ITP20001/ECE 20010 Data Structures

Data Structures

Chapter 4

- *singly linked list*
- *linked stacks and queues*
- *polynomials (and sparse matrices)*
- *doubly linked list*

Chapter 4 – Linked lists

Singly linked list implementation:

```
/**  linkedList.h
 * linkedList is a singly-linked implementation of the linked list ADT.
 * linkedList is a mutable data structure, which can grow at either end.
 * node is a node of a singly-linked list linkedList, used internally.
 * Each node has two fields:  one to an object or its data item,
 * and one to the next node in the list.
 */
  typedef  struct  node   *pNode;
  typedef struct node {
      int     item;
      pNode   next;
 } node;

typedef  struct  list  *pList;
typedef  struct  list {
    pNode        head;
    pNode        tail;
    int  size;
} list;
```

node

int	item
pNode	next

list

pNode	head
pNode	tail
int	size

Chapter 4 – Linked lists

Singly linked list implementation:

```
/**  linkedList.h
 * linkedList is a singly-linked implementation of the linked list ADT.
 * linkedList is a mutable data structure, which can grow at either end.
 * node is a node of a singly-linked list linkedList, used internally.
 * Each node has two fields:  one to an object or its data item,
 * and one to the next node in the list.
 */
  typedef  struct  node   *pNode;
  typedef struct node {
      int     item;
      pNode   next;
 } node;

typedef  struct  list  *pList;
typedef  struct  list {
    pNode        head;
    pNode        tail;
    int  size;
} list;
```

node

int	item
pNode	next

list

pNode	head
pNode	tail
int	size

```
pList newList();
pNode newNode(int item)
pNode newNodeX(pNode next, int item)
```

Chapter 4 – Linked lists

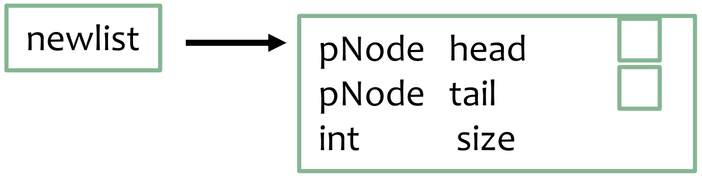
Singly linked list implementation:

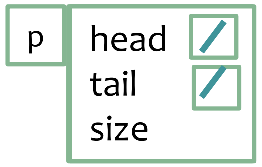
```
/**  Linkedlist.h */
pList newList();
void freeNode(pNode p);   // internal use
void freeList(pList p);   // internal use - frees a linked list
bool isEmpty(pList p);    // true if empty, false if no empty
int size(pList p);        // return size in the list
int getSize(pList p);     // internal use - count nodes in list

// inserts a node at front of linked
void insertFront(pList p, int item);

// internal use - inserts a node at the end
// scan the list to find the end, O(n)
void insertLast(pList p, int item);

// inserts a node at the end, O(1)
void insertAtTail(pList p, int itme);
// inserts a node at an index, 0 for front
void insertIndex(pList p, int item, int x);
// inserts a node in sorted order
void insertSorted(pList p, int item);

void deleteNode(pList p, int item);// deletes a node with the item
int search(pList p, int item);// returns index of item if found, -1 if not
void traverse(pList p);// shows all items in linked list
```

Chapter 4 – Linked lists

newList():

```
// linkedList.c

/**
 * This constructs singly linked list with a null pointer;
 */
pList  newList() {
    pList newlist  =  (pList)malloc(sizeof(list));
    assert(newlist!=NULL);

    newlist->head = NULL;
    newlist->tail = NULL;
    newlist->size = 0;

    return   newlist;
}
```



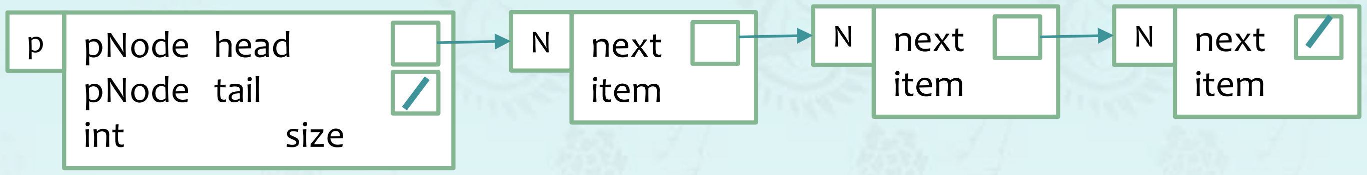


```
pList  mylist = newList();          // list  *mylist = newList();
```

Chapter 4 – Linked lists

Singly linked list implementation:

Challenge: Insert an item (not a node) at the beginning of the list p.



Chapter 4 – Linked lists

newNode() – First trial

```
/* linkedList.c */
/* node is used "internally" by the list.
 * newNode() constructs a list node referencing the data item or object;
 *
 * item - the data item or object
 * next - NULL
 * This function may return null if the memory cannot be allocated.
 */
pNode newNode(int item) {
    pNode  aNode = (pNode) malloc(sizeof(node));
    assert(aNode!=NULL);

    aNode->item = item;
    aNode->next = NULL;

    return aNode;
}
```

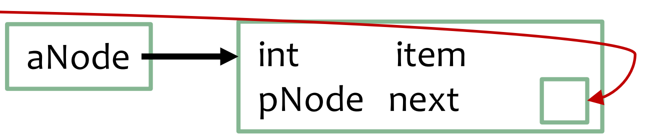
```
node
  int      item
  pNode    next
```

```
list
  pNode    head
  pNode    tail
  int      size
```

```
pNode  myNode = newNode(10);
```

newNodeX()

```
/* linkedList.c */
 * newNodeX() (with two parameters) constructs a list node referencing the
 * data item or object, whose next list node is to be "next".
 * next - reference to the next node. may be null.
 * This function may return null if the memory cannot be allocated.
 */
pNode newNodeX(pNode next, int item) {
    pNode  aNode = (pNode) malloc(sizeof(node));
    assert(aNode!=NULL);

    aNode->item = item;
    aNode->next = next;

    return aNode;
}
```

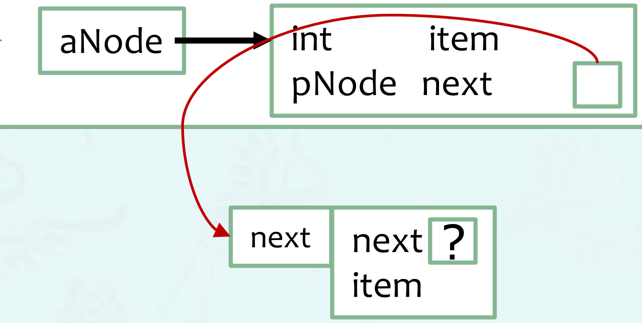


```
pNode  myNode = newNodeX(next, 10);
```

Chapter 4 – Linked lists

newNodeX()

```
/* linkedList.c */
 * newNodeX() (with two parameters) constructs a list node referencing the
 * data item or object, whose next list node is to be "next".
 * next - reference to the next node. may be null.
 * This function may return null if the memory cannot be allocated.
 */
pNode newNodeX(pNode next, int item) {
    pNode  aNode = (pNode) malloc(sizeof(node));
    assert(aNode!=NULL);

    aNode->item = item;
    aNode->next = next;

    return aNode;
}
```



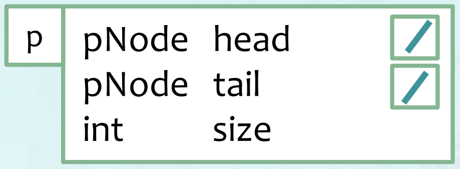
```
pNode  myNode = newNodeX(next, 10);
```

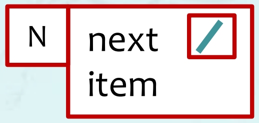
Chapter 4 – Linked lists

insertFront():

Challenge: Insert an item (not a node) at the beginning of the list p.

case 1: If head is null, the new node with the item becomes the head.





Recall: pNode **newNodeX(pNode next, int item)**

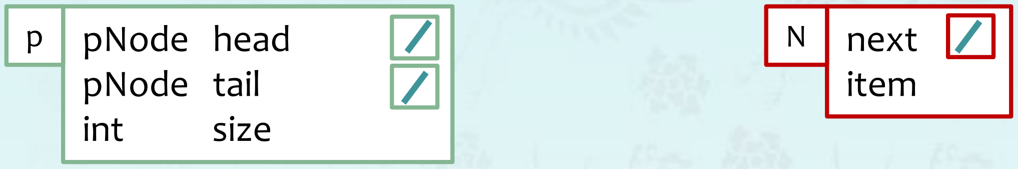
```
/* insertFront() inserts the object (item) at the beginning of the list p.
 * @param item - the object to be inserted.
 */
void insertFront(pList p, int item) {
   p->head = newNodeX(NULL, item);            // add it to the front
   p->size++;
}
```

Chapter 4 – Linked lists

insertFront():

Challenge: Insert an item (not a node) at the beginning of the list p.

case 1: If head is null, the new node with the item becomes the head.



Recall: pNode **newNodeX(pNode next, int item)**

```
/* insertFront() inserts the object (item) at the beginning of the list p.
 * @param item - the object to be inserted.
 */
void insertFront(pList p, int item) {
   p->head = newNodeX(NULL, item);
   p->size++;
}
```

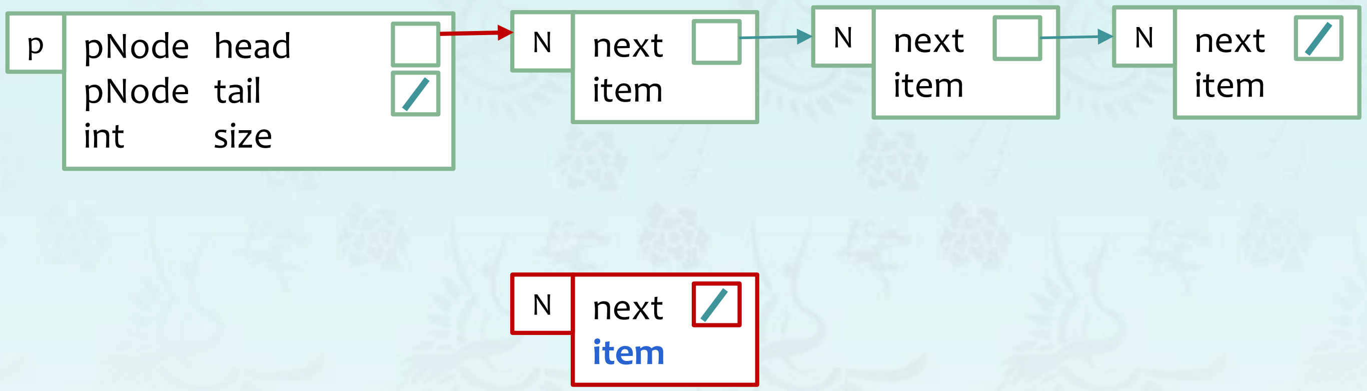
```
void insertFront(pList p, int item) {
   p->head = newNode(item);
   p->size++;
}
```

Chapter 4 – Linked lists

insertFront():

Challenge: Insert an item (not a node) at the beginning of the list p.

case 2: If head is not null, but the new node becomes the head anyway.

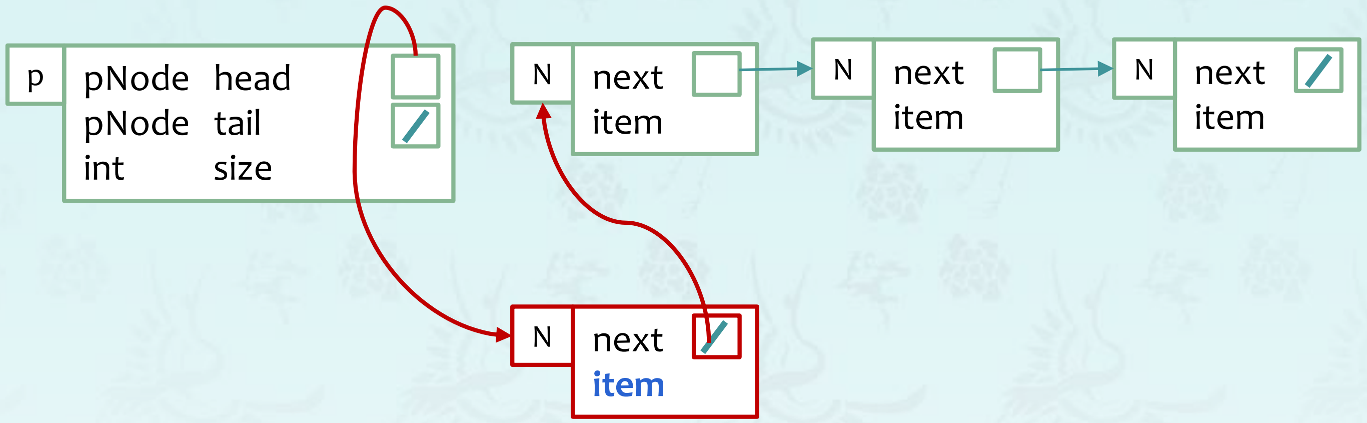


Chapter 4 – Linked lists

insertFront():

Challenge: Insert an item (not a node) at the beginning of the list p.

case 2: If head is not null, but the new node becomes the head anyway.

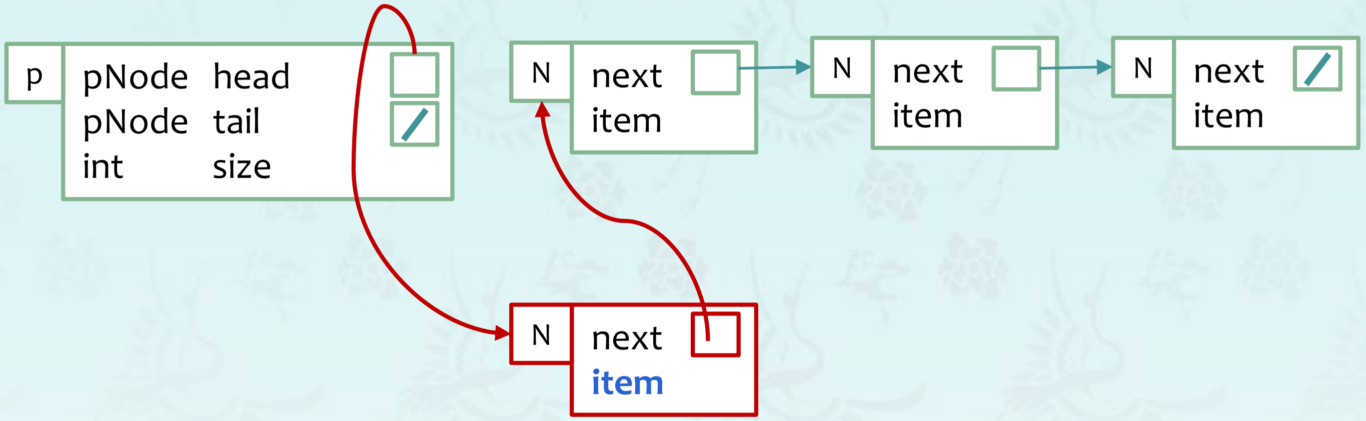


Chapter 4 – Linked lists

insertFront():

Challenge: Insert an item (not a node) at the beginning of the list p.

case 2: If head is not null, but the new node becomes the head anyway.



Recall: pNode **newNodeX(pNode next, int item)**

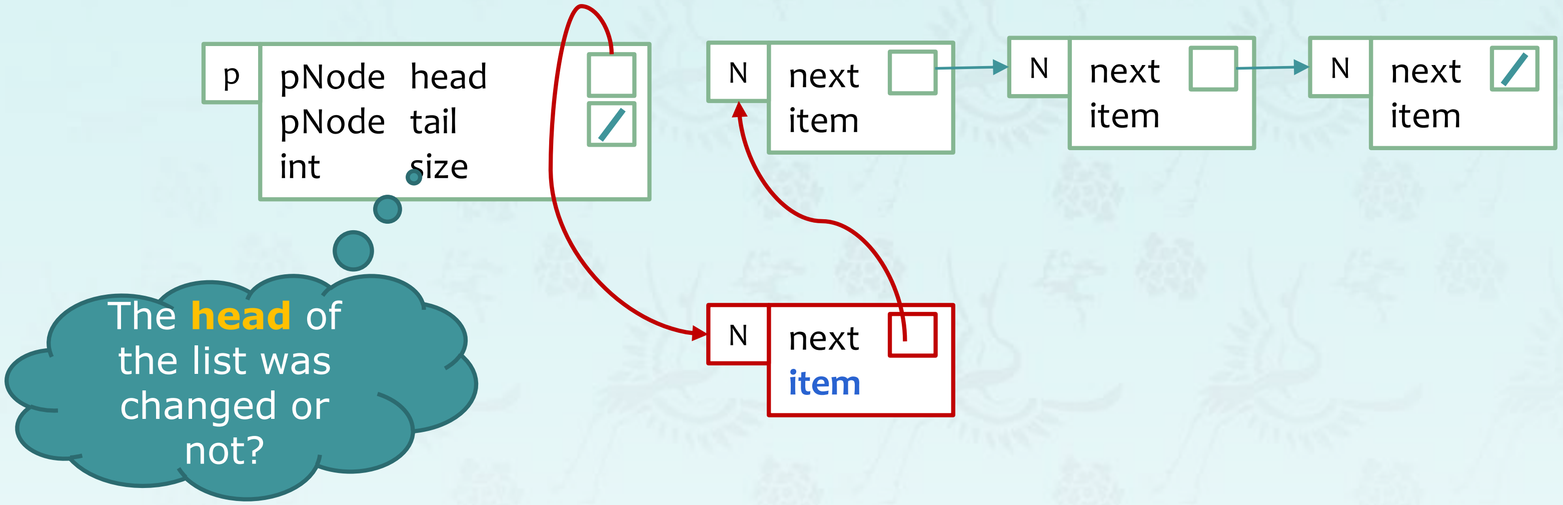
```
/* insertFront() inserts the object (item) at the beginning of the list p.
 * @param item - the object to be inserted.
 */
void insertFront(pList p, int item) {
   p->head = newNodeX(p->head, item)           // add it to the front
   p->size++;
}
```

Chapter 4 – Linked lists

insertFront():

Challenge: Insert an item (not a node) at the beginning of the list p.

case 2: If head is not null, but the new node becomes the head anyway.



```
/* insertFront() inserts the object (item) at the beginning of the list p.
 * @param item - the object to be inserted.
 */
void insertFront(pList p, int item) {
   p->head = newNodeX(p->head, item)          // add it to the front
   p->size++;
}
```

Chapter 4 – Linked lists

insertFront():

Challenge: Insert an item (not a node) at the beginning of the list p.

Observation:

case 1: If head is null, the new node with the item becomes the head.

```
/* insertFront() inserts the object (item) at the beginning of the list p.
 * @param item - the object to be inserted.
 */
void insertFront(pList p, int item) {
   p->head = newNodeX(NULL, item);              // add it to the front
   p->size++;
}
```

case 2: If head is not null, but the new node becomes the head anyway.

```
/* insertFront() inserts the object (item) at the beginning of the list p.
 * @param item - the object to be inserted.
 */
void insertFront(pList p, int item) {
   p->head = newNodeX(p->head, item)            // add it to the front
   p->size++;
}
```

Chapter 4 – Linked lists

insertFront():

Challenge: Insert an item (not a node) at the beginning of the list p.

Observation:

case 1: If head is null, the new node with the item becomes the head.

Since head in case 1 is null anyway, we can pass p->head instead of NULL, too.
Then two functions are exactly the same.
Therefore, **the function for the case 2 works for the case 1 as well.**

case 2: If head is not null, but the new node becomes the head anyway.

```
/* insertFront() inserts the object (item) at the beginning of the list p.
 * @param item - the object to be inserted.
 */
void insertFront(pList p, int item) {
   p->head = newNodeX(p->head, item)
   p->size++;
}
```

```
void insertFront(pList p, int item) {
   pNode node = newNode(item);
   node->next = p->head;
   p->head = node;
   p->size++;
}
```

Chapter 4 – Linked lists

insertLast():

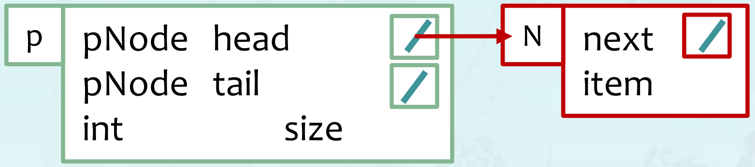
Challenge: Insert an item (not a node) at the end of the list p.

Chapter 4 – Linked lists

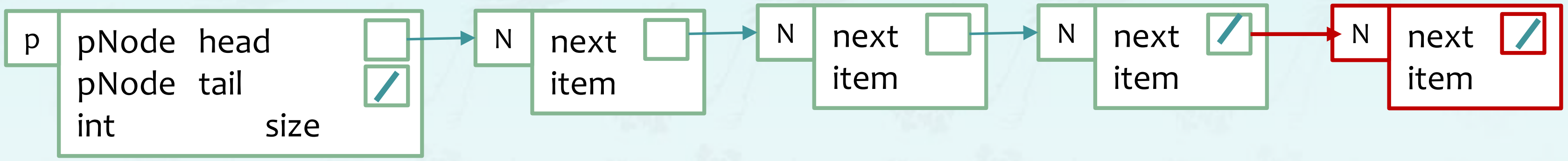
insertLast():

Challenge: Insert an item (not a node) at the end of the list p.

case 1: If head is null, the new node with the item becomes the head.



case 2: If head is not null, the new item added at the end of the list p.

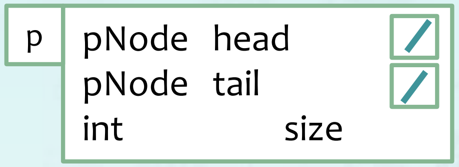


Chapter 4 – Linked lists

insertLast():

Challenge: Insert an item (not a node) at the end of the list p.

case 1: If head is null, the new node with the item becomes the head.



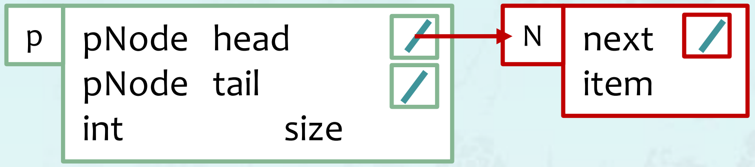
```
void insertLast(int item, pList p) {
    if (p->head == NULL) {
                =
    }
}
```

Chapter 4 – Linked lists

insertLast():

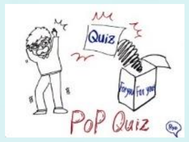
Challenge: Insert an item (not a node) at the end of the list p.

case 1: **If head is null,** the new node with the item becomes the head.



```
void insertLast(int item, pList p) {
    if (p->head == NULL)
        p->head = newNodeX(NULL, item);
}
```

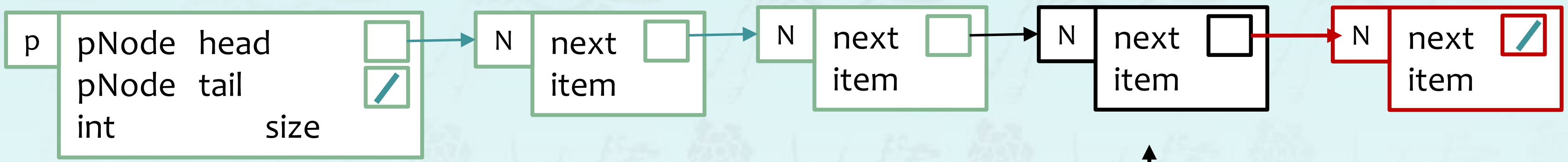
```
void insertLast(int item, pList p) {
    if (p->head == NULL)
        p->head = newNode(item);
}
```



insertLast():

Challenge: Insert an item (not a node) at the end of the list p.

case 2: If head is not null, the new item added at the end of the list p.



What is the challenge in case 2?

Finding the **last node** a challenge!

Recall: pNode **newNodeX(pNode next, int item)**

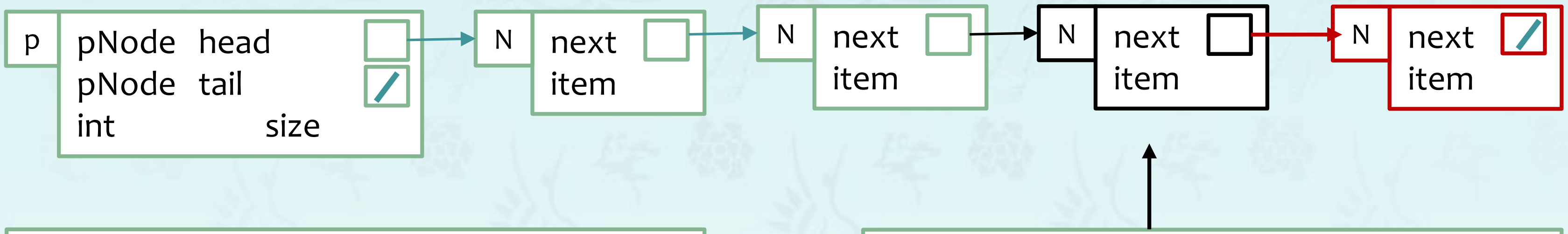
```
pNode node = p->head;
while (node->next != NULL)  // find the last node

….
```

Chapter 4 – Linked lists

insertLast():

Challenge: Insert an item (not a node) at the end of the list p.

case 2: If head is not null, the new item added at the end of the list p.



What is the challenge in case 2?

Finding the **last node** a challenge!

Recall: pNode **newNodeX(pNode next, int item)**

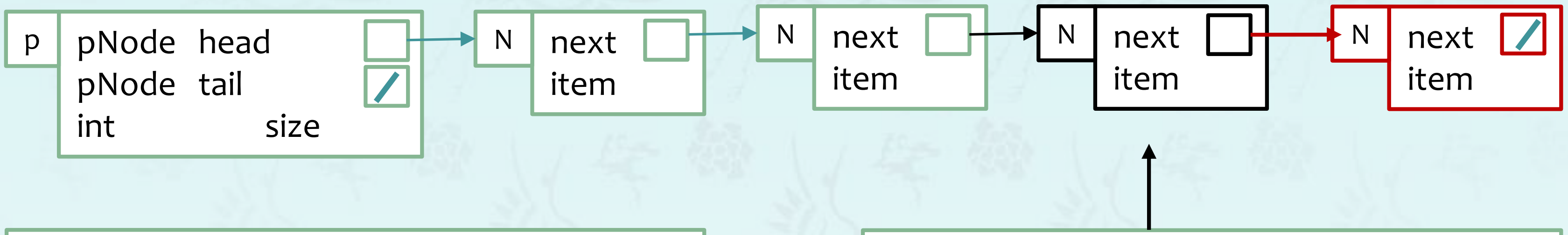
```
pNode node = p->head;
while (node->next != NULL)  // find the last node
        node = node->next;
node->next = newNodeX(NULL, item);
```

Chapter 4 – Linked lists

insertLast():

Challenge: Insert an item (not a node) at the end of the list p.

case 2: If head is not null, the new item added at the end of the list p.



What is the challenge in case 2?

Finding the **last node** a challenge!

Recall: pNode **newNodeX(pNode next, int item)**

```
pNode node = p->head;
while (node->next != NULL)  // find the last node
        node = node->next;
node->next = newNodeX(NULL, item);
```

```
pNode node = p->head;
while (node->next != NULL)
        node = node->next;
node->next = newNode(item);
```

insertLast()

```
/* linkedList.c
 * insertLast() inserts the object (item) at the end of the list p.
 * @param item - the object to be inserted.
 */
void insertLast(int item, pList p) {
    if (p->head == NULL) {

                Case 1

    }
    else {

                Case 2


    }
}
```

Chapter 4 – Linked lists

```
bool validate(pList p) {
    pNode curr, prev;
    int nodeCount = 0;
    bool validated = true;

    if (isEmptyList(p)) {
        if (p->size != 0) {
                printf("Its length(%d) should be 0.", p->size);
                validated = false;
        }
        if (p->tail != NULL) {
                printf("Its tail(%x) should be NULL.", p->tail);
                validated = false;
        }
        return;
    }
                curr = p->head;
                do {
                        prev = curr;
                        ++nodeCount;
                        curr = curr->next;
                } while (curr != NULL);

                if (nodeCount != p->size) {
                        printf("Its length(%d) is different from %d.", p->size, nodeCount);
                        validated = false;
                }
                if (prev != p->tail) {
                        printf("Its tail(%x) is different from %x.", p->tail, prev);
                        validated = false;
                }
                return validated;
}
```

Chapter 4 – Linked lists

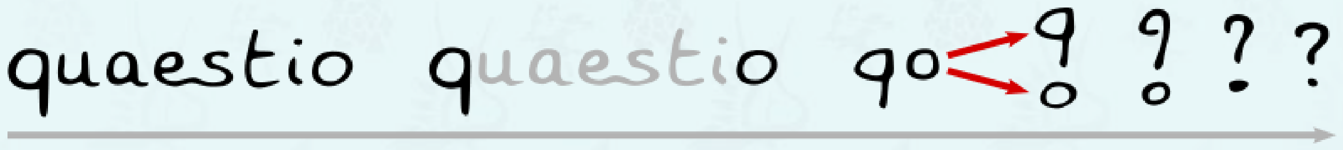
PSet 06:

Complete the singly linked list program, linkedList.c, that can be tested interactively.

- It is supposed to work like linkedList.exe executable provided.
- Your linkedList.c must be compatible with linkedList.h and LinkedListDriver.c provided.
- Don't change function signatures and/or return types in linkedList.h and linkedList.c files.

Summary

&


quaestio
quaestio
qo
?
?
?
?