COP 3502: Computer Science I
Spring 2004

– Note Set 16 –
Stacks and Queues: Linked List Implementations

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

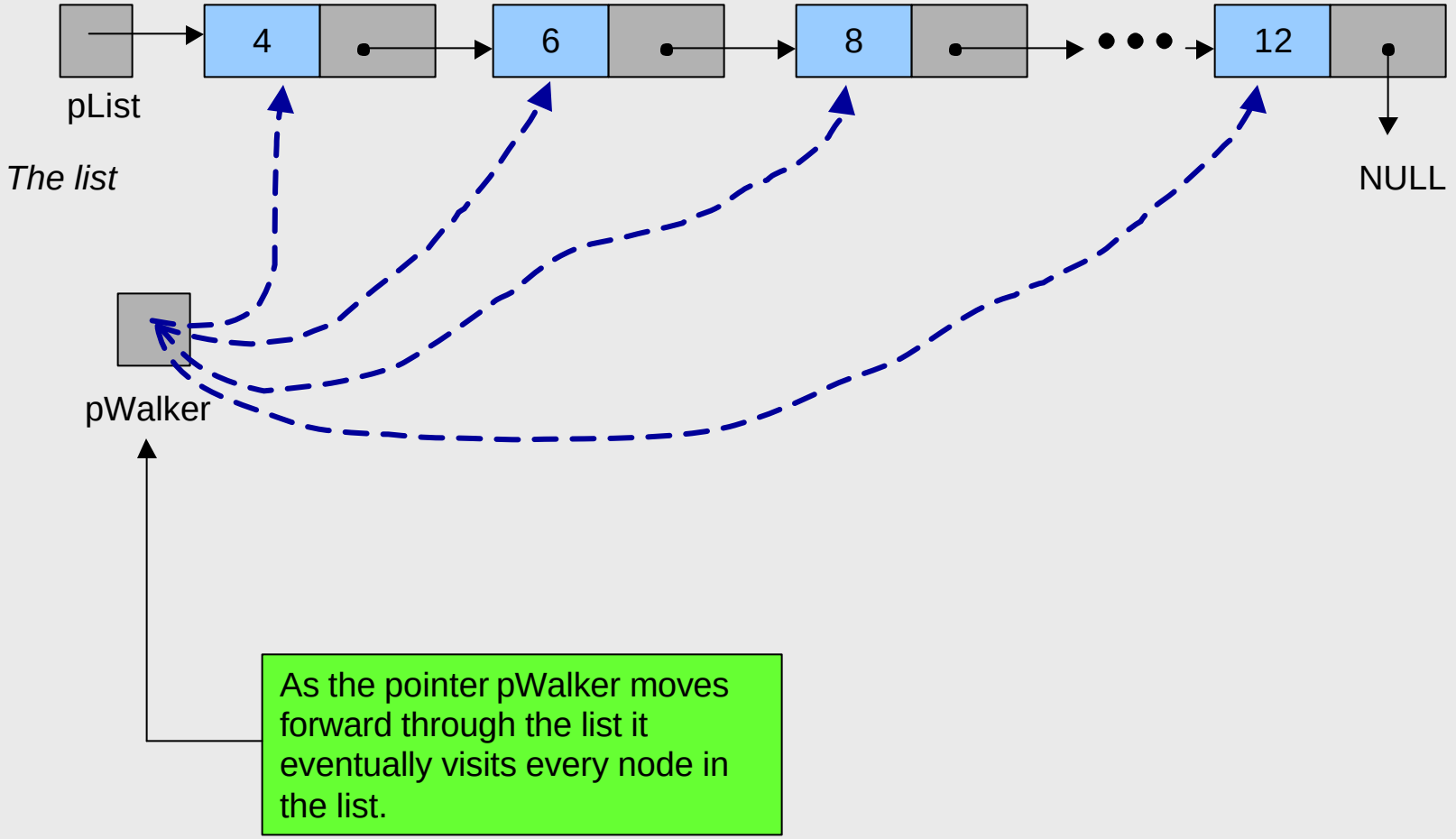
More Linked List Functions

- Before we look at specific stack and queue implementations using linked lists, we'll develop a few more functions that operate on general linked lists.
- For example, we haven't yet constructed a function that will traverse the links of a linked list and print out the values in the nodes as it encounters them.
- Neither have we developed a function that will create a linked list, so far we've just assumed that the lists we have been inserting into and deleting from have existed.

Traversing a Linked List

- Algorithms that traverse a list start at the first node and "visit" each node in succession until the last node has been "visited".
- What exactly constitutes a visit depends on the purpose of the algorithm. It may be that the algorithm is simply counting the number of nodes in the list, in which case the visit doesn't really do anything other than record the fact that the node exists. On the other hand, the algorithm may be printing the data values that appear in each of the nodes, in which case the visit will read a data value in each node and print that value.
- Let's develop an algorithm that will traverse a linked list.

Traversing a Linked List (cont.)



Traversing a Linked List (cont.)

- The basic logic to traverse a linked list is shown in the pseudocode below:

```
traverse (list)
    Set pointer to the first node in the list
    while (not end of the list)
            process (current node)
            set pointer to next node
end traverse
```

- Two basic concepts are incorporated in this algorithm.

1. An event loop is used to guard against overrunning the end of the list.
2. After processing the current node, the looping pointer is advanced to the next node.

Function: printList

```
// This function traverses a linked list and prints the values in each node
//  Since list data type is integer, function prints 10 values/line.
//  preconditions:  pList is a valid linked list.
// postconditions:  each node in the list has been printed.
void printList (NODE *pList)
{
   // local definitions
    NODE  *pWalker;
    int        lineCount = 0;

   // the code
   pWalker = pList;
   printf("\n List contains:\n");
   while (pWalker)
   {     if (++lineCount > 10)
         {          lineCount = 1;  //reset lineCount for next 10 values
                    printf("\n");
         } //end if
         printf("%3d ", pWalker->data.key);
         pWalker = pWalker->link;
   } //end while
   printf("\n");
   return;
} //end printList
```

ANSI/ISO C guarantees that the evaluation of a null pointer will be false. So this expression is equivalent to:

while (pWalker != NULL)

Practice Problem: Traversing a Linked List

- As a practice problem write a C function, similar to the one we just developed, that will return the average of all the values in a linked list.
- Write your function in the same style that we have developed. What type of value should your function return? What are the input parameters to your function?
- One possible solution to this problem appears on the next page. Don't look at it until you've written one of your own.

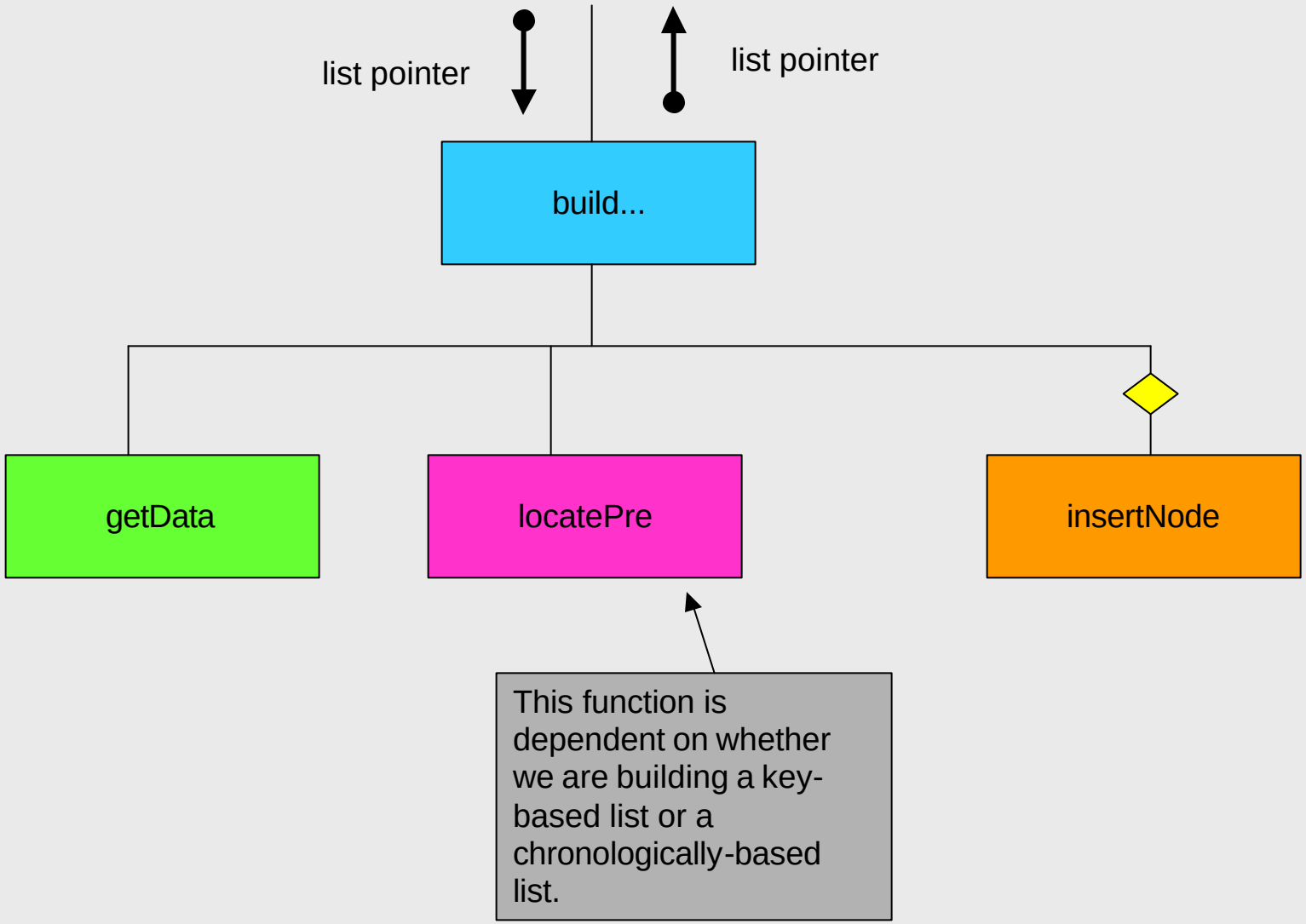
Function: averageList

```
// This function traverses a linked list and averages the values in the list
//  preconditions:  pList is a valid linked list of integers.
// postconditions:  the average value in the list is returned
void averageList (NODE *pList)
{
   // local definitions
    NODE  *pWalker;
    int         total = 0;
    int         count = 0;

   // the code
   pWalker = pList;
   while (pWalker)
   {     total += pWalker->data.key;
         count++;
         pWalker = pWalker->link;
   } //end while
   return (double) total/count;
} //end averageList
```


Building a Linked List

- We've developed several low-level functions for manipulating a linked list, insertion, deletion, and key-based searching as well as two applications, printing and determining the average.
- Now we need to consider how to construct a linked list from scratch.
- Again, we'll consider building a key-based linked list for the time being.
- Regardless of how the list nodes are ordered, the basic insertion logic remains the same. We need to get the data, create the node, determine the insertion point in the list, and then insert the new node. This process is illustrated on the next page.

Design for Building a Linked List



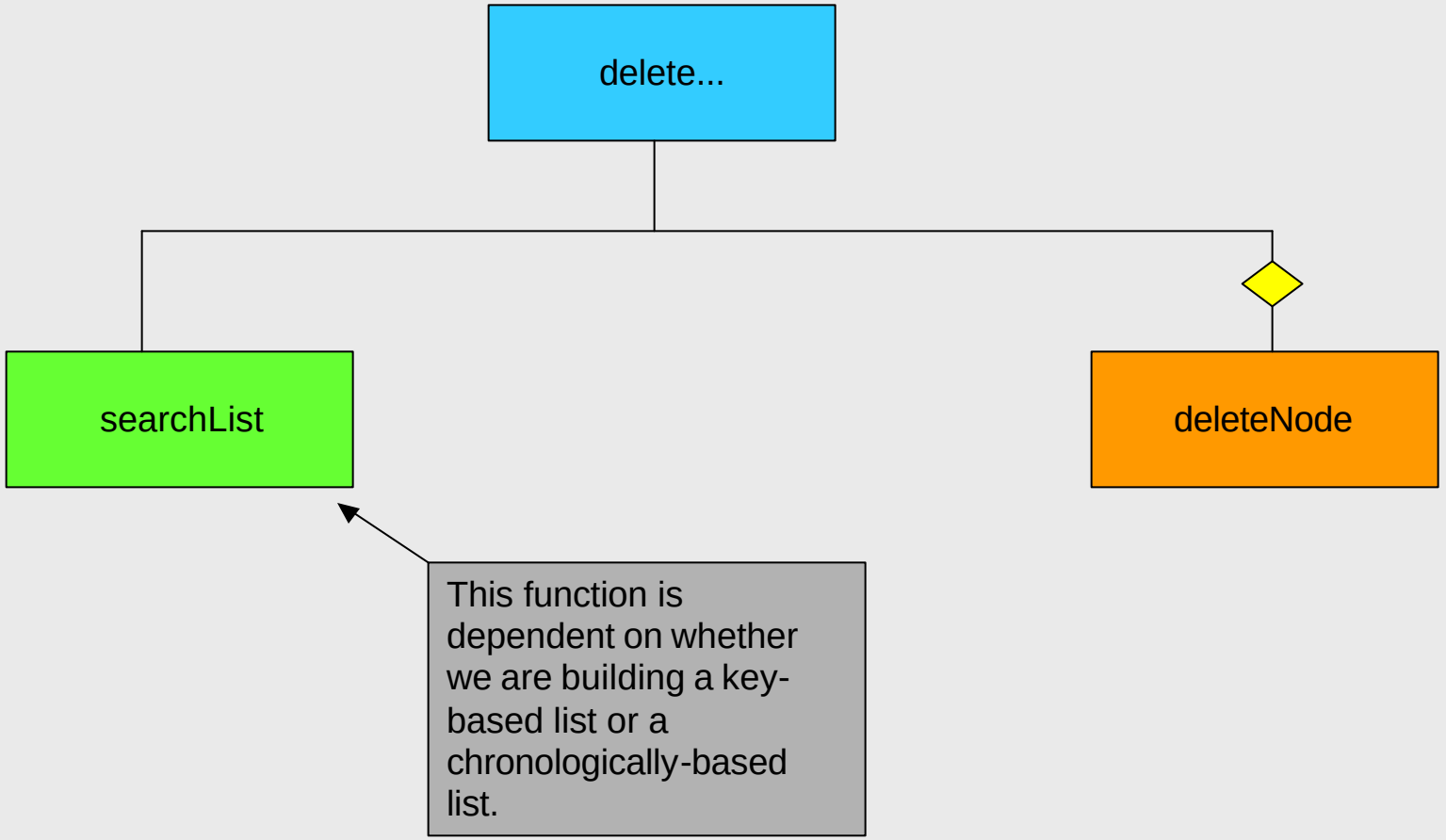
Function: buildKeyList

```
// This function constructs a key-based linked list from a file of data.
//  preconditions:  fileID is the file that contains the data.
// postconditions:  the list is built and a pointer to the head of the list is returned.
NODE  *buildKeyList (char  *fileID)
{
   // local definitions
    DATA    data;
    NODE   *pList;  // head pointer
    NODE   *pPre;  // logical predecessor pointer
    NODE   *pCur; // current pointer
    FILE     *fpData; // data file pointer

   // the code
   pList = NULL;
   fpData = fopen(fileID, "r");  //open file for reading
   if (!fpData)
   {   printf("Error opening file %s\a\n", fileID);
       exit(210);
   } // end if
   while (getData (fpData, &data))
   {     // find correct insert location
         searchList(pList, &pPre, &pCur, data.key);
         pList = insertNode(pList, pPre, data);
   } // end while
   return pList;
} //end buildKeyList
```


Removing Nodes From a Linked List

- Recall that our function to delete a node from a linked list was a low-level function. It assumed that some other function had set the values of the predecessor and current pointers, i.e., had found the node to be deleted.
- As with the high-level design for building (inserting into) a linked list, we need a similar high-level design for removing nodes from a linked list.
- This design is illustrated on the next page.

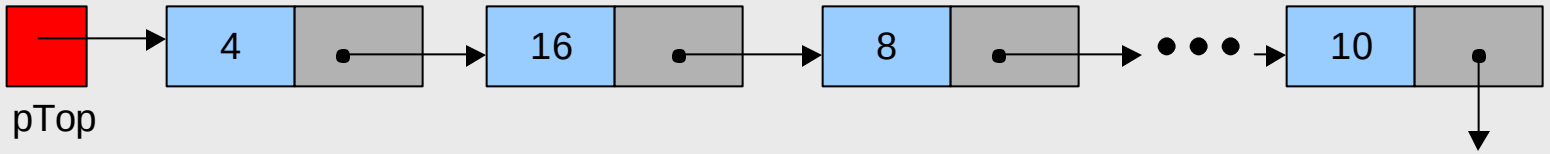
Design for Removing Nodes From a Linked List



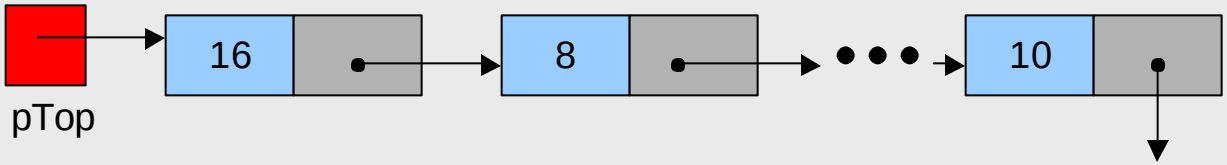
Function: deleteKeyNode

```
// This function deletes a specified target value from a key-based linked list.
//  preconditions:  pList is a valid list.
// postconditions:  the head pointer is returned and the list no longer includes the target node.
NODE  *deleteKeyNode (Node *pList)
{
   // local definitions
    int         target;  // value to be deleted
    int         found;
    NODE   *pPre;  // logical predecessor pointer
    NODE   *pCur; // current pointer

   // the code
   if (!pList)
   {     printf("List is empty...can't delete. \n");
         return pList;
   }
   printf("Enter value of node to be deleted: ");
   scanf("%d", target);
   found = searchList(pList, &pPre, &pCur, target);
   if (found)
   {    // target acquired – delete it
        pList = deleteNode(pList, pPre, pCur);
   }
   else  //target not found
   {    printf("Target value does not exist.\n");
   }
   return pList;
} //end deleteKeyNode
```

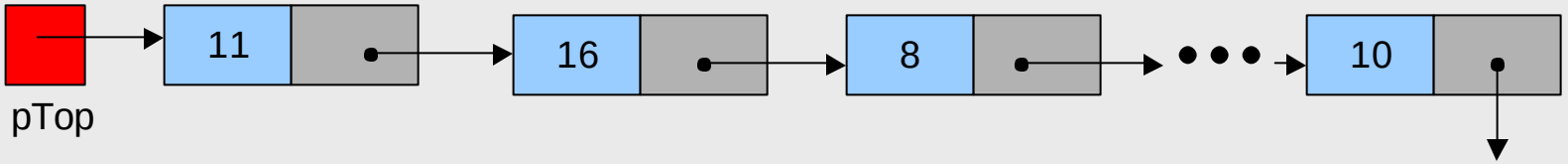

Visualizing a Stack Implemented with a Linked List [Structure Diagrams]



The stack at some point in time



The stack after a pop operation has occurred



The stack after a push(11) operation has occurred

Linked List Implementation of a Stack

```
// Arup Guha
// 11/7/01
// Small stack implementation
#include <stdio.h>

// Struct used to form a stack of integers.
struct stack {
  int data;
  struct stack *next;
};

// Prototypes
int push(struct stack **front, int num);
struct stack* pop(struct stack **front);
int empty(struct stack *front);
int top(struct stack *front);
void init(struct stack **front);
```


Linked List Implementation of a Stack (cont.)

```
int main() {

  struct stack *stack1, *temp;
  int tempval;

  init(&stack1);

  if (!push(&stack1, 3))
    printf("Push failed.\n");
  if (!push(&stack1, 5))
    printf("Push failed.\n");

  temp = pop(&stack1);
  if (temp !=NULL)
    printf("Pop stack = %d\n", temp->data);
```


Linked List Implementation of a Stack (cont.)

```
if (empty(stack1))
    printf("Empty stack\n");
  else
    printf("Contains elements.\n");

  tempval = top(stack1);
  if (tempval != -1)
    printf("Top of Stack = %d\n", tempval);

  temp = pop(&stack1);
  temp = pop(&stack1);
  if (temp != NULL)
    printf("Top of Stack = %d\n", temp->data);
  else
    printf("Tried to pop an empty stack.\n");

  return 0;
}
void init(struct stack **front) {
  *front = NULL;
}
```


Linked List Implementation of a Stack (cont.)

```
// Pre-condition: front points to the top of the stack.
// Post-condition: a new node storing num will be pushed on top of the
//        stack if memory is available. In this case a 1 is returned. If no memory is found,
//        no push is executed and a 0 is returned.

int push(struct stack **front, int num) {

  struct stack *temp;
  // Create temp node and link it to front.
  temp = (struct stack *)malloc(sizeof(struct stack));

  if (temp != NULL) {
    temp->data = num;
    temp->next = *front;
    *front = temp;
    return 1;
  }
  else
    return 0;
}
```

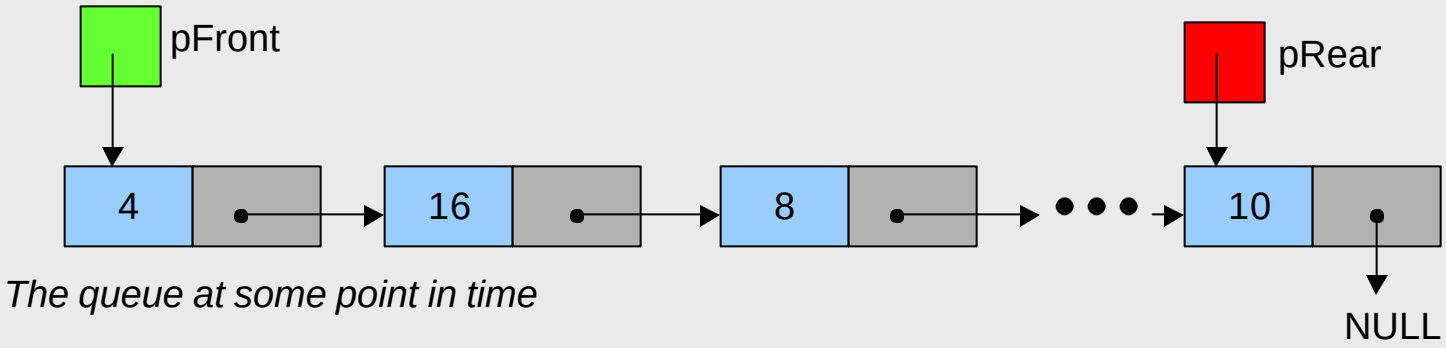

Linked List Implementation of a Stack (cont.)

```
// Pre-condition: front points to the top of a stack
// Post-condition: A pointer to a node storing the top value from the
//                 stack will be returned. If no value exists, the pointer
//                 returned will be pointing to null.

struct stack* pop(struct stack **front) {

  struct stack *temp;
  temp = NULL;

  if (*front != NULL) {
    temp = (*front);
    *front = (*front)->next;
    temp -> next = NULL;
  }
  return temp;
}
```

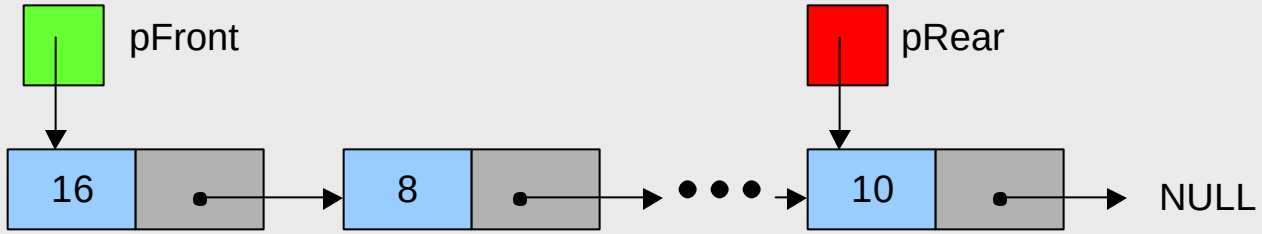

Linked List Implementation of a Stack (cont.)

```
// Pre-condition: front points to the top of a stack
// Post-condition: returns true if the stack is empty, false otherwise.
int empty(struct stack *front) {
  if (front == NULL)
    return 1;
  else
    return 0;
}

// Pre-condition: front points to the top of a stack
// Post-condition: returns the value stored at the top of the stack if the
//                 stack is non-empty, or -1 otherwise.
int top(struct stack *front) {

  if (front != NULL) {
    return front->data;
  }
  else
    return -1;
}
```


Visualizing a Queue Implemented with a Linked List [Structure Diagrams]

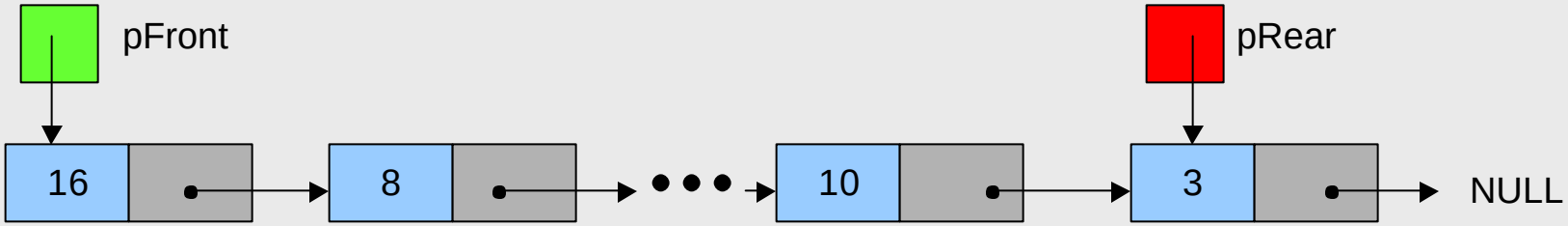


The queue at some point in time



The queue after a dequeue operation

Visualizing a Queue Implemented with a Linked List [Structure Diagrams] (cont.)



The queue after an enqueue(3) operation

Before the next class you write a linked list implementation for a queue. We'll look at the code for this next class.

Don't forget that Exam #2 is Thursday March 18th