Outline

• Linked List
• Linked List variations
• Doubly Linked list
• Singly Linked list operations
  – Insert
  – Delete
• Linked List: Stack Implementation
• Linked List: Queue Implementation
• Deque ADT (if time permits!)
Linked List

• a **linked list** is a data structure which consists of nodes linked in a linear fashion.
• Each node in the linked list consists of two fields
  – Data Field
  – Pointer to next node
• Links are 1–1 with elements, allocated and released as necessary

```c
struct Link {               /* Single link. */
    TYPE val;                /* Data contained by this link. */
    struct Link *next;    /* Pointer to next link. */
};
```
Linked List variations

- Singly Linked list with **header** (special node to denote the start of linked list)
  ![Diagram of Singly Linked List with Header](image)

- Singly linked list with null as terminator
- Single linked list with no header
  - Use a random value to denote the start of the list.
- Singly linked list with a sentinel value for terminating the list
  - Use some random value or null data field to denote the end of the list.
- Pointer to first element only, or pointer to first and last
  - In case of Queue ADT
Doubly Linked List

• The doubly linked list consists of data fields and two links – **next** and **previous**

```c
struct Link {
    /* Double link. */
    TYPE val;   /* Data contained by this link. */
    struct Link *next; /* Pointer to next node. */
    struct Link *prev; /* Pointer to previous node. */
};
```

• **Illustration**
Singly Linked List Operations

• Traverse List

- Traverse till the end of the list

• Delete a specific node
  - Traverse to the node preceding the node to be deleted.
  - Change the pointer field to point the node succeeding the node to be deleted.
  - Free the memory for the deleted node.
  - To delete node 8 in the illustration below,

1) Traverse
2) Change the pointer field
3) Free the memory
Linked List Operation: Insert

- Insert Beginning of the list
  - 1) Allocate new link
  - 2) Update Pointers for new link

- Insert end of the list
  - 1) Allocate new link
  - 2) Update Pointers for new link
  - 3) Point new link to NULL

- Insert after a specific node
  - 1) Allocate new link
  - 2) Traverse to the link
  - 3) Update Pointers for new link
Linked list stack

• **Implementing a stack interface with a linked list:**
  - Header with head reference only: null if empty
  - No sentinel: null terminated
  - Singly linked
  - Elements added or removed from front
  - Only access first element
  - Worksheet 17 deals with this exercise.

• **Illustration**

```c
struct link {
    TYPE value;
    struct link * next;
};

struct linkedListStack {
    struct link *firstLink;
};
```
Linked List Stack: Operations

- **Push**
  ```c
  void linkedListStackPush(struct linkedListStack *s, double d){
    // Push operation of Stack using Linked list
    struct link * newLink = (struct link *) malloc(sizeof(struct link));
    assert (newLink != 0); //Create new link to store the value
    newLink->value = d;
    newLink->next = s->firstLink;
    s->firstLink = newLink; //Assign new link immediate to first link
  }
  ```

- **Top of the stack**
  ```c
  EleType linkedListStackTop (struct linkedListStack *s) {
    //Retrieving element in the top of the stack
    assert (! linkedListStackIsEmpty(s));
    return (s->firstLink->value);
  }
  ```
 Linked List: Stack Operations (contd…)

- **Pop**
  
  ```c
  void linkedListStackPop (struct linkedListStack *s) {
    //Pop operation of the stack
    struct link * lnk = s->firstLink //Create temporary link
    assert (! linkedListStackIsEmpty(s));
    s->firstLink = lnk->next;
    free(lnk); //Free memory for popped element
  }
  ```

- **isEmpty stack**
  
  ```c
  int linkedListStackIsEmpty (struct linkedListStack *s) {
    //To check if stack is empty
    return (s->firstLink == 0);
  }
  ```
Linked List: Implementation of Queue

• Queue ADT follows a FIFO (First-in-First-out) interface.
• Conceptually similar to a line (queue) of waiting people:
  - A person joins the queue by adding themselves at the end
  - The next person is removed from the front of the queue.

```c
struct link {
    TYPE value;
    struct link * next;
};

struct listQueue {
    struct link *firstLink;
    struct link *lastLink;
};
```

• Illustration
Queue ADT with Sentinels

- A sentinel is a special marker at the front and/or back of the list
- Has no value and never removed
- Helps remove special cases due to null references since it’s never null
- An empty list always has a sentinel

Illustration
Queue ADT: Operations

- **Insert Back**
  - Insert node at the back of the queue.
```c
void listQueueAddBack (struct listQueue *q, TYPE e) {
    //Adding new element to the back of the queue
    struct link *lnk = (struct link *) malloc(sizeof(struct slink));
    assert(lnk != 0);  //Allocate memory for new link
    lnk->value = e;
    lnk->next = 0;
    q->lastLink->next = lnk; //Make the tail pointer point to the new link
}
```
- **Is Queue Empty**
```c
int listQueueIsEmpty (struct listQueue *q) {
    //To check if queue is empty
    return (q->firstLink == q->lastLink);
}
```
Queue ADT: Operations (contd...)

- **Remove Front**
  - Remove node in the front of the queue.
  ```c
  void listQueueRemoveFront (struct listQueue *q) {
      //To remove front element from the queue
      struct link * lnk = q->firstLink->next;
      assert ( ! listQueueIsEmpty(q));
      q->firstLink->next = lnk->next;
      free (lnk);
  }
  ```

- **Front of the queue**
  - Retrieve the element in front of the queue.
  ```c
  TYPE listQueueFront (struct listQueue *q) {
      //Retrieve front element in the queue
      assert (! listQueueIsEmpty(q));
      return (q->firstLink->next->value);
  }
  ```
Deque ADT (Next Time..)

- What if we want to add and remove elements from both front and back?
- Need to use links going both forward and backwards
- Makes adding a new link harder, as must maintain both forward and backward links.
- Illustration

- Deque can be implemented with sentinels.
Deque ADT: Operations

- **Insert Last**
  - Insert the new node to the end of the deque.
- **Insert Front**
  - Insert the new node to the beginning of the deque.
- **Remove Last**
  - Remove the last node from the deque.
- **Remove Front**
  - Remove the first node from the deque.
- **Traverse**
  - Move through the nodes in the deque.
Deque ADT Operations: Insert

- Insert Back
  - Update the tail pointer to point the newly added node.
  - Update the prev pointer of the newly added node to point the old last node.
  - Update the next pointer of the newly added node to null.
  - Update the next pointer of the old last node to point the newly added node.

- Insert Front
  - Update the head pointer to point the newly added node.
  - Update the prev pointer of the newly added node to point the header node.
  - Update the next pointer of the newly added node to the old first node.
  - Update the prev pointer of the old first node to point the newly added node.
Deque ADT Operations: Delete

- **Delete Back**
  - Update the tail pointer to point the node previous to the removed node.
  - Update the next pointer of the node previous to the removed node to null.

- **Delete Front**
  - Update the head pointer to point the node next to the removed node.
  - Update the prev pointer of the node next to the removed node to point the header node.