

**CSE1303 Part A**  
**Data Structures and Algorithms**  
**Summer Semester 2003**

**Lecture A9 – Linked Lists**

Kymerly Fergusson

**Overview**

- Operations for Lists.
- Implementation of Linked Lists.
- Double Linked Lists.

2

**List Operations**

- Go to a position in the list.
- Insert an item at a position in the list.
- Delete an item from a position in the list.
- Retrieve an item from a position.
- Replace an item at a position.
- Traverse a list.

3

**Comparison**

**Linked Storage**

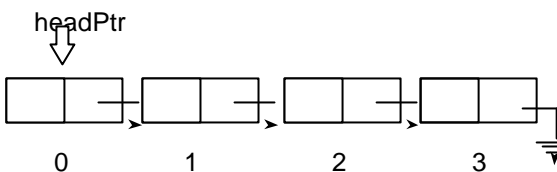
- Unknown list size.
- Flexibility is needed.

**Contiguous Storage**

- Known list size.
- Few insertions and deletions are made within the list.
- Random access

4

**Linked List**



5

```
#ifndef LINKEDLISTH
#define LINKEDLISTH
#include <stdbool.h>
#include "node.h"

struct LinkedListRec
{
    int count;
    Node* headPtr;
};

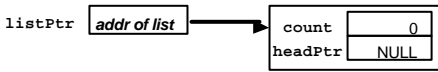
typedef struct LinkedListRec List;

void initializeList(List* listPtr);
bool listEmpty(const List* listPtr);
Node* setPosition(const List* listPtr, int position);
void insertItem(List* listPtr, float item, int position);
float deleteNode(List* listPtr, int position);

#endif
```

6

### Initialize List



```
void initializeList(List* listPtr)
{
    listPtr->headPtr = NULL;
    listPtr->count = 0;
}
```

7

### Set Position

- check if position is within range
- start with address of head node
- set count to 0
- while count is less than position
  - follow link to next node
  - increment count
- return address of current node

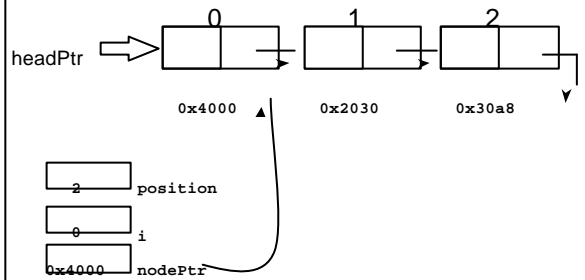
8

```
Node* setPosition(const List* listPtr, int position)
{
    int i;
    Node* nodePtr = listPtr->headPtr;

    if (position < 0 || position >= listPtr->count) {
        fprintf(stderr, "Invalid position\n");
        exit(1);
    }
    else {
        for (i = 0; i < position; i++) {
            nodePtr = nodePtr->nextPtr;
        }
    }
    return nodePtr;
}
```

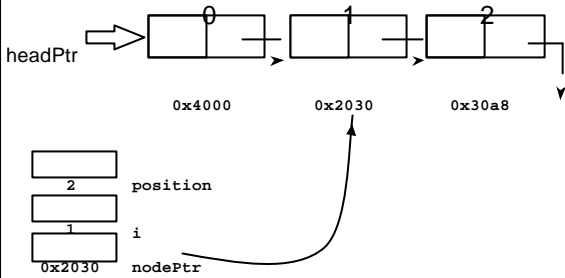
9

### Set Position



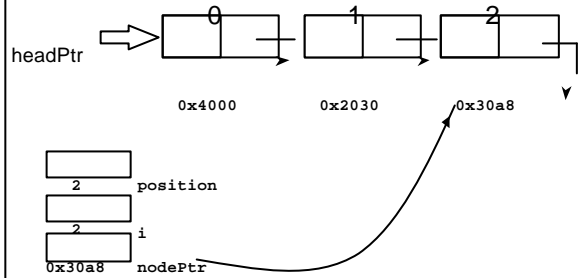
10

### Set Position

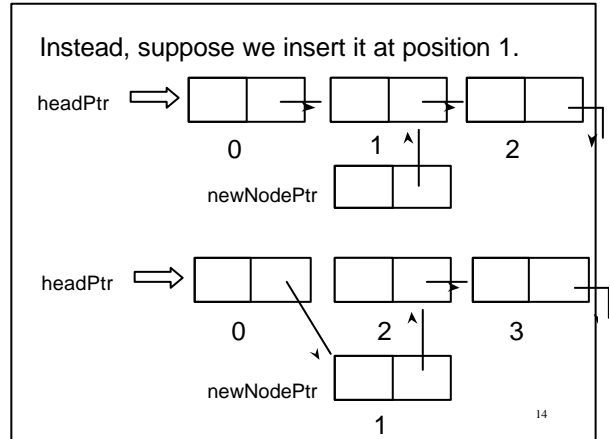
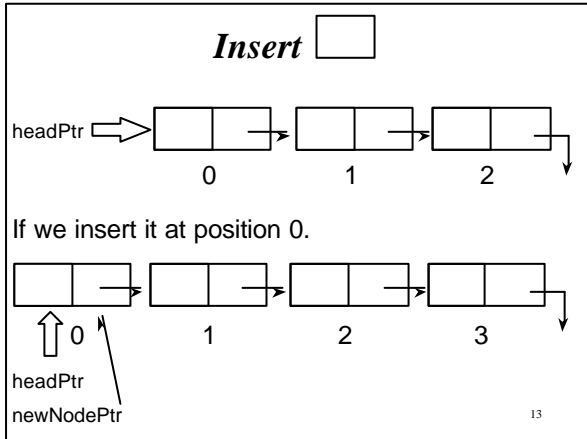


11

### Set Position



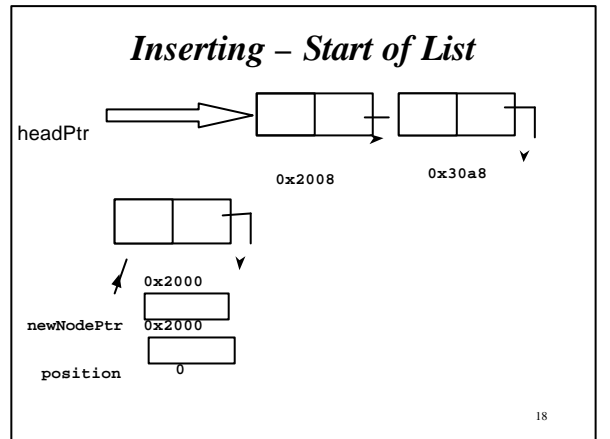
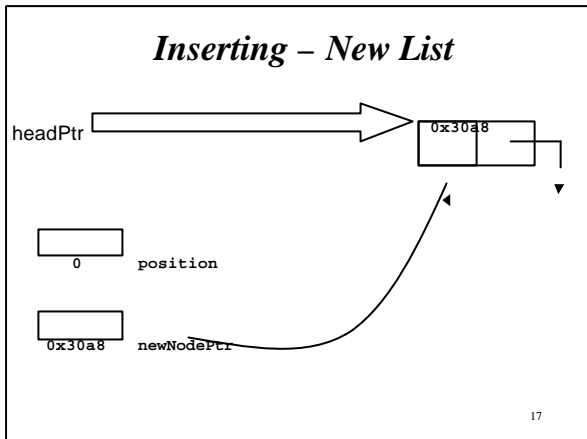
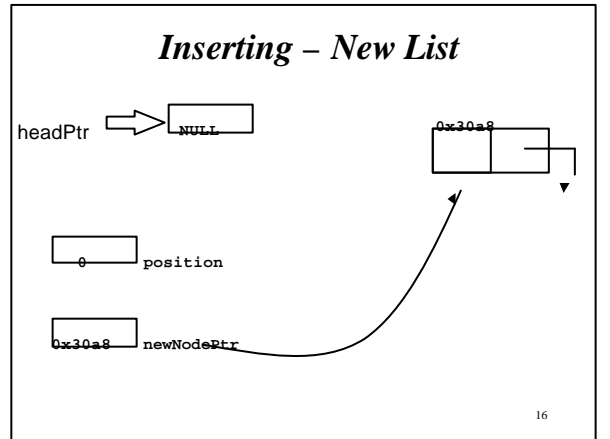
12

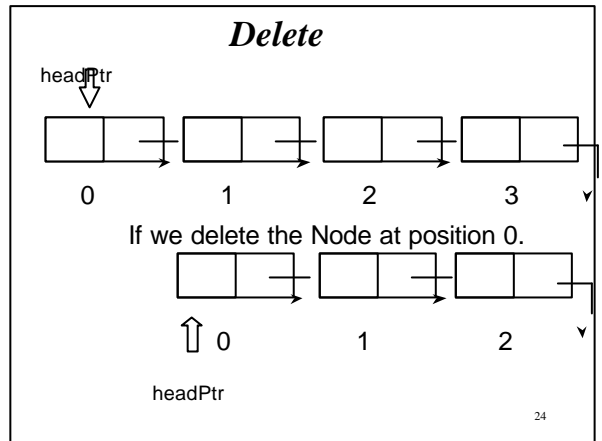
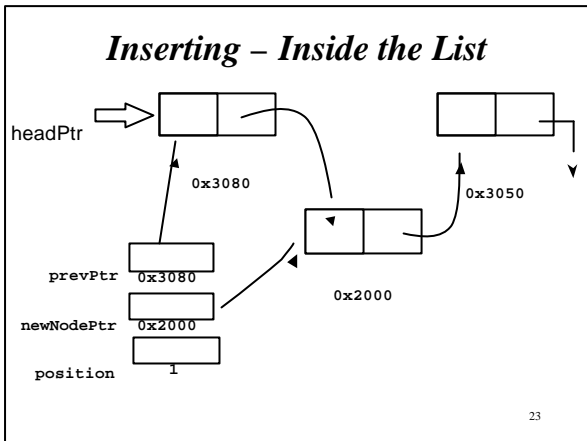
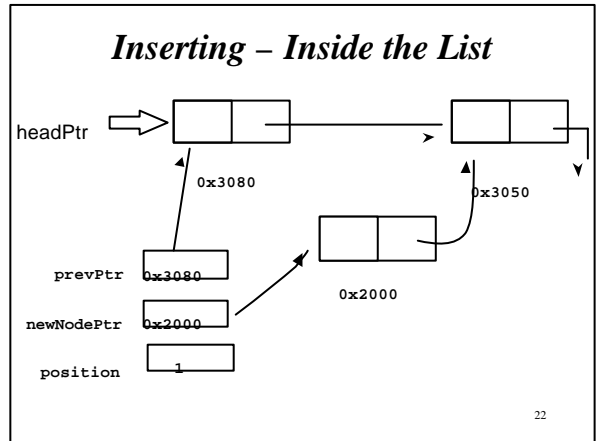
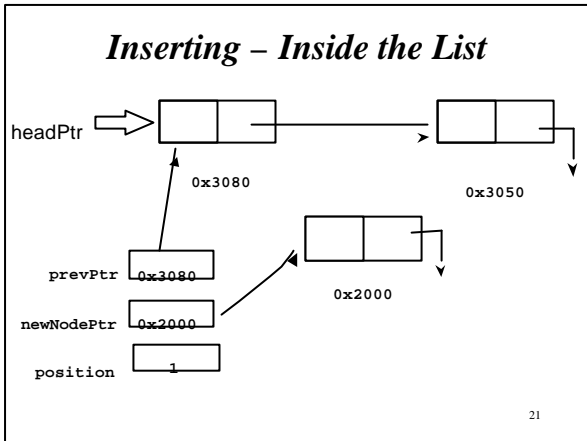
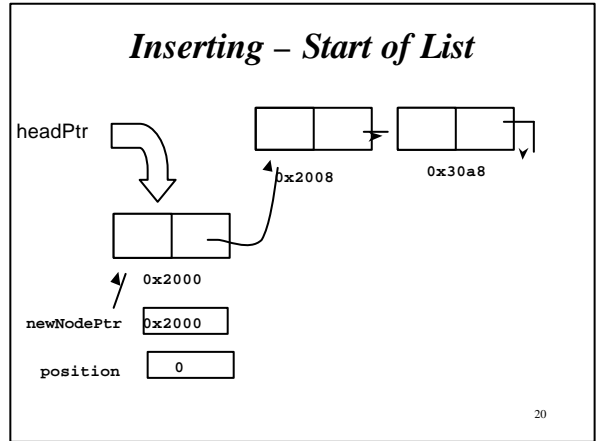
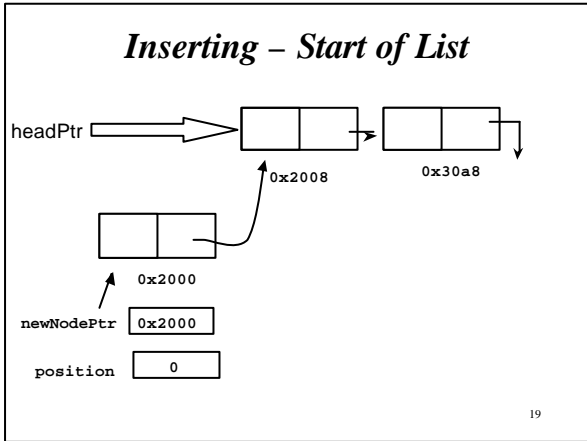


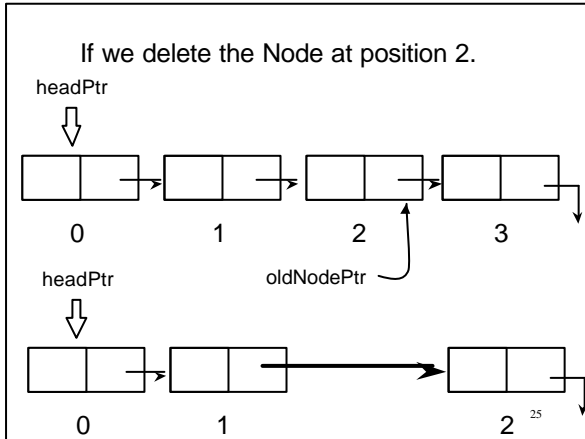
```
void insertItem(List* listPtr, float item, int position)
{
    Node* newNodePtr = makeNode(item);
    Node* prevPtr = NULL;

    if (position == 0)
    {
        newNodePtr->nextPtr = listPtr->headPtr;
        listPtr->headPtr = newNodePtr;
    }
    else
    {
        prevPtr = setPosition(listPtr, position-1);
        newNodePtr->nextPtr = prevPtr->nextPtr;
        prevPtr->nextPtr = newNodePtr;
    }
    listPtr->count++;
}
```

15





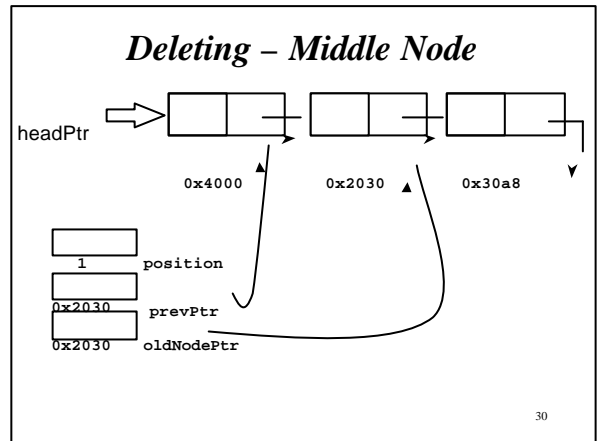
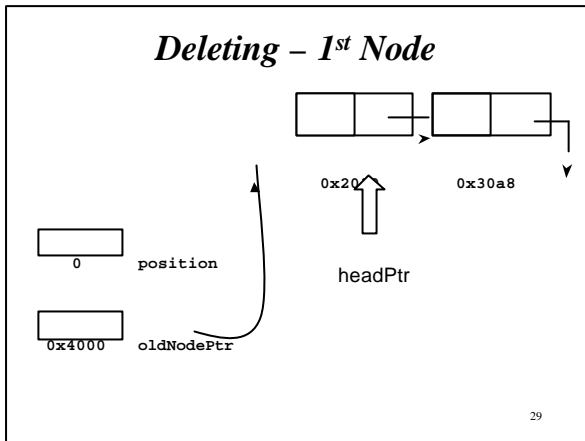
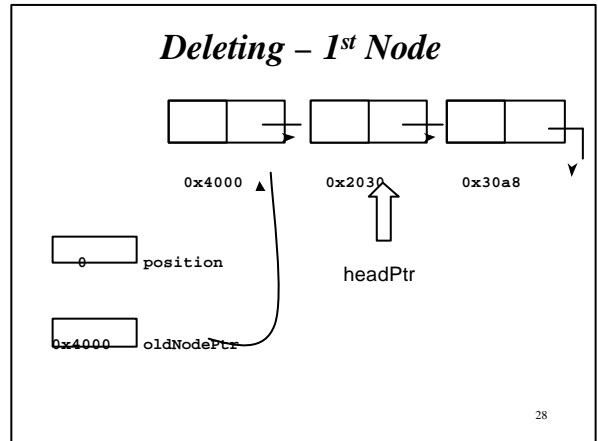
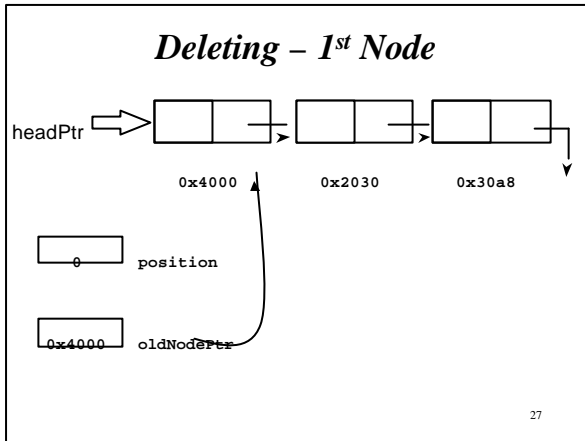


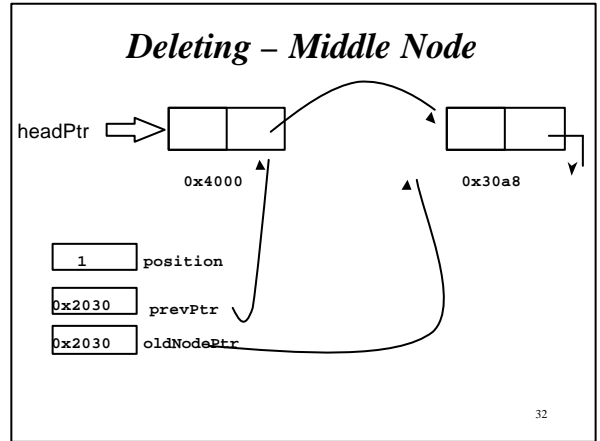
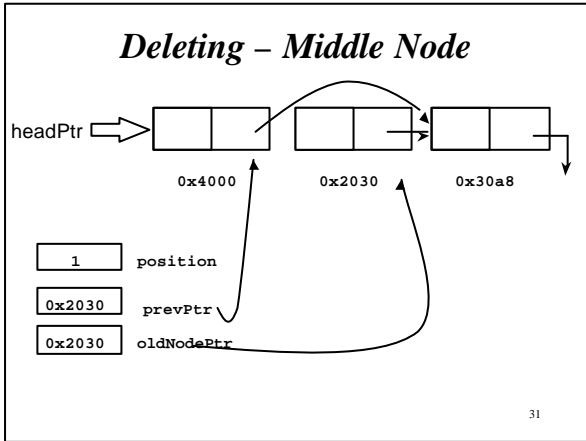
```

void deleteNode(List* listPtr, int position)
{
    Node* oldNodePtr = NULL;
    Node* prevPtr = NULL;

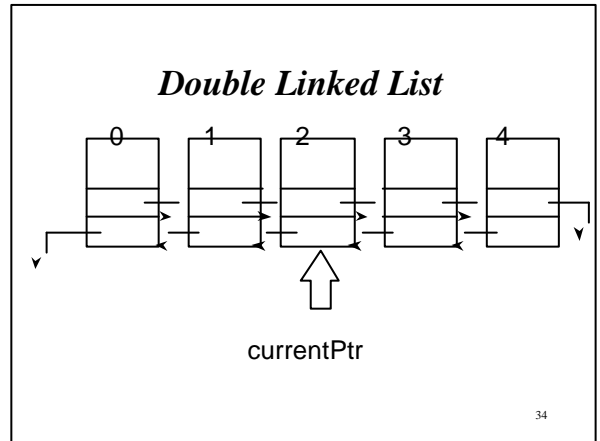
    if (listPtr->count > 0 && position < listPtr->count)
    {
        if (position == 0) {
            oldNodePtr = listPtr->headPtr;
            listPtr->headPtr = oldNodePtr->nextPtr;
        }
        else {
            prevPtr = setPosition(listPtr, position - 1);
            oldNodePtr = prevPtr->nextPtr;
            prevPtr->nextPtr = oldNodePtr->nextPtr;
        }
        listPtr->count--;
        free(oldNodePtr);
    }
    else {
        fprintf(stderr, "List is empty or invalid position.\n");
        exit(1);
    }
}
    
```

26





- ### Double Linked List Operations
- Go to a position in the list.
  - Insert an item in a position in the list.
  - Delete an item from a position in the list.
  - Retrieve an item from a position.
  - Replace an item at a position.
  - Traverse a list, *in both directions*.
- 33



```

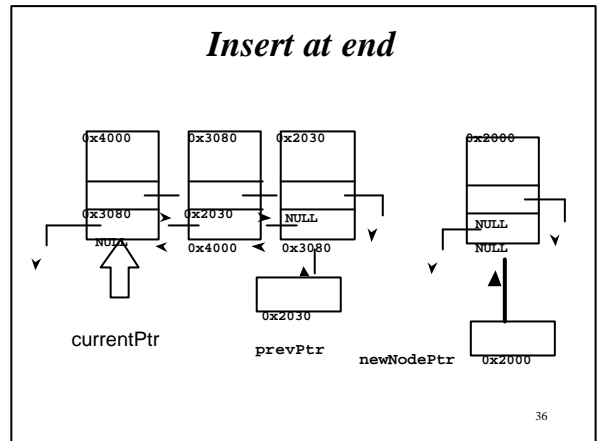
struct DoubleLinkNodeRec
{
    float          value;
    struct DoubleLinkNodeRec* nextPtr;
    struct DoubleLinkNodeRec* previousPtr;
};

typedef struct DoubleLinkNodeRec Node;

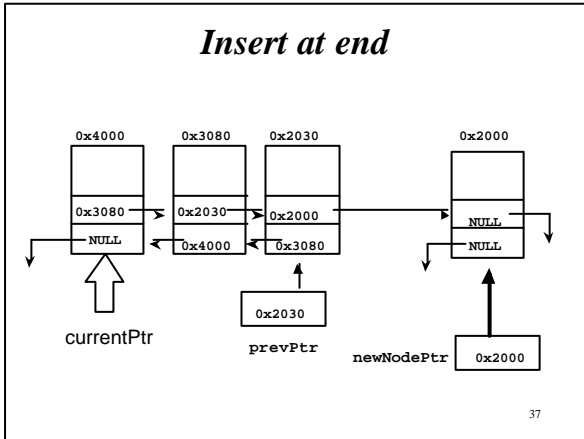
struct DoubleLinkedList
{
    int    count;
    Node*  currentPtr;
    int    position;
};

typedef struct DoubleLinkedListRec DoubleLinkedList;
    
```

35

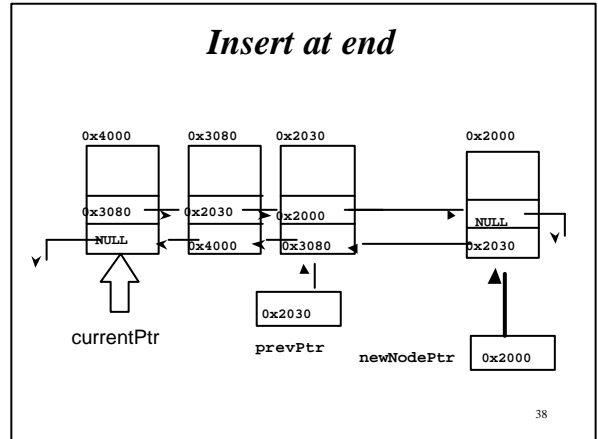


*Insert at end*



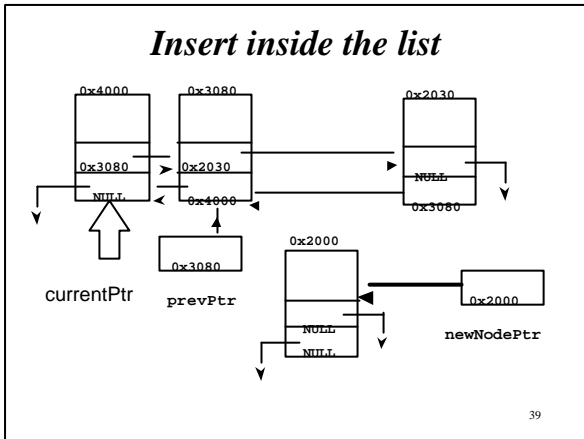
37

*Insert at end*



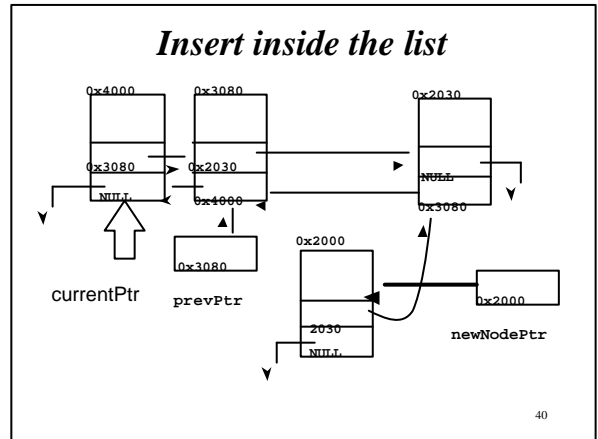
38

*Insert inside the list*



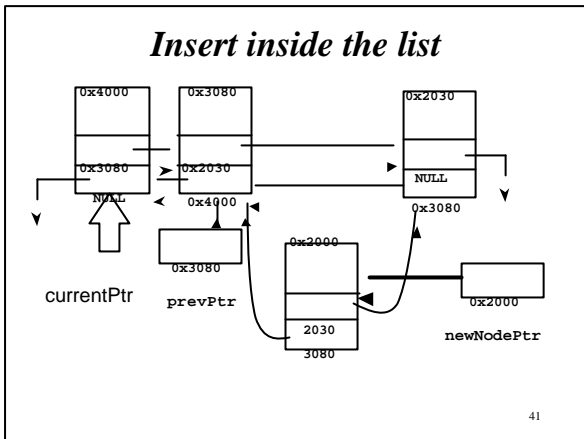
39

*Insert inside the list*



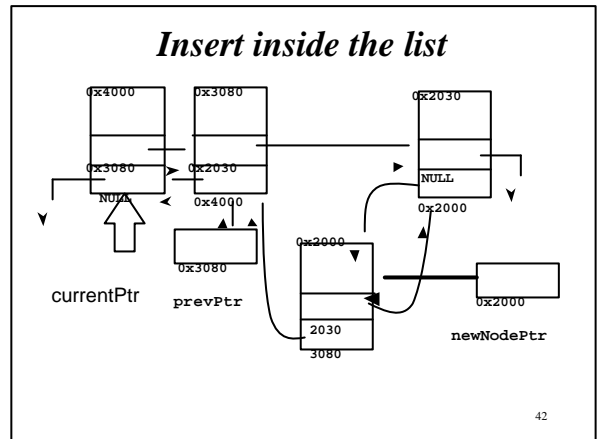
40

*Insert inside the list*

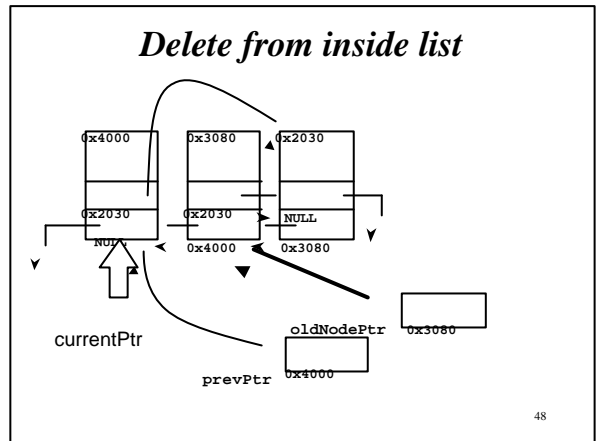
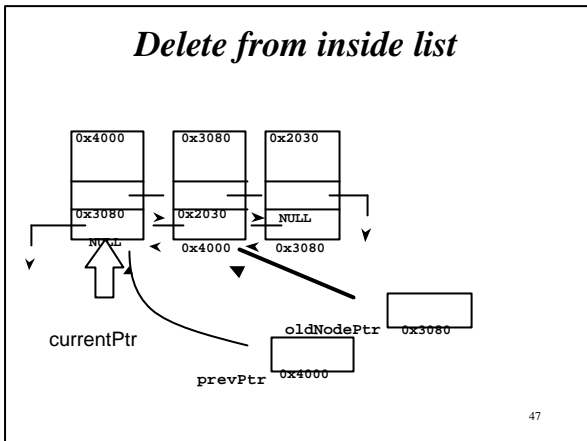
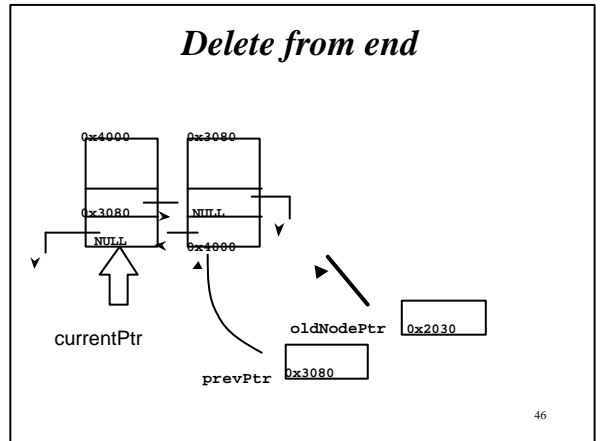
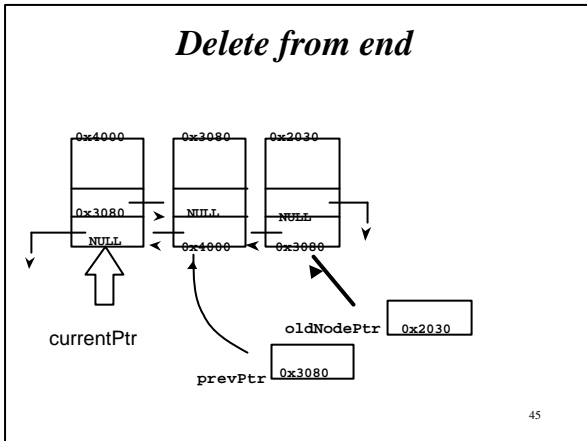
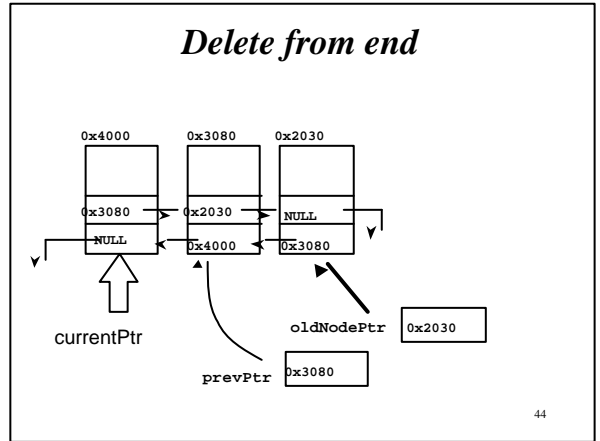
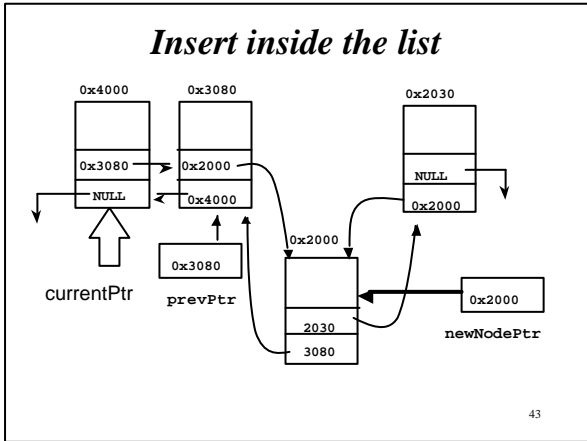


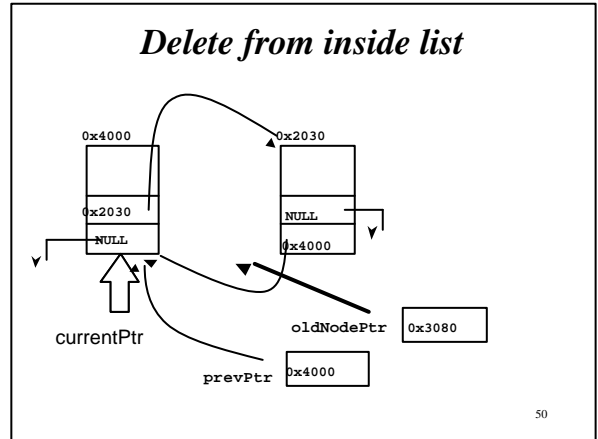
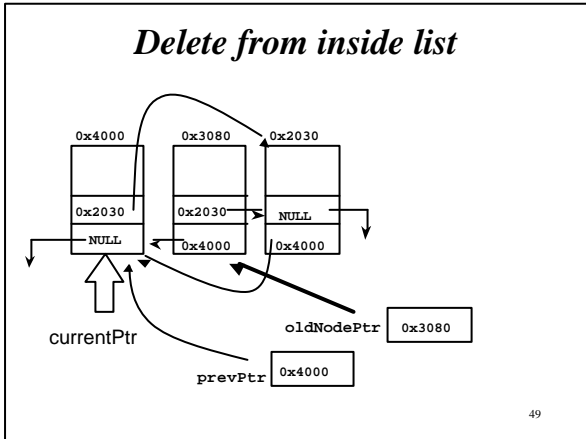
41

*Insert inside the list*



42





**Revision**

- Linked List.
- Benefits of different implementations.
- Double Linked List.

**Revision: Reading**

- Kruse 5.2.2 - 5.2.5
- Standish 2.4 - 2.5

**Preparation**

Next lecture: *Elementary Algorithms*

- Read Chapter 6 in Kruse et al.

51