COP 3502: Computer Science I
Spring 2004

– Note Set 15 –
Data Structures: Linked Lists

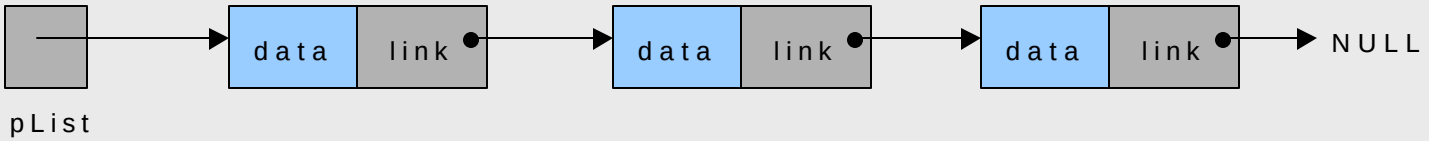
Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

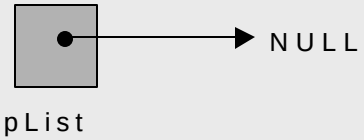
Linked Lists

- A linked list is an ordered collection of data in which each element (generally called nodes) contains the location of the next element; that is, each element contains two parts: a data part and a link part.
- The data part holds the useful information (as far as the user is concerned). The link is used to chain the data together. It contains a pointer that identifies the next node in the list.
- A pointer variable points to the first node in the list. The name of the list is the same as the name of this pointer variable.

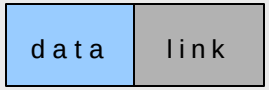
Example Linked Lists



A linked list containing three elements



An empty linked list



A list node

Linked List Nodes

- The nodes in a linked list are called self-referential structures.
- Each instance of the structure contains a pointer to another instance of the same type.
- The code on the next page represents the type definitions necessary to define a generic linked list. Some of the code will need to be filled in for a particular application.

A Generic Linked List Structure

```
// Global declarations
    typedef int KEY_TYPE;     //application dependent

    typedef struct
        {
            KEY_TYPE  key;
            ...  //other data fields as necessary
         } DATA;

    typedef struct node
        {
             DATA    data;
             struct node *link;
         } NODE;
```


Pointers to Linked Lists

- One of the attributes of a linked list is that its data are not stored with physical adjacency, i.e., next to each other as is the case in an array.
- Without a physical relationship between the nodes, we need some mechanism to distinguish the beginning of the list, i.e., a way to identify the first logical node in the list.
- This is typically done with a pointer referred to as a head pointer or list pointer.
- Although it is not a requirement to have a head pointer, it is very convenient and is commonly used. We'll adopt this mechanism for our linked lists.
- Typically, there will be additional pointers that reference nodes within a linked list. These are commonly used to assist in traversing (walking) the list or other application dependent uses.

Linked List Order

- Because a linked list is a linear structure, it always has an order. The order can be based on many things including chronological order (order of arrival or insertion into the list) and key-based order (lexical ordering based on the key value of the data items, such as alphabetic, numeric, etc.). Another common way to order the nodes in a linked list is based on the priority of the objects represented by the nodes of the list.
- Chronological linked lists are ordered by time. We've already seen two chronologically ordered lists in stacks and queues. In general there are FIFO lists and LIFO lists.
- Key-based linked lists are probably the most common type of linked list. New data is added at the correct point in the list based on the lexical ordering of the key values and the data which is already in the list at the time of the insertion.

Primitive Linked List Functions

- As with any data structure, to work with a linked list, we need some basic operations that manipulate the nodes.
- For example, we need to be able to insert nodes in the list, delete nodes from the list, search for the location of data (nodes) within the list, and so on. Given these primitive list functions, we can build functions that will process any linked list.

Linked List Functions: Design Approach

- Before going any further we need to discuss our design approach.
- Functions that change the contents of the list, such as insertion and deletion, will return the list head pointer. This design allows us to maintain the list easily by assigning the function's return value to the head.

  pList = insertNode (…);

  If the head of the list changes, it is automatically updated. If it doesn't change, then the list head is simply reset to its original address.

Linked List Functions: Design Approach (cont.)

- Functions that do not change the contents of the list return values that are consistent with their purpose.
  - For example, a function to locate a node will return an integer to indicate found or not found.
  - A function to determine the number of nodes in the list will return an integer count.
- Functions that process the entire list, such as printing the list, usually will return void.

Insertion Into A Linked List

- The following four steps are necessary to insert a node into a linked list.

  1. Allocate memory for the new node.
  2. Determine the insertion point. To identify this position we need only to know the new node's logical predecessor.
  3. Point the new node's link field to its logical successor.
  4. Point the predecessor to the new node.

- As indicated by step 2, we need to determine the location of the logical predecessor of the new node. There are four cases that arise as to the location of this logical predecessor. Can you identify all four cases?

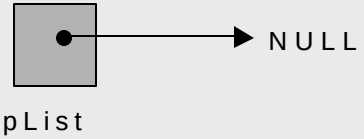
Location of the Logical Predecessor Node

- The fours cases for the location of the new node's logical predecessor are:

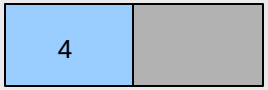
  1. It has none, the list is empty. The new node will become the first node in the list.
  2. It is in the first location of the list, meaning that the new node will become the first node in the list after the insertion occurs.
  3. It is in the last location of the list, meaning that the new node will become the last node in the list after the insertion occurs.
  4. It is at some arbitrary point which is neither the first node nor the last node in a list, meaning that the new node will be embedded in the middle of the list and will not be either the first nor last node in the list after the insertion occurs.

- We need to determine how these four cases are similar and how they are different, but we must be able to handle all of them.

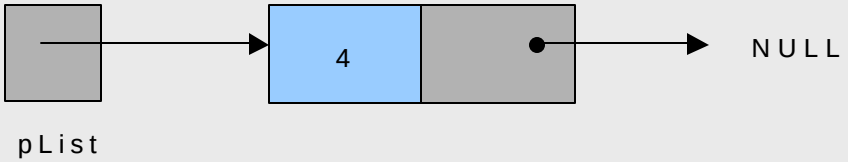
Case 1: Insertion Into An Empty List



An empty linked list

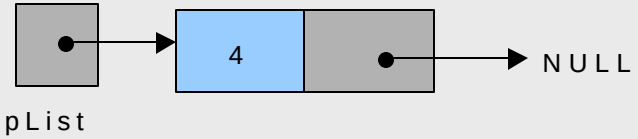


The new node to be inserted

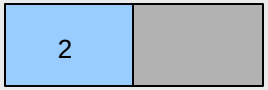


A list after the insertion of the new node

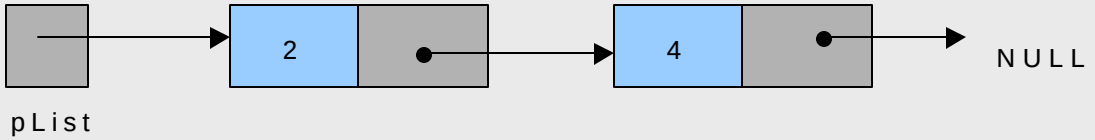
Case 2: Insertion At The Head Of A List



The initial linked list



The new node to be inserted
Assume a key-based list.

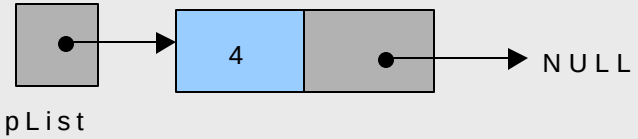


A list after the insertion of the new node

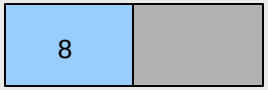
A Closer Look At Cases 1 and 2

- Notice how similar are cases 1 and 2.
- In both cases the head pointer pList winds up pointing to the newly inserted node.
- In both cases the new node winds up pointing to the same location that pList was pointing prior to the insert.

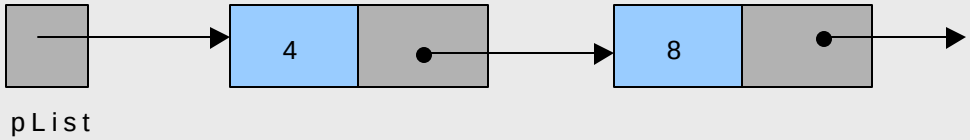
Case 3: Insertion At The End Of A List



The initial linked list

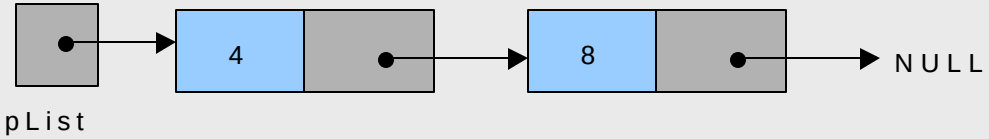


The new node to be inserted
Assume a key-based list.



A list after the insertion of the new node

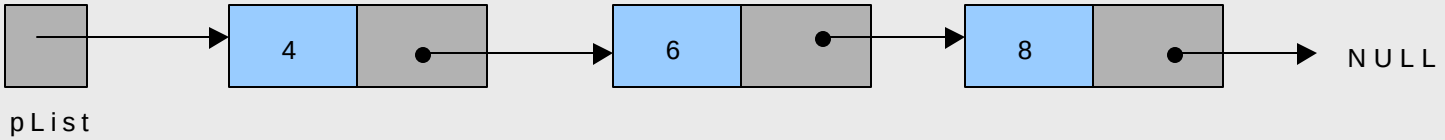
Case 4: Insertion In The Middle Of A List



The initial linked list

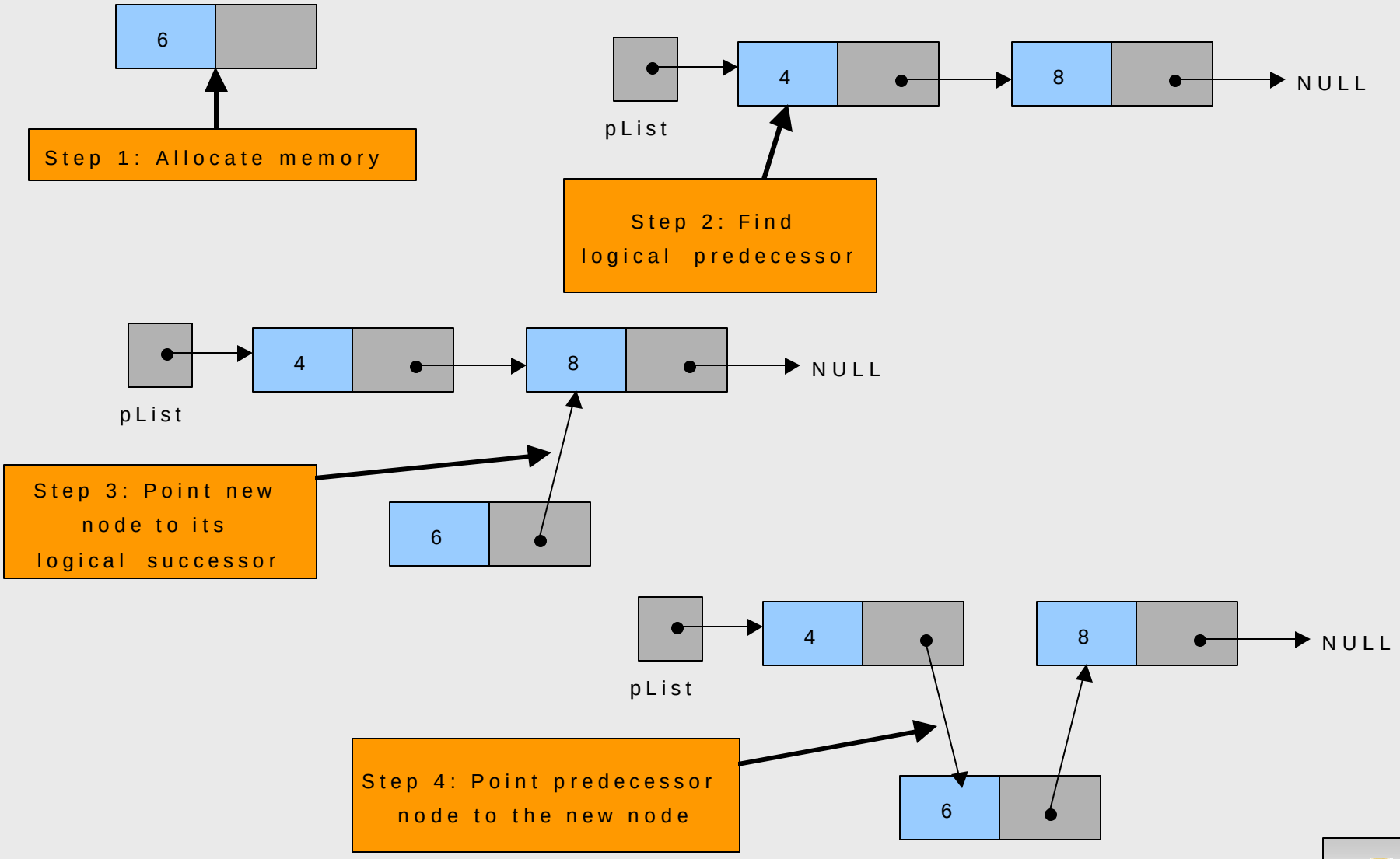
The new node to be inserted

Assume a key-based list.



A list after the insertion of the new node

Case 4: Details of the Insertion



A Closer Look At Cases 3 and 4

- Notice again, as with cases 1 and 2, how similar are cases 3 and 4.
- In both cases the new node winds up pointing to the same location that its logical predecessor pointed prior to the insertion.
- In both cases the logical predecessor of the new node winds up pointing to the new node.
- Since cases 1 and 2 are similar and as are cases 3 and 4, we'll be able to implement our insertion algorithm using only two cases representing the combination of these cases as we've outlined their similarities.

Function: insertNode

```
//insertNode
//  This function inserts a single node into a linked list.
//  preconditions:  pList is a pointer to the list, which might be null.  pPre points to
//                  the new node's logical predecessor.  Item contains the data.
//  postconditions: returns the head pointer.
NODE *insertNode (NODE *pList;   NODE *pPre;    DATA item)
{
    // local definitions
    NODE *pNew;   //node for the new data item to be inserted.

    if (! (pNew = NODE *) malloc(sizeof(NODE))))
        printf("\aMemory overflow in insert\n"), exit (100);

    pNew->data = item;   //set the data field in the new node.
    // Code for the various insertion cases.
```


Function: insertNode (cont.)

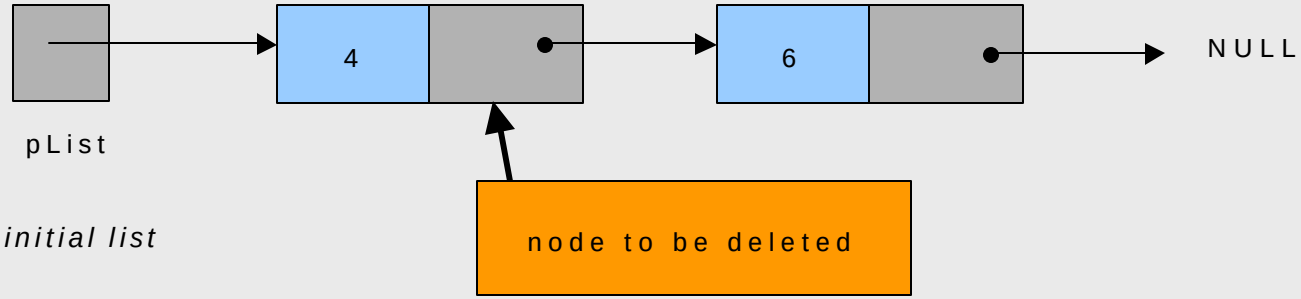
```
if (pPre == NULL)  //insertion into logical first location or list is empty.
      {
         pNew->link = pList;  //set new node to point to logical successor.
         pList = pNew;    //set logical predecessor to point to new node.
       }
else  // insertion in the middle of the list or at the logical end of the list.
      {
          pNew->link = pPre->link;
          pPre->link = pNew;
        }

return pList;
}  //end insertNode
```

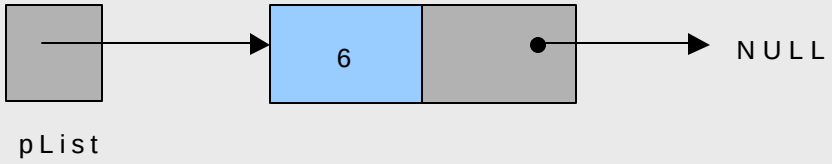
Deletion of Nodes in a Linked List

- Deleting a node from a linked list requires that we remove the node from the linked list by changing various link pointers and then physically deleting the node from the heap.
- The situations that can arise for deletion parallel those of insertion. Namely, we can:
  1. Delete the first node of a list.
  2. Delete any arbitrary node which is neither the first nor last node in the list.
  3. Delete the last node of a list.
  4. A special case occurs when we are deleting the only node in a list causing the resulting list to become empty.
- To logically delete a node, we must first find the node itself. We'll use a pointer defined as pCur for this purpose. We must also identify that node's logical predecessor.

Case 1: Deletion of the First Node in a List

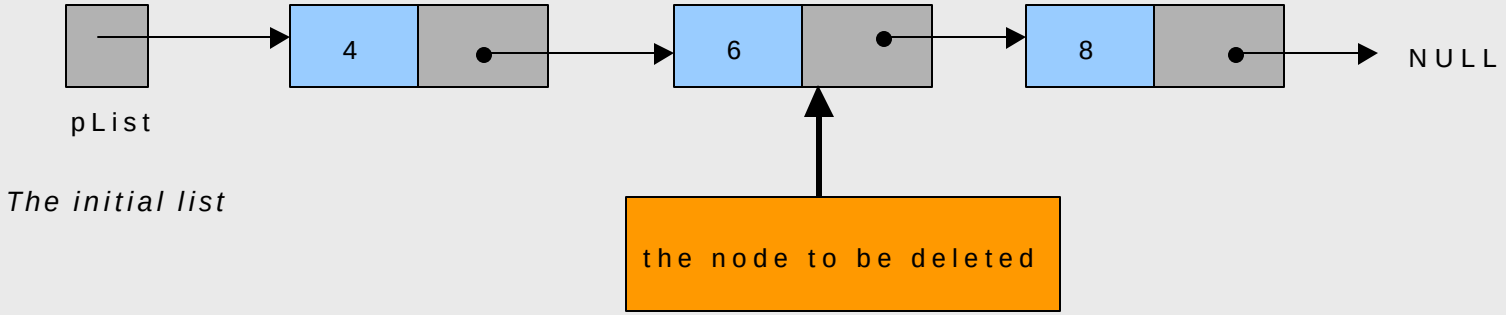


The initial list

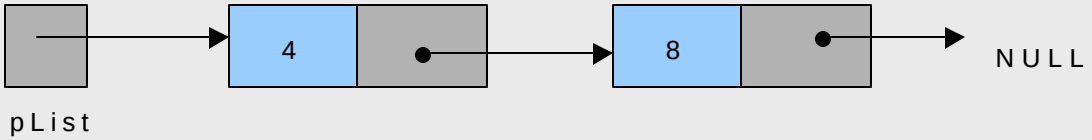


The list after deleting the first node

Case 2: Deletion From The Middle Of A List

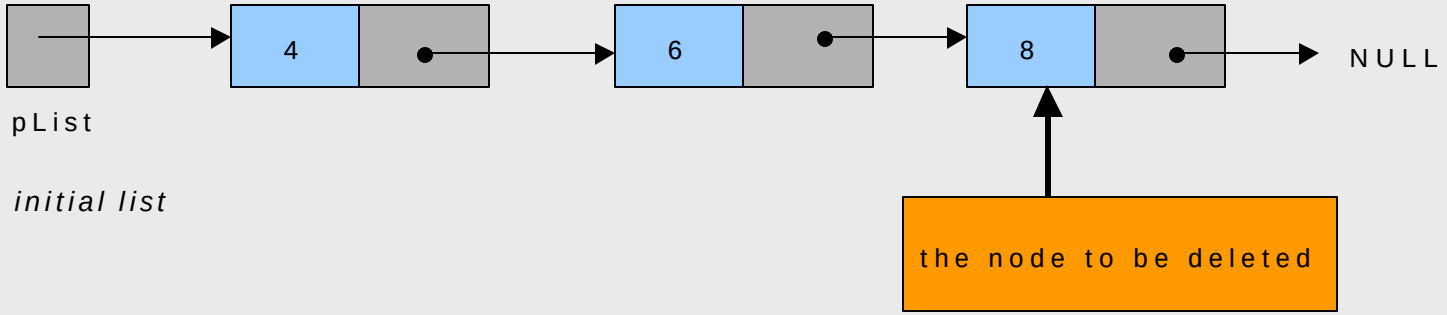


The initial list

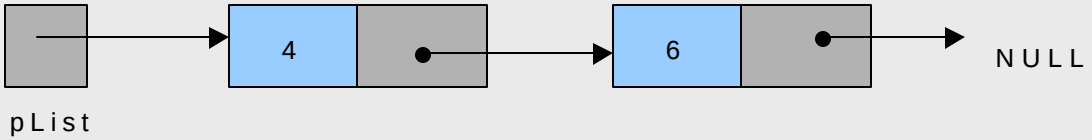


The list after the deletion has occurred

Case 3: Deletion Of The Last Node In A List

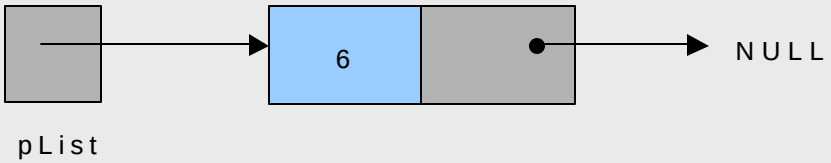


The initial list

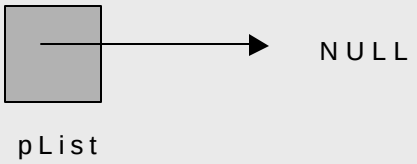


The list after the deletion has occurred

Case 4: Special Case Of Deleting The Only Node In A List



The initial list



The list after deleting the only node

Deleting the last node in a list is special only in the sense that the head pointer value which is returned by the function will be null instead of pointing to a valid node. However, under our design criteria this is what we would expect to have happened.

A Closer Look At The Deletion Cases

- As with insertion, the various cases of deletion have similarities that allow us to combine the cases in our implementation.
- Notice cases 1 and 4 are similar in that the head pointer winds up pointing to the logical successor of the deleted node.
- Similarly, cases 2 and 3 are similar in that the logical predecessor of the node which is deleted winds up pointing to the node that was the logical successor of the deleted node.
- As before, we'll combine these cases in the implementation.

Function: deleteNode

```
//deleteNode
//  This function deletes a single node from a linked list.
//  preconditions:  pList is a pointer to the list.  pPre points to the logical predecessor
//                  of the node to be deleted.  pCur points to the node to be deleted.
//  postconditions: deletes and recycles pCur. Returns the head pointer.
NODE *deleteNode (NODE *pList;   NODE *pPre;    NODE *pCur)
{
      if (pPre == NULL)    //deleting the first node in the list.
         pList = pCur->link;  //set the head pointer to the deleted node's logical successor.
      else  //deleting an arbitrary node in the list.
         pPre->link = pCur->link;

      free(pCur);
      return(pList);
} //end deleteNode
```


A Closer Look At The deleteNode Function

- There are three points in the deleteNode function worth mentioning.

1. The most important of these is that this function requires that the node to be deleted be identified before calling the function. It assumes that pCur and pPre are set at the time of the call.

2. Only one free() call is needed, there is no need to duplicate the free call in each block of the if statement.

3. As per our design criteria, the function returns the head pointer.

Searching A Key-Based Linked List

- Our design of the insertNode and deleteNode functions requires that we find the insertion point and the node to be deleted respectively.
- The search methods on linked lists vary depending on the logical ordering of the nodes in the list. For now, we'll restrict our considerations to key-based lists only.
- As we've seen, insertion requires us to identify the logical predecessor to the new node and for deletion we must identify both the node itself and its logical predecessor. If you think about other functions such as adding a node to a count of nodes or printing the contents of a node, these too will require knowing the location of the node. While we could write separate search functions for all three cases, it would be more efficient if we write one search function that will satisfy all of these requirements. Thus our search function will return both the predecessor and the current (found) locations.

Searching A Key-Based Linked List (cont.)

- Given a target key, the search function will attempt to locate the requested node in the linked list.
- If a node in the list matches the target value, the search function should return true (1); if no key matches, it will return false (0). The predecessor and current pointers are set according to the rules shown on the next page.
- Since the list is maintained in key sequence order, we can use a modified version of a sequential search which is commonly referred to as an "ordered sequential search".
  - Start at the beginning of the list and search sequentially until the target is no longer greater than the current node's key. At this point the target value is either less than or equal to the current node's key value. The predecessor at this point is pointing to the node which logically precedes the current node. A test if pCur->data matches the target will determine the success of our search.

Searching A Key-Based Linked List (cont.)

Condition	pPre	pCur	return value
target < first node	NULL	first node	0
target == first node	NULL	first node	1
first < target < last	largest node < target	first node > target	0
target == middle node	node's predecessor	equal node	1
target == last node	last's predecessor	last node	1
target > last node	last node	NULL	0

Function: searchList

```
//searchList
//  Given a key value, this function find the location of a node matching this key value.
//  preconditions:  pList is a pointer to the list.  pPre points to a variable to receive
//                  the logical predecessor.  pCur points to a variable to receive the
//                  current node.
//  postconditions: pCur points to the first node with equal/greater key value, or null if
//                   target > key of the last node in the list.  pPre points to the largest node
//                   smaller than the key value, or null if the target < key of first node in
//                   the list.  Function returns 1 if found and 0 otherwise.
int searchList (NODE *pList,   NODE **pPre,    NODE **pCur;  KEY_TYPE target)
{
     //local definitions
     int found = 0;

     pPre = NULL;  //no initial predecessor.
     pCur = pList;   //set current pointer to head of the list.
```


Function: searchList (cont.)

```
        //start the search from the beginning of the list.
        while (*pCur != NULL && target > (*pCur)->data.key)
             {
                  *pPre = *pCur;  // advance pointer.
                   *pCur = (*pCur)->link;   //advance pointer.
              }
         if (*pCur && target == (*pCur)->data.key)
               found = 1;  //successful search – target acquired

         return found;
} //end searchList
```