CX1107
Data Structures and Algorithms

Linked Lists and Its Implementation

College of Engineering

School of Computer Science and Engineering

Dr. Loke Yuan Ren

yrloke@ntu.edu.sg

N4-02B-69A

Overview of Today Lecture

1. **What is the linked list?**
2. **How to create a linked list?**
3. **How to use the linked list?**
4. **Why do you need a linked list?**

What is the linked list?

Memory Allocation

3 scenario

1. **Known the data size before compile**
2. **Known the data size at the beginning**
3. **Unknown the data size. The size can be increased or decreased over the time while the program is running**

1. **Static Data Allocation (in stack memory)**
2. **Dynamic Data Allocation (in heap memory)**
3. **Dynamic Data with linked list structure**

Memory Address	Name	Matric No
0x1000	John	0001
0x1004	Anna	0002
0x1008	Peter	0003
0x100C	Jane	0004
	…	…

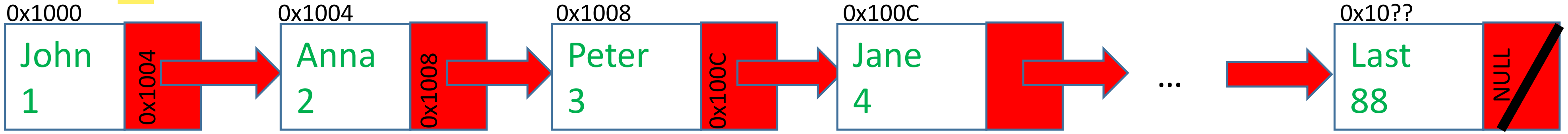
Linked List

- **Structure:** a collection of variables with different types:

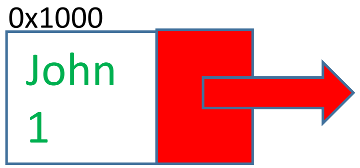
```
struct student{
        char Name[15];
        int matricNo;
}
```

- **Self-referential structure:** a pointer member that points to a structure of the same structure type

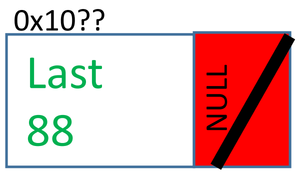
```
struct node{
        char Name[15];
        int matricNo;
        struct node *nextPtr;   //link
}
```



Linked List




```
struct node{
          char Name[15];
          int matricNo;
          struct node *nextPtr;   //link
}
```



1. **Each node contains data and link**

2. **The link contains the address of next node**

3. **If user knows the address of first node, the next node can be found from the link.**

4. **The link of the last node is a NULL pointer**

5. **The example is known as singly-linked list**

   - **There is only ONE link in the node**

How to create a linked list?

Definition and Declaration

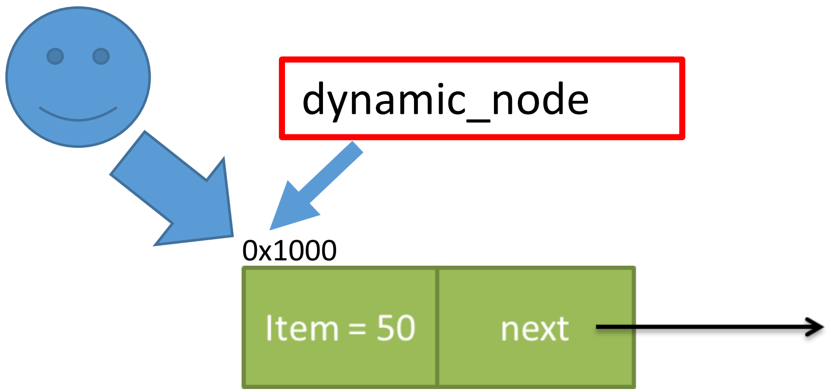
```c
#include <stdio.h>
#include <stdlib.h>

struct _listnode
{
    int item;
    struct _listnode *next;
};
typedef struct _listnode ListNode;

int main(void)
{
//static node
    ListNode static_node;
    static_node.data = 50;
    static_node.next = NULL;

//dynamic node
    ListNode* dynamic_node= (ListNode*) malloc(sizeof(ListNode));
    dynamic_node->data = 50;
    dynamic_node->next = NULL;
    free(dynamic_node);

    return 0;
}
```

- Define a self-referential structure, ListNode
- Dynamically allocate a ListNode node
- Free the node
- malloc() does not allocate NULL to the next link
- free() does memory deallocation but not delete
- After dynamic_node is freed, dynamic_node is **NOT** NULL

DEFINE AND CREATE A LINKED LIST

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   struct _listnode
5   {
6        int item;
7        struct _listnode *next;
8   };
9   typedef struct _listnode ListNode;
10
11  int main(void)
12  {
13  //static node
14       ListNode static_node;
15       static_node.data = 50;
16       static_node.next = NULL;
17
18  //dynamic node
19       ListNode* dynamic_node=malloc(sizeof(ListNode));
20       dynamic_node->data = 50;
21       dynamic_node->next = NULL;
22
23       ListNode* head = dynamic_node;
24       free(dynamic_node);
25
26       return 0;
27  }
```

- Create a head
    - ListNode* head;
- Multiple ListNode pointers can be created but the node just need to free once in the end



What is head after line 24?

Is There any bug?

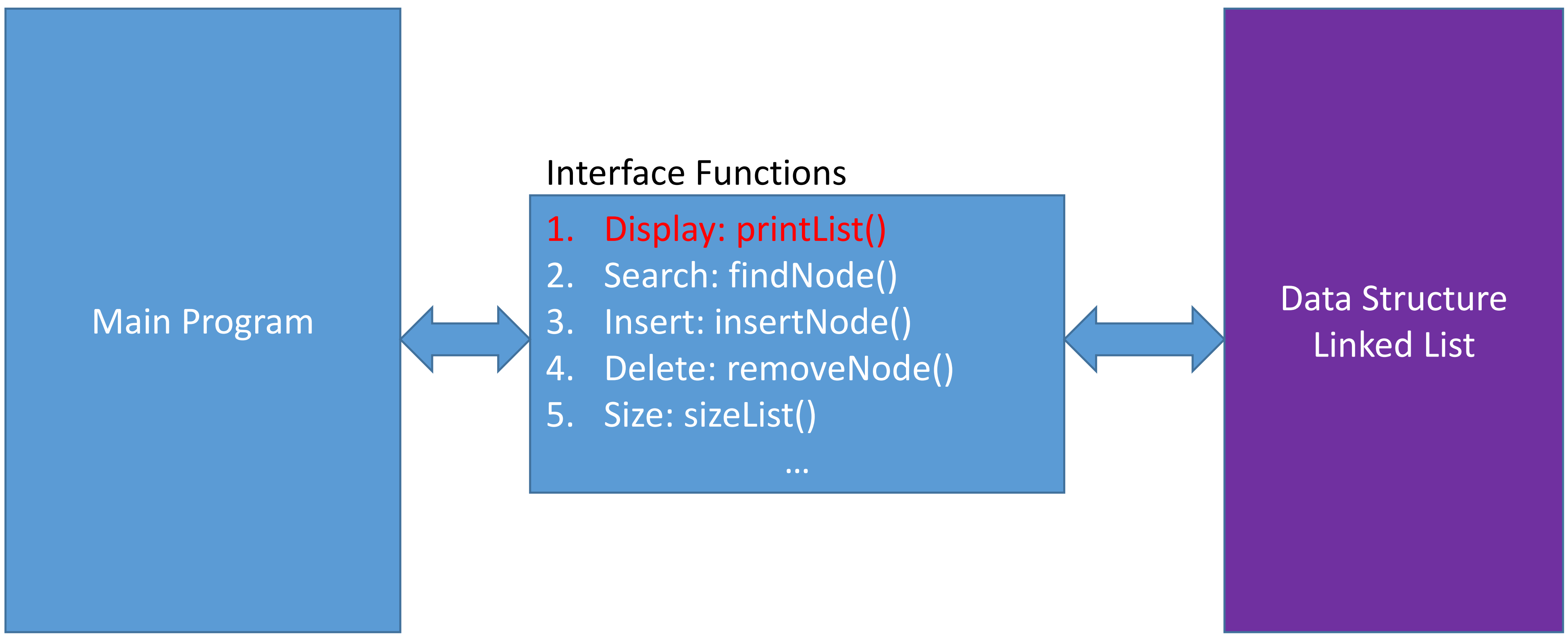
```
typedef struct node{
    int item; struct node *next;
} ListNode;

int main(){
    ListNode *head = NULL, *temp;
    int i = 0;

    while (scanf("%d", &i)){
        if (head == NULL){
            head = malloc(sizeof(ListNode));
            temp = head;
        }
        else{
            temp->next = malloc(sizeof(ListNode));
            temp = temp->next;
        }
        temp->item = i;
    }
    temp->next = NULL;
    return 0;
}
```

A. Yes

B. No

How to use the linked list?

HOW TO USE THE LINKED LISTS?

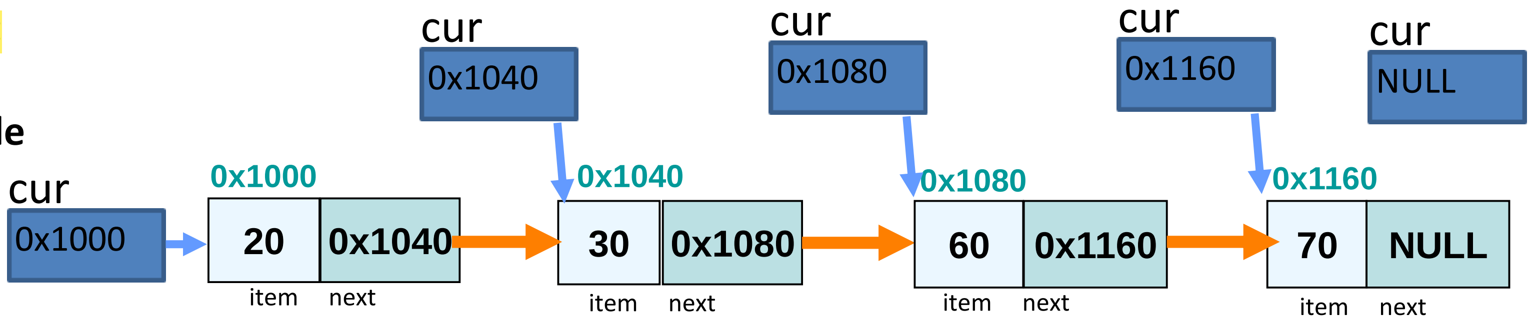


DISPLAY: printList()

`void printList(ListNode *cur);`

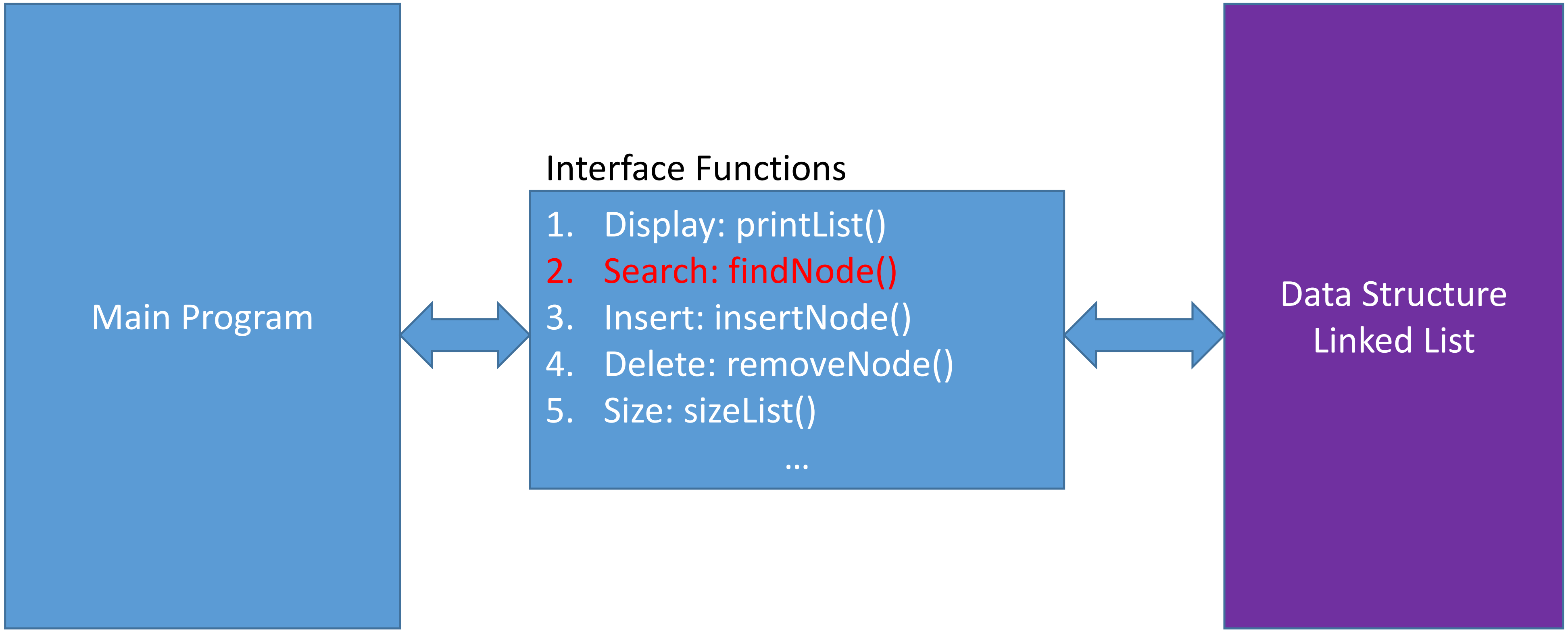
1. **Given the head pointer of the linked list**
2. **Print all items in the linked list**
3. **From first node to the last node**

```c
void printList(ListNode *cur){
    while (cur != NULL){
        printf("%d\n", cur->item);
        cur = cur->next;
    }
}
```




```
20
30
60
70
```

HOW TO USE THE LINKED LISTS?



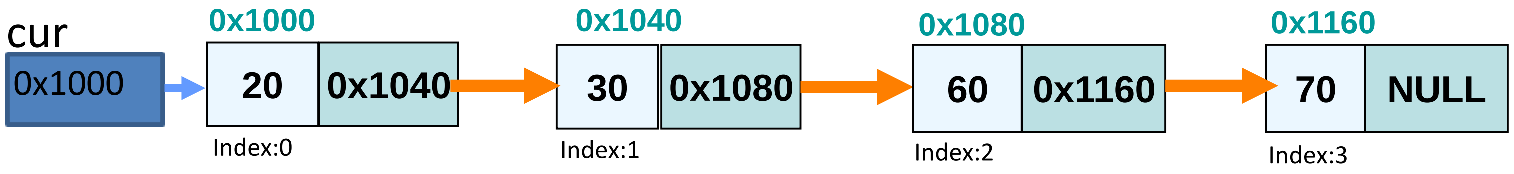
SEARCH: findNode()

```
ListNode* findNode(ListNode *cur, int i);
```

Looking for the i^{th} node in the list

1. **Given the head pointer of the linked list and index *i***
2. **Return the pointer to the i^{th} node**
3. **NULL will be return if index *i* is out of the range or the linked list is empty**

```
ListNode *findNode(ListNode* cur, int i)
{
    if (cur==NULL || i<0)
        return NULL;
    while(i>0){
        cur=cur->next;
        if (cur==NULL)
            return NULL;
        i--;
    }
    return cur;
}
```



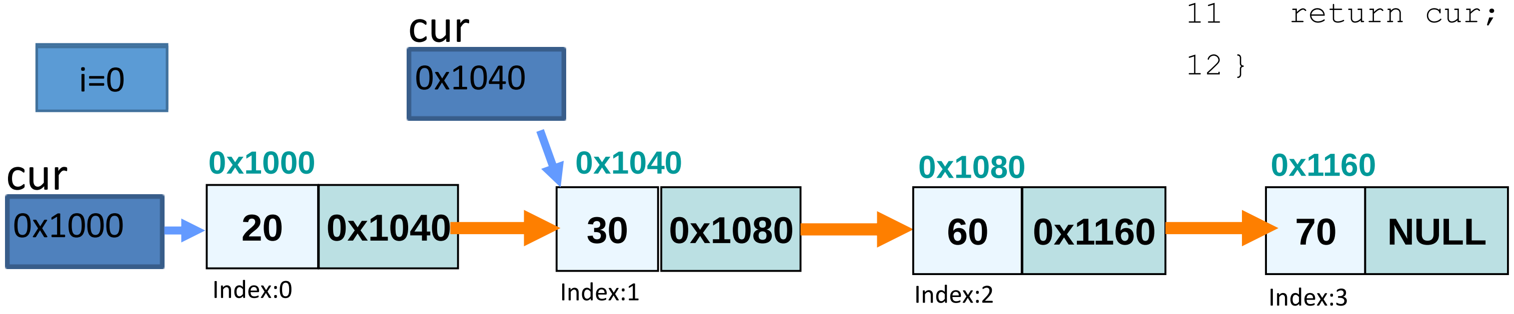
SEARCH: findNode()

```
ListNode* findNode(ListNode *cur, int i);
```

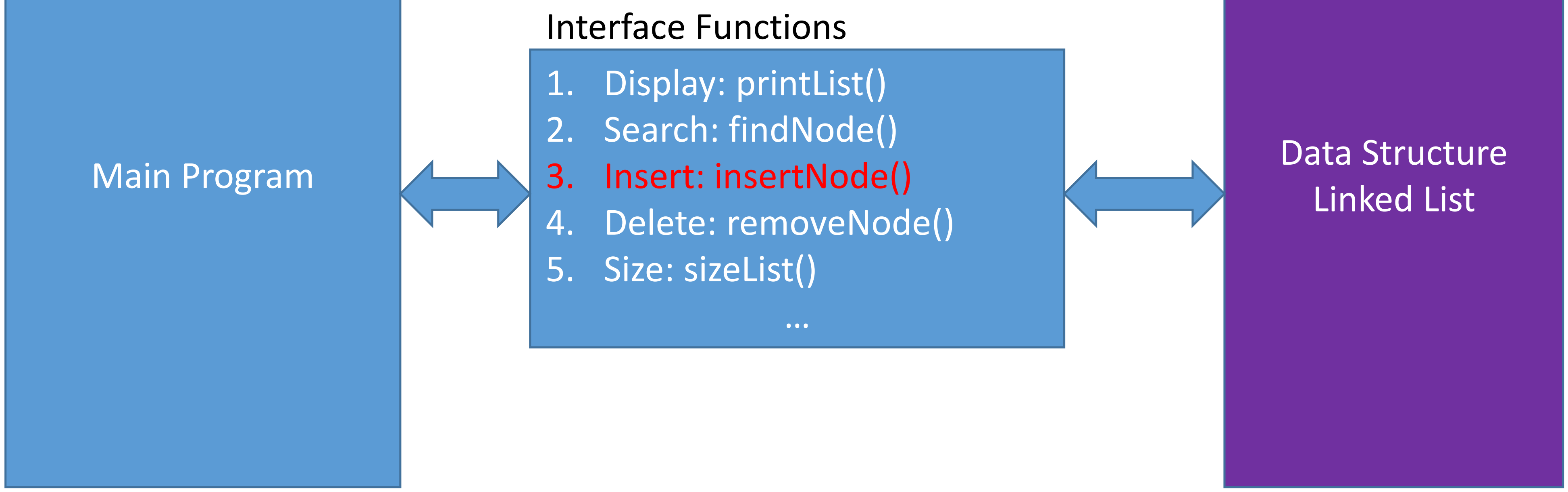
Looking for the *1st* node in the list

1. **Given the head pointer of the linked list and index *i=1***

```
ListNode *findNode(ListNode* cur, int i)
{
    if (cur==NULL || i<0)
        return NULL;
    while(i>0){
        cur=cur->next;
        if (cur==NULL)
            return NULL;
        i--;
    }
    return cur;
}
```



HOW TO USE THE LINKED LISTS?



INSERT: insertNode()

```
int insertNode(ListNode **ptrHead, int i, int item);
```

Add a node in the linked list

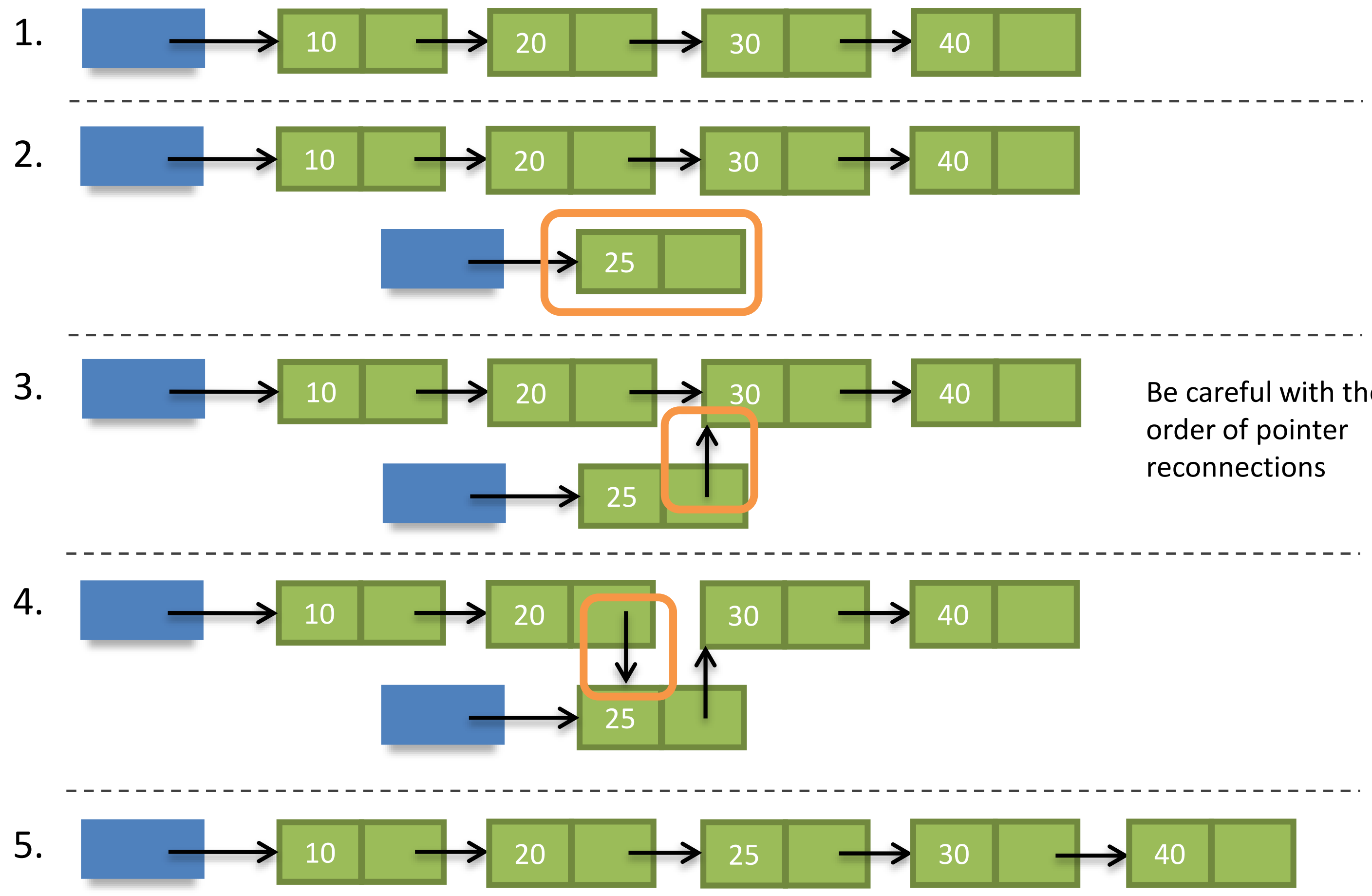
Given

- **the head pointer of the linked list**
- **index *i* where the node to be inserted**
- **the item for the node**

Return SUCCESS (1) or FAILURE (0)

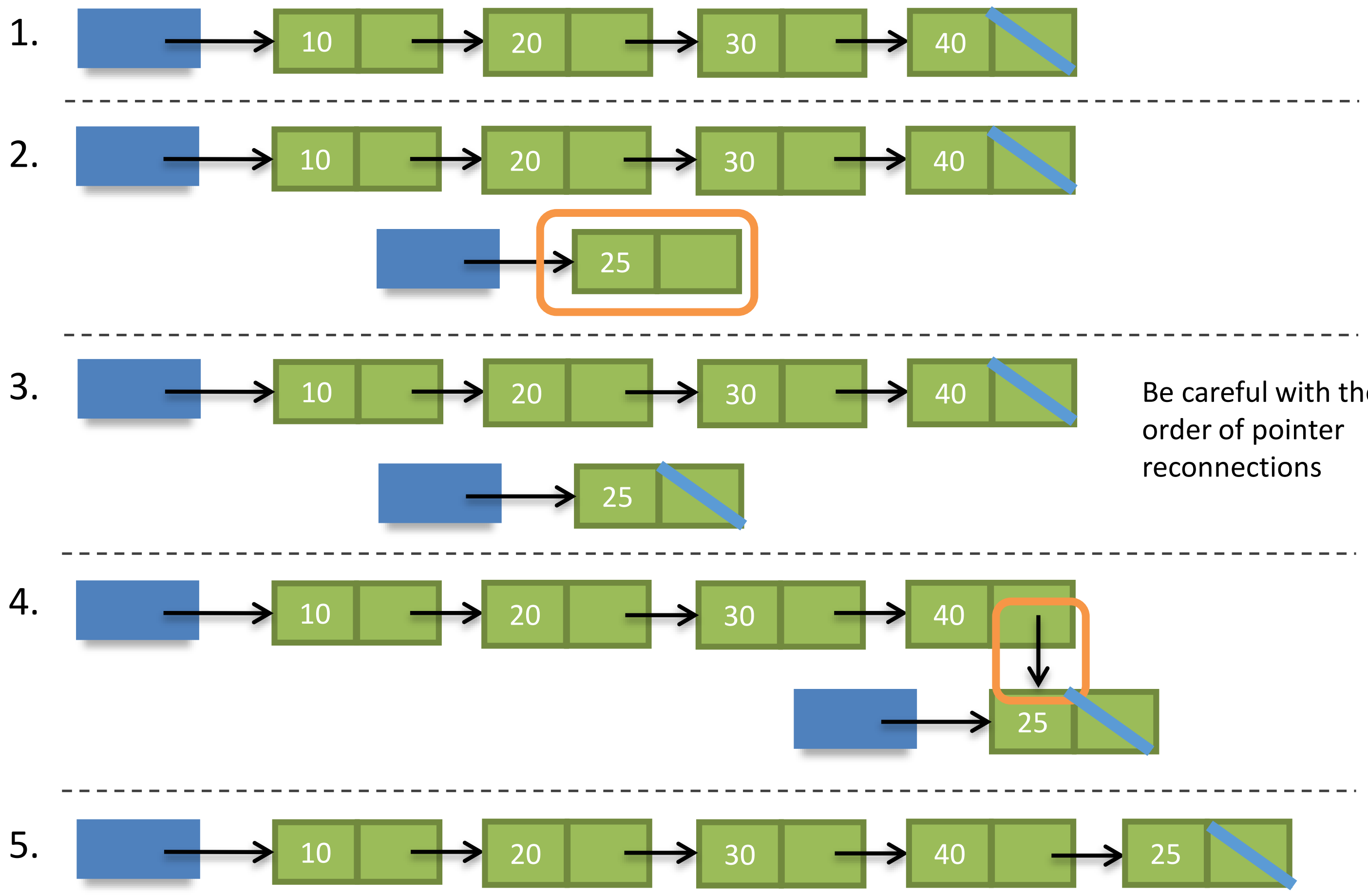
1. **Create a node by the given item**
2. **Insert the node at**
   1. **Front**
   2. **Middle**
   3. **Back**

INSERT A NODE IN MIDDLE



Be careful with the order of pointer reconnections

INSERT A NODE IN BACK



Be careful with the order of pointer reconnections

INSERT A NODE FRONT

- What is common to both special cases?
  - Empty list

```
head = malloc(sizeof(ListNode))
```

  - Inserting a node at index 0


```
// Save address of the first node
head = malloc(sizeof(ListNode))
head->next = [address of first node]
```

Need to modify the content of head pointer

INSERT: insertNode()

```
int insertNode(ListNode **ptrHead, int i, int item);
```

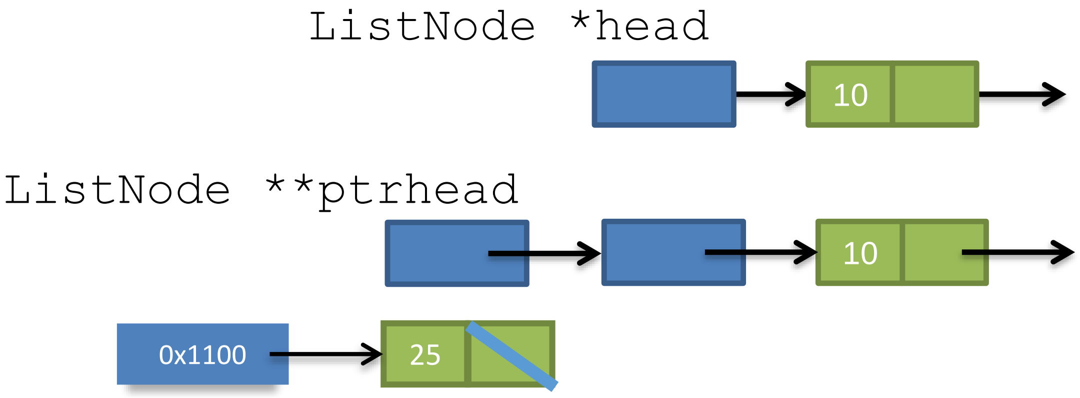
Add a node in the linked list

Given

- **the head pointer of the linked list**
- **index *i* where the node to be inserted**
- **the item for the node**

Return SUCCESS or FAILURE

1. **Create a node by the given item**
2. **Insert the node at**
   1. **Front**
   2. **Middle**
   3. **Back**



Memory Address	Data
0x1000	head=&ListNode 0x1080
0x1060	ptrhead=&head 0x1000
0x1080	item=10 next=0x10c0
0x1100	Item=25 next=NULL

INSERT: insertNode()

```
int insertNode(ListNode **ptrHead, int i, int item);
```

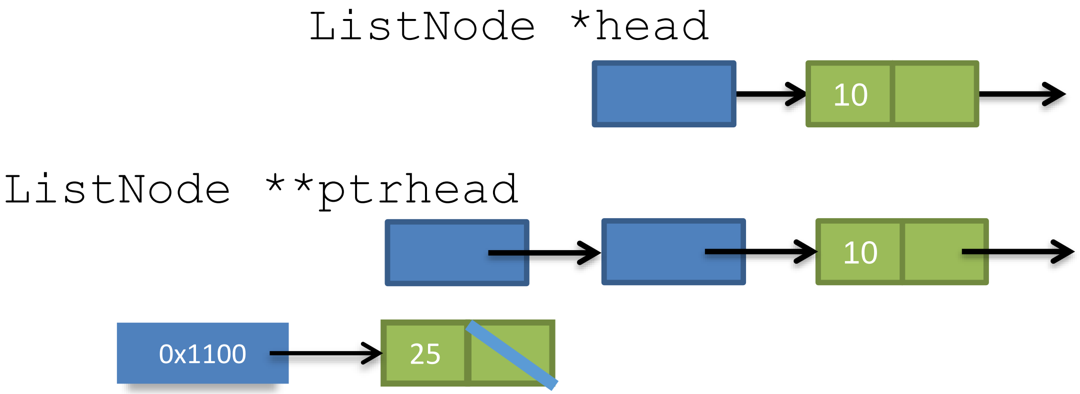
If we only pass head (0x1080) to insertNode(),

we only can access item=10 and next=0x10c0

we cannot modify the content in 0x1000.

When we are back from insertNode() to main(),

0x1000 still remain as 0x1080

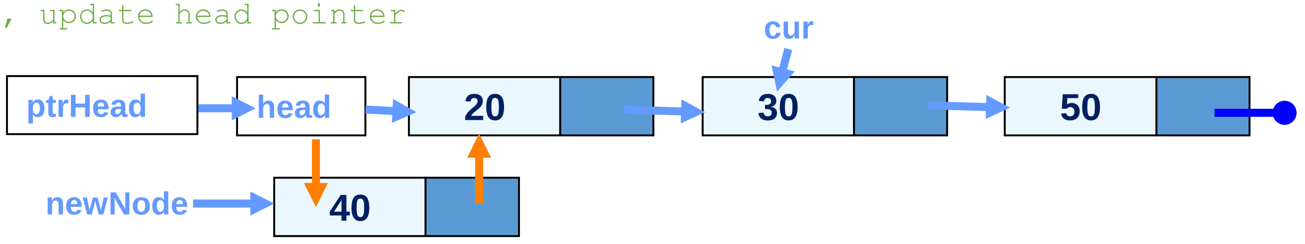


Memory Address	Data
0x1000	head=&ListNode 0x1080
0x1060	ptrhead=&head 0x1000
0x1080	item=10 next=0x10c0
0x1100	item=25 next=NULL

insertNode()

Is there any bug?

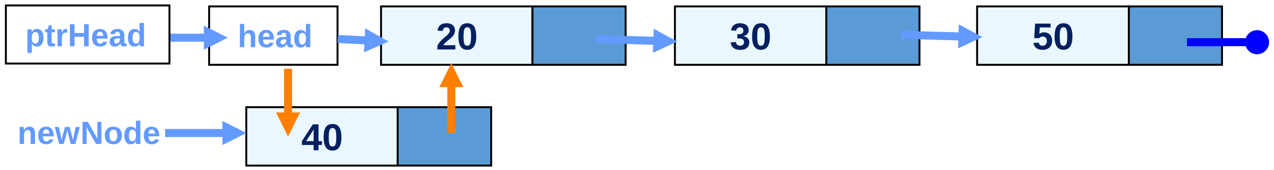
```
int insertNode(ListNode **ptrHead, int i, int item){
    ListNode  *cur, *newNode;
    // If empty list or inserting first node, update head pointer
    if (*ptrHead == NULL || i == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
        return 1;
    }
    // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    else if ((cur = findNode(*ptrHead, i-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = cur->next;
        cur->next = newNode;
        return 1;
    }
    return 0;
}
```



insertNode()

i=0 **item=40**

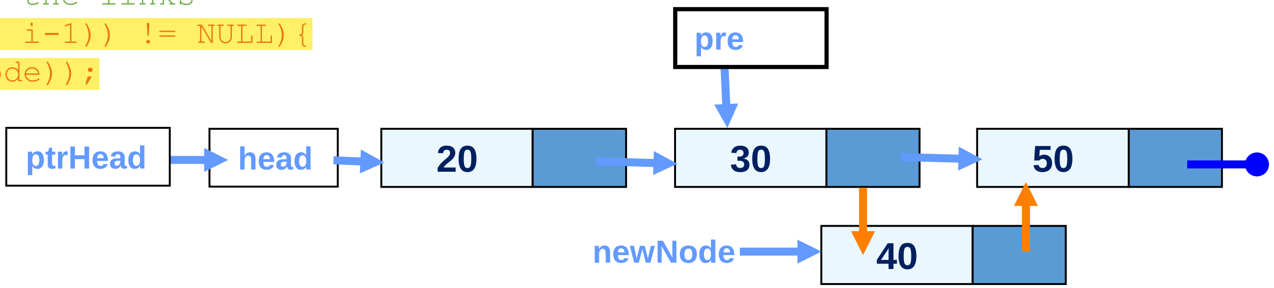
```
int insertNode(ListNode **ptrHead, int i, int item){
    ListNode  *pre, *newNode;
    // If empty list or inserting first node, update head pointer
    if (i == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
        return 1;
    }
    // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    else if ((pre = findNode(*ptrHead, i-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        return 1;
    }
    return 0;
}
```



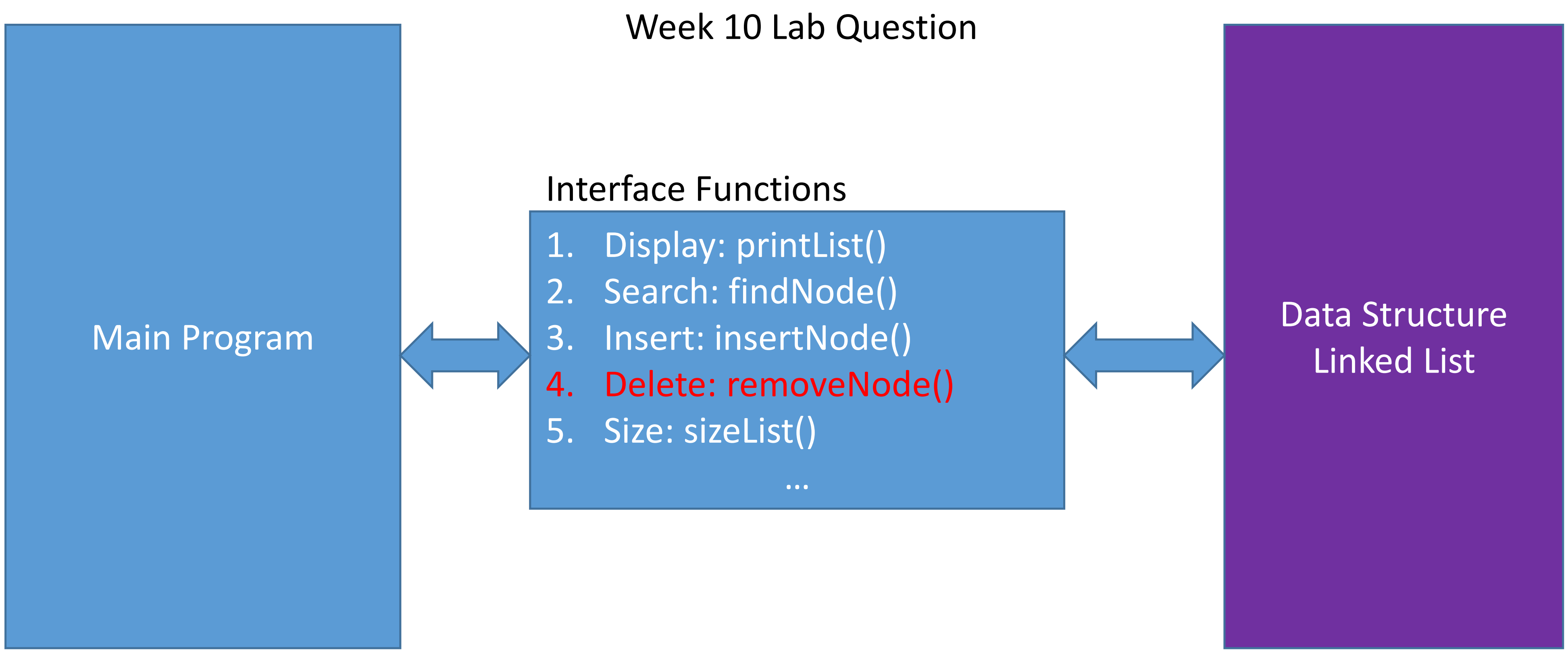
insertNode()

i=2 item=40

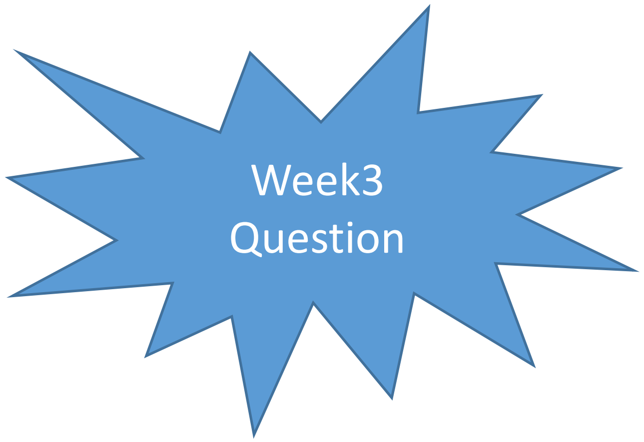
```
int insertNode(ListNode **ptrHead, int i, int item){
    ListNode  *pre, *newNode;
    // If empty list or inserting first node, update head pointer
    if (i == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
        return 1;
    }
    // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    else if ((pre = findNode(*ptrHead, i-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        return 1;
    }
    return 0;
}
```



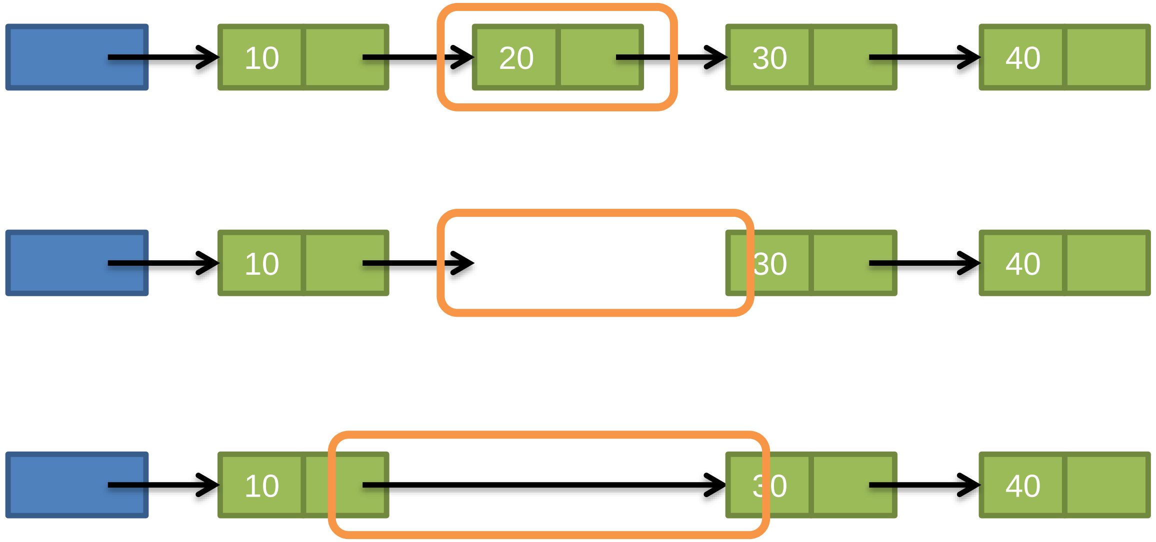
HOW TO USE THE LINKED LIST?



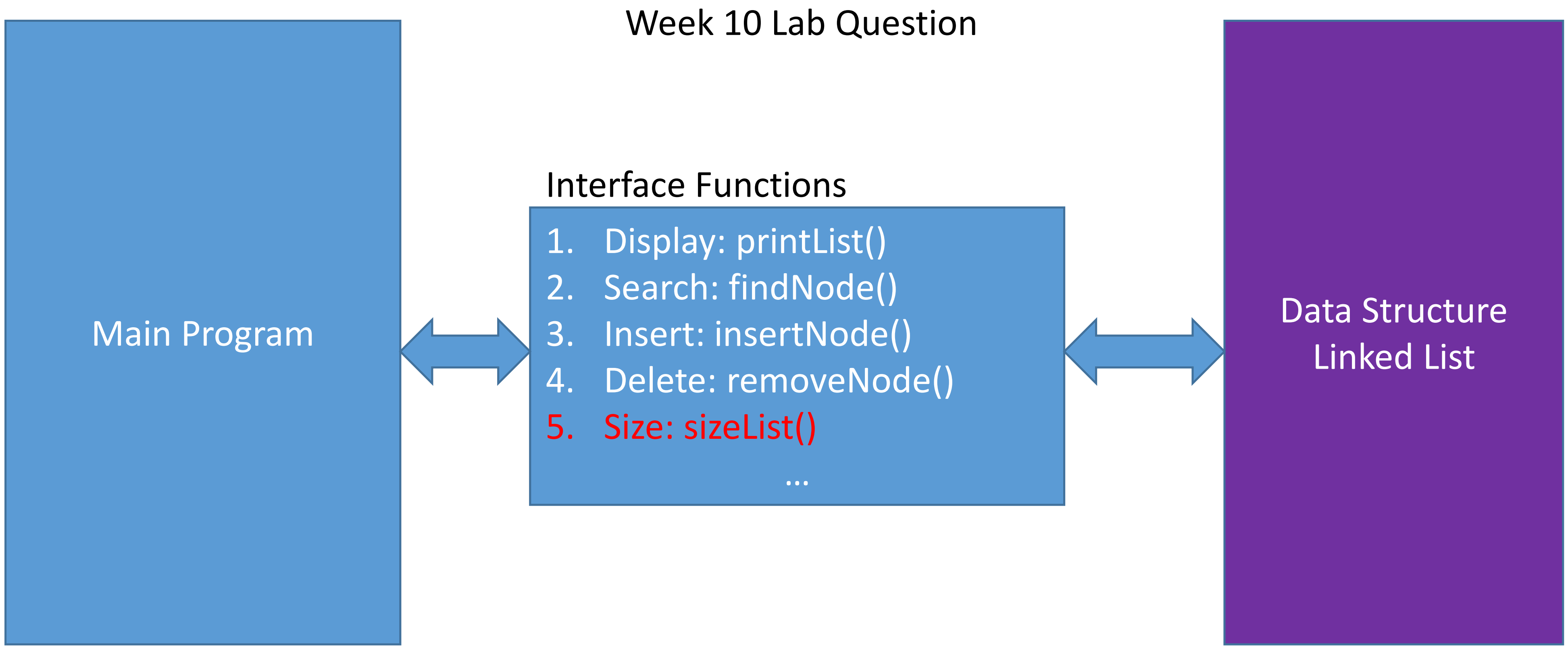
REMOVE A NODE: removeNode()



- Remember to free up any unused memory
- Remove a node at
  1. Front
  2. Middle
  3. Back



HOW TO USE THE LINKED LIST?



SIZE: sizeList()

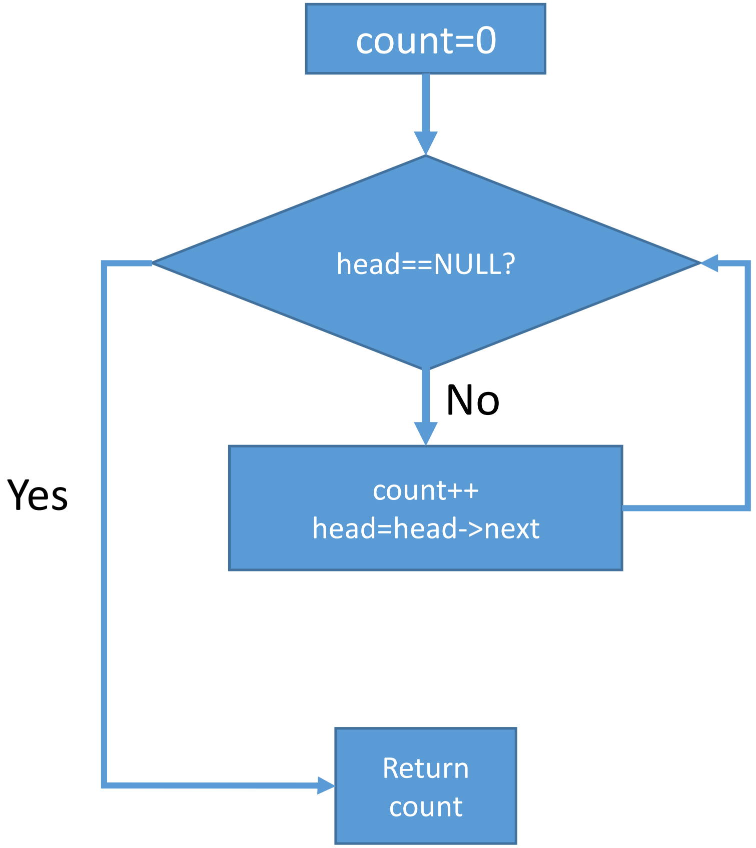
```
int sizeList(ListNode *head);
```

Given

- **the head pointer of the linked list**

Return the number of nodes in the linked list

1. **Declare a counter and initialize it to zero**
2. **Check the pointer whether is NULL or not**
3. **Increase the counter**
4. **Head move to next node**
5. **Repeat step 2**
6. **Return the counter**



SIZE: sizeList()

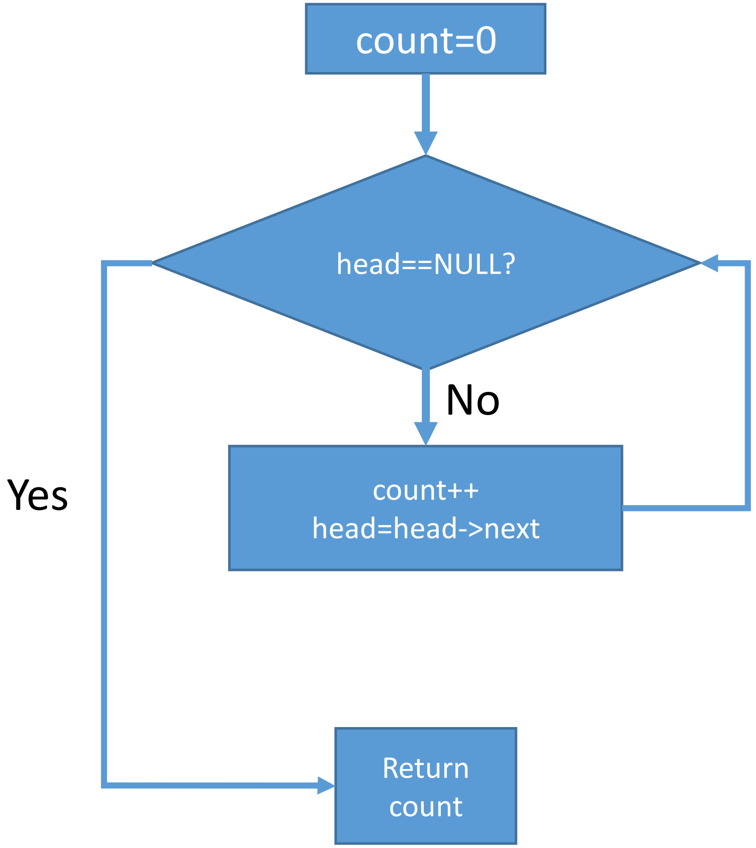
```
int sizeList(ListNode *head);
```

Given

- **the head pointer of the linked list**

Return the number of nodes in the linked list

```
int sizeList(ListNode *head){

    int count = 0;

    while (head != NULL){
        count++;
        head = head->next;
    }

    return count;
}
```



Why do you need a linked list?

LINKED LIST VS ARRAY

1. **Display: Both are similar**
2. **Search: Array is better**
3. **Insert and Delete: Linked List is more flexible**
4. **Size: Array is better**

Can we improve our sizeList()?

Interface Functions

1. Display: printList()
2. Search: findNode()
3. Insert: insertNode()
4. Delete: removeNode()
5. Size: sizeList()

...

```
void printList(ListNode *cur){
    while (cur != NULL){
        printf("%d\n", cur->item);
        cur = cur->next;
    }
}
```

```
int sizeList(ListNode *head){
    int count = 0;
    while (head != NULL){
            count++;
            head = head->next;
    }
    return count;
}
```

```
ListNode *findNode(ListNode* cur, int i){
   if (cur==NULL || i<0)
      return NULL;
   while(i>0){
      cur=cur->next;
      if (cur==NULL)
         return NULL;
      i--;
   }
   return cur;
}
```

```
int insertNode(ListNode **ptrHead, int i, int item){
    ListNode  *pre, *newNode;
    if (i == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
        return 1;
    }
    else if ((pre = findNode(*ptrHead, i-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        return 1;
    }
    return 0;
}
```

ARRAYS VS. LINKED LISTS

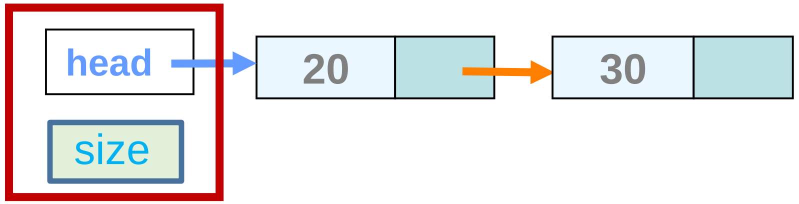
- **Arrays**
  - Efficient random access
  - Difficult to expand, re-arrange
  - When inserting/removing items in the middle or at the front, computation time scales with size of list
  - Generally a better choice when data is immutable
- **Linked lists (dynamic-pointer-based and static-array-based)**
  - "Random access" can be implemented, but more inefficient than arrays
  - cost of storing links, only use internally.
  - Easy to shrink, rearrange and expand (but array-based linked list has a fixed size)
  - Insert/remove operations only require fixed number of operations regardless of list size. no shifting
- Know when to choose an array vs a linked list

CAN WE IMPROVE OUR sizeList()?

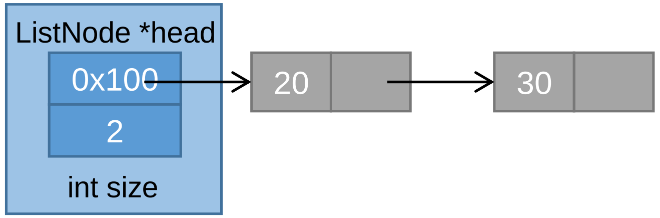
```
int sizeList(ListNode *head){
    int count = 0;
    while (head != NULL){
            count++;
            head = head->next;
    }
    return count;
}
```

- Solution:
  - Define another C struct, LinkedList
  - Wrap up all elements that are required to implement the Linked List data structure

```
typedef struct _linkedlist{
    ListNode *head;
    int size;
} LinkedList;
```




```
int sizeList(LinkedList ll){
    return ll.size;
}
```



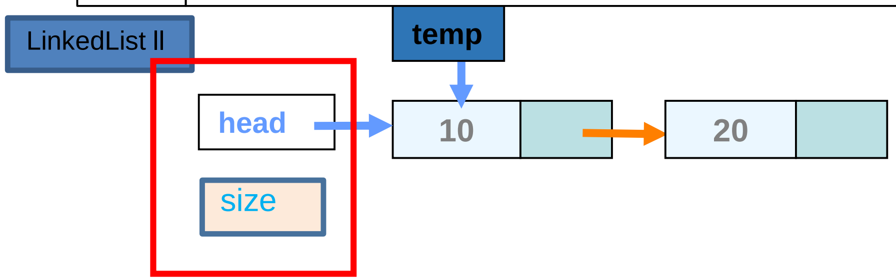
- Remember to change size when adding/removing nodes

LINKED LIST FUNCTIONS USING LinkedList STRUCT

- Original function prototypes:
  - void printList(ListNode *head);
  - ListNode *findNode(ListNode *head);
  - int insertNode(ListNode **ptrHead, int i, int item);
  - int removeNode(ListNode **ptrHead, int i);
- New function prototypes:
  - **void printList(LinkedList ll);**
  - **ListNode *findNode(LinkedList ll, int i);**
  - **int insertNode(LinkedList *ll, int index, int item);**
  - **int removeNode(LinkedList *ll, int i);**

NEW printList()

```
void printList(ListNode *cur){
    while (cur != NULL){
        printf("%d\n", cur->item);
        cur = cur->next;
    }
}
```




```
typedef struct _linkedlist{
   ListNode *head;
   int size;
}LinkedList;
```

```
void printList(LinkedList ll){
    ListNode *temp = ll.head;

    while (temp != NULL){
        printf("%d\n", temp->item);
        temp = temp->next;
    }
}
```

NEW findNode()

```
ListNode *findNode(ListNode* cur, int i){
   if (cur==NULL || i<0)
      return NULL;
   while(i>0){
      cur=cur->next;
      if (cur==NULL)
         return NULL;
      i--;
   }
   return cur;
}
```

```
typedef struct _linkedlist{
   ListNode *head;
   int size;
}LinkedList;
```

```
ListNode *findNode(LinkedList ll, int i){
    ListNode *temp = ll.head;
    if (cur==NULL || i < 0|| i >ll.size)
        return NULL;

    while (i > 0){
        temp = temp->next;
        if (temp == NULL)
            return NULL;
        i--;
    }
    return temp;
}
```

HOMEWORK

```
typedef struct _linkedlist{
   ListNode *head;
   int size;
}LinkedList;
```

```
int insertNode(ListNode **ptrHead, int i, int item){
    ListNode  *pre, *newNode;
    if (i == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
        return 1;
    }
    else if ((pre = findNode(*ptrHead, i-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        return 1;
    }
    return 0;
}
```

```
int insertNode(LinkedList *ll, int i, int item){
















}
```