BBM 201
DATA STRUCTURES

Lecture 10:
Doubly Linked Lists

2018-2019 Fall
Doubly Linked Lists

Each node stores not only the address of the next node, but also the address of the previous node. So, each node stores three fields.

**Advantage of doubly linked list:** Reverse look-up that we could not do using a linked list. For example, deletion is much faster than for a linked list.

For **temp** being 600, **temp->next** points to the address 800 and **temp->prev** points to the address 400.
Note: **head** is a global variable. Each node inside the `InsertAtHead` function is created locally and the node `myNode` does not exist after the function is executed. Therefore, local node allocation is NOT preferred.
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
struct Node* head; // global variable - pointer to head node.
struct Node* GetNewNode(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
void InsertAtHead(int x) {

}

Each node “newNode” is created in the dynamic memory and the node exists after the function is executed. Now, we create a new node in a separate function, called GetNewNode.
Now, one node is created in the list with head pointing to it using the line `head = newNode`.

We have two nodes, `head` is pointing to the node at address 400 and `newNode` is pointing to the node at address 600.

**Typo:** The data-field at address 600 has value 4 not 2.
Set the **prev-field** of the **head** node as 600 (address of the new node). Then, set the **next-field** of the **new node** as 400 (the address of the head node). And now, head can point to 600, that is the address of the final head node.

**Typo:** The data-field at address 600 has value 4 not 2.
Doubly Linked List - Implementation

```c
void InsertAtHead(int x) {
    struct Node* newNode = GetNewNode(x);
    if(head == NULL) {
        head = newNode;
        return;
    }
    head->prev = newNode;
    newNode->next = head;
    head = newNode;
}
```

Insert At Head (2)
Insert At Head (4)
Reverse Printing

```c
void ReversePrint() {
    struct Node* temp = head;
    if(temp == NULL) return; // empty list, exit
    // Going to last Node
    while(temp->next != NULL) {
        temp = temp->next;
    }
    // Traversing backward using prev pointer
    printf("Reverse: ");
    while(temp != NULL) {
        printf("%d ",temp->data);
        temp = temp->prev;
    }
    printf("\n");
}
```

In Reverse-printing, the code first goes to the end of the list and then traverses backwards.
```c
void ReversePrint() {
    struct Node* temp = head;
    if(temp == NULL) return; // empty list, exit
    // Going to last Node
    while(temp->next != NULL) {
        temp = temp->next;
    }
    // Traversing backward using prev pointer
    printf("Reverse: ");
    while(temp != NULL) {
        printf("%d",temp->data);
        temp = temp->prev;
    }
    printf("\n");
}

int main() {  
    head = NULL; // empty list.
    InsertAtHead(2); Print(); ReversePrint();
    InsertAtHead(4); Print(); ReversePrint();
    InsertAtHead(6); Print(); ReversePrint();
}
```

```
Forward: 2
Reverse: 2
Forward: 4 2
Reverse: 2 4
Forward: 6 4 2
Reverse: 2 4 6
```
Doubly Circular Linked List
typedef struct node *node_pointer;
typedef struct node{
    node_pointer leftlink;
    element item;
    node_pointer rightlink;};

ptr = ptr->leftlink->rightlink = ptr->rightlink->leftlink
void dinsert(node_pointer node, node_pointer newnode) {
    /* insert newnode to the right of node */
    newnode->leftlink = node;
    newnode->rightlink = node->rightlink;
    node->rightlink->leftlink = newnode;
    node->rightlink = newnode;
}

Insertion into an empty doubly linked circular linked list:
Deletion from a doubly-linked circular list:

```c
void ddelete(node_pointer node, node_pointer deleted) {
    if(node == deleted) {
        printf("Deletion of head node not permitted.\n");
    } else {
        deleted->leftlink->rightlink = deleted->rightlink;
        deleted->rightlink->leftlink = deleted->leftlink;
        free(deleted);
    }
}
```

Deletion from a doubly linked circular linked list:
Doubly vs. Singly Linked List

It uses extra space for previous pointers.

Insertion/Deletion has extra work.

You have ready access\insert on both ends.

It can work as a Queue and a Stack at the same time.

Node deletion requires no additional pointers.
Reverse a doubly linked list

```java
Node reverse(Node head)
{
    Node n = head, next;
    while (n.next != null){
        next = n.next;
        n.next = n.prev;
        n.prev = next;
        n = next;
    }
    //for the last node
    next = n.next;
    n.next = n.prev;
    n.prev = next;
    // n is the new head.
    return n;
}
```