ECE 220 Computer Systems & Programming

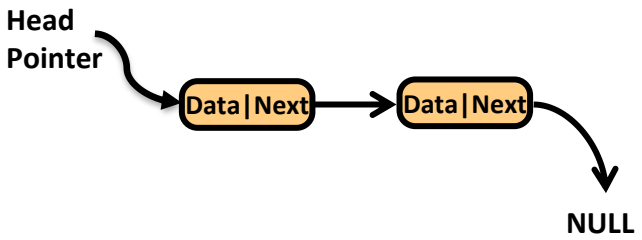
Lecture 18: **Problem Solving with Linked List**

ECE ILLINOIS

ILLINOIS

Exercise: Student Record (recap from last class)

```
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;
}
```

Creating Linked List

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

```
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;
}
```

Another way: Using a single pointer

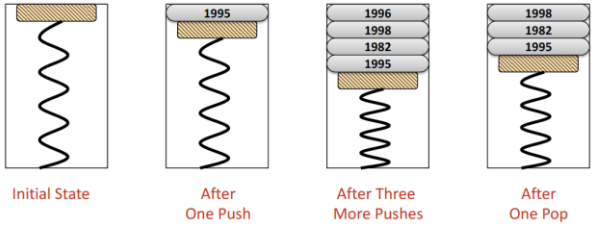
```
int main(){
   student head;
   head.next = NULL;

   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;
}
```

Stack data types

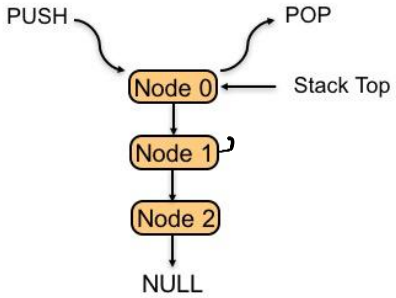
Stack

- 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);
```

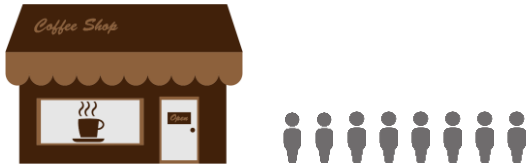
Pop for Stack

```
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

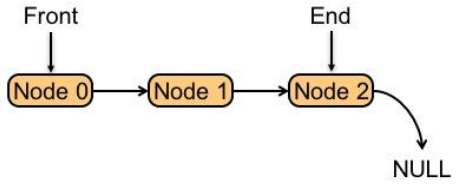
Queue

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



Front **End**

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;
}
```

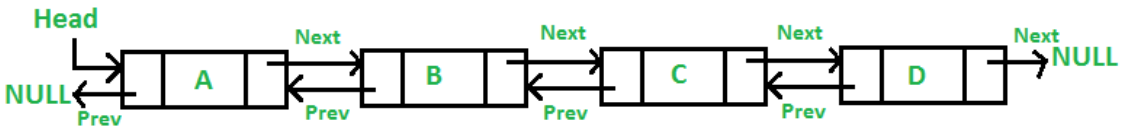
void dequeue(node **headpptr)

```
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;
}
```

Doubly Linked List

Each node in a doubly linked list contains three components:

- **Data:** data is the actual information stored in the node.
- **Next:** next is a pointer that links to the next node in the list.
- **Prev:** previous **is** a pointer that links to the previous node in the list.




```
// defining a node
typedef struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
} Node;
```

Ref: https://www.geeksforgeeks.org/c/doubly-linked-list-in-c/

code

```c
int main()
{
    Node* head = NULL;

    // Demonstrating various operations
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtBeginning(&head, 5);
    insertAtPosition(&head, 15, 2); // List: 5 15 10 20

    printf("After Insertions:\n");
    printListForward(head);
    printListReverse(head);

    deleteAtBeginning(&head); // List: 15 10 20
    deleteAtEnd(&head); // List: 15 10
    deleteAtPosition(&head, 2); // List: 15

    printf("After Deletions:\n");
    printListForward(head);

    return 0;
}
```

code

```
// Function to insert a node at the beginning
void insertAtBeginning(Node** head, int data)
{
    // creating new node
    Node* newNode = createNode(data);

    // check if DLL is empty
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    newNode->next = *head;
    (*head)->prev = newNode;
    *head = newNode;
}
```

```
// Function to create a new node with given value as data
Node* createNode(int data)
{
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}
```

code

```c
// Function to insert a node at the end
void insertAtEnd(Node** head, int data)
{
    // creating new node
    Node* newNode = createNode(data);

    // check if DLL is empty
    if (*head == NULL) {
        *head = newNode;
        return;
    }

    Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
```

```c
// Function to create a new node with given value as data
Node* createNode(int data)
{
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}
```

code

```c
// Function to insert a node at a specified position
void insertAtPosition(Node** head, int data, int position)
{
    if (position < 1) {
        printf("Position should be >= 1.\n");
        return;
    }

    // if we are inserting at head
    if (position == 1) {
        insertAtBeginning(head, data);
        return;
    }
    Node* newNode = createNode(data);
    Node* temp = *head;
    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }
    if ((temp==NULL)||(temp->next==NULL)) {
        printf(
            "Position greater than the number of nodes.\n");
        return;
    }
    newNode->next = temp->next;
    newNode->prev = temp;
    //if (temp->next != NULL) {
        temp->next->prev = newNode;
 //   }
    temp->next = newNode;
}
```

code

```c
// Function to delete a node from the beginning
void deleteAtBeginning(Node** head)
{
    // checking if the DLL is empty
    if (*head == NULL) {
        printf("The list is already empty.\n");
        return;
    }
    Node* temp = *head;
    *head = (*head)->next;
    if (*head != NULL) {
        (*head)->prev = NULL;
    }
    free(temp);
}
```

code

```
// Function to delete a node from the end
void deleteAtEnd(Node** head)
{
    // checking if DLL is empty
    if (*head == NULL) {
        printf("The list is already empty.\n");
        return;
    }

    Node* temp = *head;
    if (temp->next == NULL) {
        *head = NULL;
        free(temp);
        return;
    }
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->prev->next = NULL;
    free(temp);
}
```

code

```c
// Function to delete a node from a specified position
void deleteAtPosition(Node** head, int position)
{
    if (*head == NULL) {
        printf("The list is already empty.\n");
        return;
    }
    Node* temp = *head;
    if (position == 1) {
        deleteAtBeginning(head);
        return;
    }
    for (int i = 1; temp != NULL && i < position; i++) {
        temp = temp->next;
    }
    if (temp==NULL) {
        printf("Position is greater than the number of "
               "nodes.\n");
        return;
    }
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    free(temp);
}
```

code

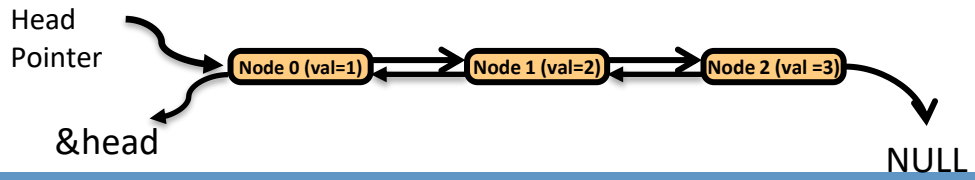
```c
// Function to print the list in forward direction
void printListForward(Node* head)
{
    Node* temp = head;
    printf("Forward List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

```c
// Function to print the list in reverse direction
void printListReverse(Node* head)
{
    Node* temp = head;
    if (temp == NULL) {
        printf("The list is empty.\n");
        return;
    }
    // Move to the end of the list
    while (temp->next != NULL) {
        temp = temp->next;
    }
    // Traverse backwards
    printf("Reverse List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
    printf("\n");
}
```

Example: doubly LinkedList (alternative approach)

```
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);
}
```

```
/* 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;;
1 dll_insert_sorted(&head, 3);
2 dll_insert_sorted(&head, 1);
3 dll_insert_sorted(&head, 2);
}
```