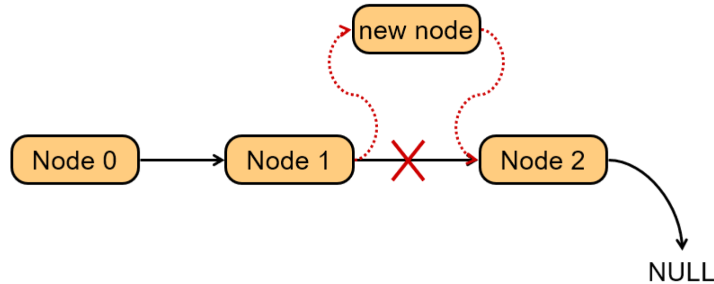
```
void insert_tail(student **head, student *data)
{
        while(*head)
        {
                head=&((*head)->next);
        }

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

        *head=tmp;
}
```

Add a new student before a known student




```c
void insert_before(student **head, student *data, int uin)
{
student *previous=*head;
student *current=*head;
while(current)
{
        if(current->UIN == uin)
                break;
                previous=current;
                current=current->next;
}
if (current==NULL)
{
        printf("student does not exist\n");
        return;
}
        student *tmp=(student *)malloc(sizeof(student));
                *tmp=*data;
if(current==*head)
{
        *head=tmp;
        tmp->next=current;
}
else
{
        previous->next=tmp;
        tmp->next=current;
}
}
```

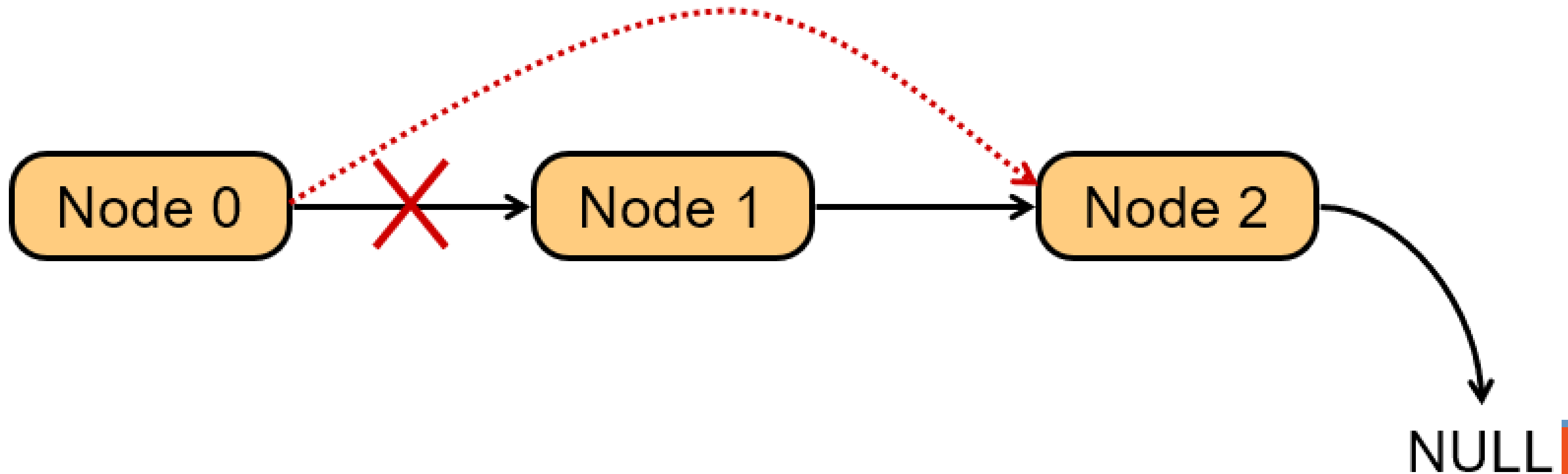
Add a new student after a known student



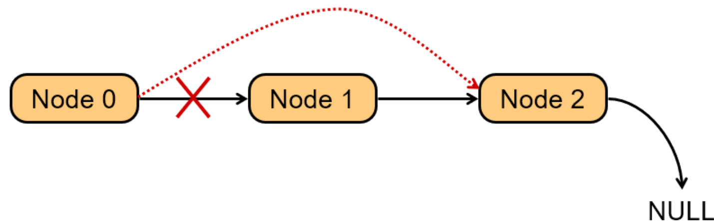

```
void insert_after(student **head, student *data, int uin)
{
student *head_next;
student *current=*head;
while(current)
{
        if(current->UIN == uin)
                break;
                current=current->next;
}
if (current==NULL)
{
        printf("student does not exist\n");
        return;
}
        student *tmp=(student *)malloc(sizeof(student));
                *tmp=*data;
if(current==*head)
{
        head_next=(*head)->next;
        (*head)->next=tmp;
        tmp->next=head_next;
}
else
{
        head_next=current->next;
        current->next=tmp;
        tmp->next=head_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.



int remove_student(student **head, int uin)



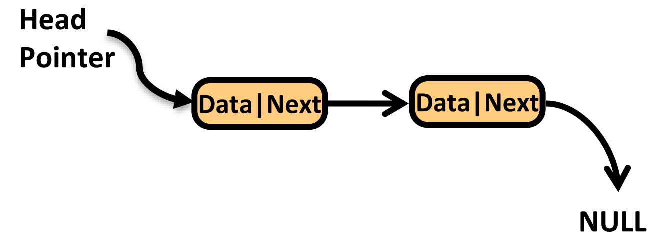

```c
int remove_student(student **head, int uin)
{
        student *previous;
        student *current;
        current=*head;
        while(current)
        {
                if(current->UIN==uin)
                        break;
                previous=current;
                current=current->next;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

        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;
}
}
```

Exercise: Create a Sorted Student Record based on GPA

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



1. Create a list of 5 students sorted in descending order according to the GPA
2. Remove a student record from the list.
3. 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;

void insert_sorted_GPA(student **headptr, student *data);
int remove_student(student **headptr, int uin);
void delete_record(student **headptr);
void print_data(student *headptr);
```

```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_sorted_GPA(&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_sorted_GPA(&headptr, &temp);
//third student node
        temp.name = (char *)malloc(sizeof("cdef")+1);
        strcpy(temp.name, "cdef");
        temp.UIN=1114;
        temp.GPA=3.9;
        temp.next=NULL;
        insert_sorted_GPA(&headptr, &temp);
```

void insert_sorted_GPA(student **head, student *data)

```
void insert_sorted_GPA(student **head, student *data){
    student *temp = (student*) malloc(sizeof(student));
    *temp = *data;
    while(*head &&  (*head)->GPA > data->GPA)
        head = &((*head)->next);

    temp->next = *head;
    *head = temp;
}
```

Another way: Using a single pointer

```
int main(){
   student head;
   head.next = NULL;

   student data;
   data.UIN = 1;
   ...

   insert_head_base(&head, &data);
```

```
void insert_head_base(student *headptr, student *data){
   student *temp = (student*) malloc(sizeof(student));
   *temp = *data;
   temp->next = headptr->next;
   headptr->next = temp;
}
```

VS

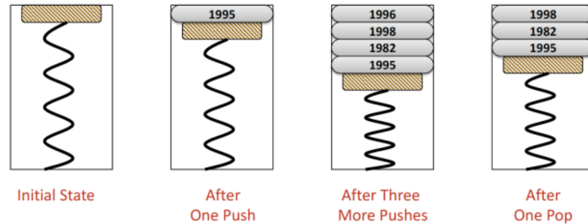
```
int main(){
   student *headptr = NULL;

   student data;
   data.UIN = 0;
         ...

   insert_head(&headptr, &data);
```

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

Stack data types

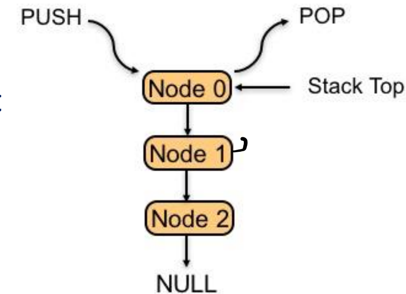
Stack

- First item in is the last item out - ________________
- Two operations for data movement: ____________ & ____________



Stack can be implemented as a linked list in which adding and removing elements occurs at the top of the list (LIFO)

Functions to add and remove elements from a stack: push and pop



Push for Stack

Same as insert_head

```
typedef struct StudentStruct{
   int UIN;
   char *netid;
   float GPA;
   struct StudentStruct *next;
}node;
```

```
void push(node **headpptr, node *data){
    node *temp = (node*) malloc(sizeof(node));
    *temp = *data;

    temp->next = *headpptr;
    *headpptr = temp;
}
```

```
int main(){

    node *headptr = NULL;

    node temp;
    temp.UIN = 1234;
    // reserve heap mem (without actual contents)
    temp.netid = (char*) malloc(sizeof("kirby") + 1);
    strcpy(temp.netid, "kirby");
    temp.GPA = 3.0;
    temp.next = NULL;

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

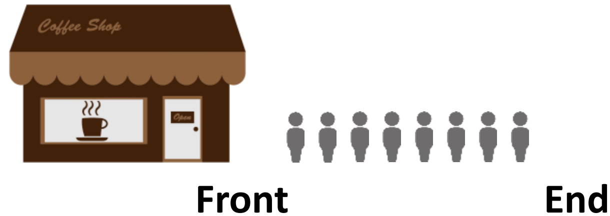
Pop for Stack

```
void pop(node **headpptr){
    node *next;
    if(*headpptr == NULL){
        printf("stack is empty.\n");
        return;
    }
    next = (*headpptr)->next;
    free((*headpptr)->netid);
    free(*headpptr);
    *headpptr = next;
}
```

Queue data types

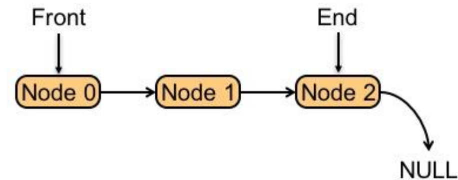
Queue

- First item in is the first item out - ________________
- Two operations for data movement: ____________ & ____________



Queue is a linked list in which adding a new element occurs at the end of the list and removing an element occurs at the start of the list.

Functions to add and remove elements: enqueue and dequeue



void enqueue(node **headpptr, node *data)

```
void enqueue(node **headpptr, node *data){
    node *temp = (node*) malloc(sizeof(node));
    *temp = *data;
    temp->next = NULL;

    while(*headpptr != NULL)
        headpptr = &((*headpptr)->next);

    *headpptr = temp;
}
```

void dequeue(node **headpptr)

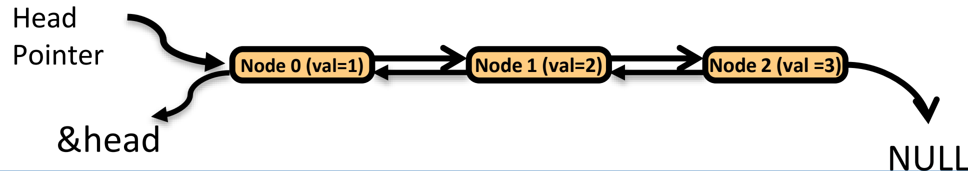
```
void dequeue(node **headpptr){
    node *next;

    if(*headpptr == NULL){
        printf("Queue is empty.\n");
        return;
    }

    next = (*headpptr)->next;
    free((*headpptr)->netid);
    free(*headpptr);
    *headpptr = next;
}
```

Example: doubly LinkedList and its runtime stack

```
typedef struct dll_node_t {
   int val;
   struct dll_node_t *next;
   struct dll_node_t **prev;
} dll_node;

int main()
{
  dll_node *head = NULL;;
 dll_insert_sorted(&head, 3);
 dll_insert_sorted(&head, 1);
 dll_insert_sorted(&head, 2);
}
```

1
2
3

```
/* add to the doubly linked list */
void dll_insert_sorted(dll_node **head, int v)
{
   dll_node *tmp = malloc(sizeof(*tmp));
   tmp->val = v;
   while (*head && ((*head)->val < v))
      head = &((*head)->next);
   tmp->next = *head;
   if (*head)
      (*head)->prev = &(tmp->next);
   tmp->prev = head;
   *head = tmp;
}
```



```
/* add to the doubly linked list */
void dll_insert_sorted(dll_node **head, int v)
{
    dll_node *tmp = malloc(sizeof(*tmp));
    tmp->val = v;
    while (*head && ((*head)->val < v))
        head = &((*head)->next);
    tmp->next = *head;
    if (*head)
        (*head)->prev = &(tmp->next);
    tmp->prev = head;
    *head = tmp;
}
```

```
int main()
{
  dll_node *head = NULL;;
  dll_insert_sorted(&head, 3);   // 1
  dll_insert_sorted(&head, 1);   // 2
  dll_insert_sorted(&head, 2);   // 3
}
```