**Lecture 28 - Chapter 12: C Data Structures – Wed Nov 29 or Thurs Nov 30**

**Announcements**

Reading:

* Chapter 12

Assignments:

* Assignment #11 - due on **Dec 4** (MW class) or **Dec 5** (TR class) **(no late assignments accepted)**

**Today’s Goals**

1. Linked Lists

**Today’s Terminology**

**Terminology**

* Self-Referential Structure
  + When a structure contains as a member that is a pointer to its same structure type
* Dynamic Data Structures
  + A data structure that can grow and shrink during execution time
* Dynamic Memory Allocation
  + Obtaining and releasing memory during execution time
* Linked Lists
  + Linear collection of self-referential structures!
  + Insertions and deletions are made anywhere in the list

**Linked Lists**

**Singly Linked List Examples**

* Need a pointer to first node – called the **head** pointer
* Each node is created as needed
* Will do the following **Going step by step**
  + Insert **Tie together at end**
  + Delete **Cover steps book doesn’t**

**Insert Example #1: (create simple linked list)**

* Last lecture we inserted nodes and created this linked list

headPtr

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

**Insert Example #2 (inserting in middle)**

* Insert node with data = 5 in proper place in above linked list

**struct** node {

**int** data;

**struct** node \*nextPtr;

};

**int** **main**(**void**) {

**int** number = 5;

// Allocate memory for node on the heap and setup the node

nodePtr = **malloc**(**sizeof**(**struct** node));

**if** (nodePtr != NULL) {

nodePtr->data = number;

nodePtr->nextPtr = NULL;

// Figure out where this node belongs in linked list

// Need some pointers to help us keep track

**struct** node \*previousPtr = NULL;

**struct** node \*currentPtr = headPtr;

// While haven't reached end of list and current node's number <= new node’s number

**while** (currentPtr != NULL && currentPtr->data <= number ) {

previousPtr = currentPtr;

currentPtr = currentPtr->nextPtr;

}

// Previous is pointing to node **BEFORE** insertion point

// Current is pointing to node **AFTER** insertion point

// Connect new node into list without losing any nodes in the process!

previousPtr->nextPtr = nodePtr;

nodePtr->nextPtr = currentPtr;

// Walk list and print values

**printf** ("The list is: ");

currentPtr = headPtr;

**while** (currentPtr != NULL) {

**printf** ("%d --> ", currentPtr->data);

currentPtr = currentPtr->nextPtr;

}

**puts** ("NULL");

} // if nodePtr != NULL

**else** {

**puts** ("No memory for node");

}

} // main

**Displays**

The list is: 2 --> 4 --> 5 --> 6 --> NULL

**Visually** (inserting node with data = 5)

**Step1:** **headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr = NULL currentPtr = head**Ptr**

**Step 2: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr currentPtr

**Step 3: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr currentPtr

**Step 4: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr->nextPtr = nodePtr;

nodePtr->next = current;

|  |  |  |  |
| --- | --- | --- | --- |
| 5 |  |  | nodePtr |

**Step5: Complete!**

headPtr

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 5 |  |  | 6 |  |  | Null |

**Insert Example #3 (inserting at end)**

* Handle insertion at **end of list**
  + Already covered with current code!

**Step: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr currentPtr

**Step: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr->nextPtr = nodePtr;

nodePtr->next = current;

|  |  |  |  |
| --- | --- | --- | --- |
| 7 |  |  | nodePtr |

**Insert Example #4 (inserting at beginning)**

* Handle insertion at **front of list**
  + Say node data = 0
  + Current code we are building case by case crashes!
  + Here is the code from above:

// Figure out where this node belongs in linked list

// Need some pointers to help us keep track

**struct** node \*previousPtr = NULL;

**struct** node \*currentPtr = headPtr; number = 0 so we skip while

**while** (currentPtr != NULL && currentPtr->data <= number ) {

previousPtr = currentPtr;

currentPtr = currentPtr->nextPtr;

}

// Previous is pointing to node BEFORE insertion point

// Current is pointing to node AFTER insertion point

// Connect new node into list without losing any nodes in the process!

previousPtr->nextPtr = nodePtr; **CODE CRASHES HERE!**

nodePtr->nextPtr = currentPtr; Attempting to access NULL

pointer (previousPtr)

* + **To fix, replace above access to previousPtr with the following:**

headPtr = nodePtr; Reassign head pointer to

nodePtr->nextPtr = currentPtr; new node to correctly insert

**Step 1: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr = NULL currentPtr = head**Ptr**

**Step 2: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

headPtr = nodePtr

nodePtr->nextPtr = currentPtr;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 |  |  | nodePtr |  |

**Insert Example #5: Tying it all together -- insert anywhere in the list**

* Note that the insert method takes ***address of headPtr***!
* Need address so can modify headPtr’s address when adding new node to front!

**struct** node {

**int** data;

**struct** node \*nextPtr;

};

**int** **main**(**void**) {

**int** number;

**printf** ("Enter a value for node to add to list ");

**scanf** ("%d", &number);

List's address is necessary **ONLY** when node is added as 1st node. Why? Because providing address enables value in headPtr to be modified (call by reference)

insertNode(&headPtr, number);

printList (headPtr);

}

// Insert a node into the linked list

**void** **insertNode** (**struct** node \*\*headPtr, **int** number) {

// Allocate memory for node in the heap and setup node

**struct** node \*nodePtr = **malloc**(**sizeof**(**struct** node));

**if** (nodePtr != NULL) {

nodePtr->data = number;

nodePtr->nextPtr = NULL;

// Figure out where new node belongs in linked list, create 2 pointers to help

**struct** node \*previousPtr = NULL;

**struct** node \*currentPtr = \*headPtr;

// While haven't reached end of list and current node's number <= number

**while** (currentPtr != NULL && currentPtr->data <= number ) {

previousPtr = currentPtr;

currentPtr = currentPtr->nextPtr;

}

// previousPtr pointing to node **BEFORE** insertion point - if NULL, inserting as 1st node!

// currentPtr pointing to node **AFTER** insertion point

// Connect new node into list

**if** (previousPtr == NULL) {

\*headPtr = nodePtr;

}

**else** {

previousPtr->nextPtr = nodePtr;

}

nodePtr->nextPtr = currentPtr;

} // nodePtr != NULL

**else** {

**printf** ("No memory to create node for %d\n", number);

}

} // insertNode

// Walk down list printing the value in each node

**void** **printList** (**struct** node \*listPtr) {

// Walk list and print values

**if** (listPtr != NULL) {

**printf** ("The list is: ");

**struct** node \*currentPtr = listPtr;

**while** (currentPtr != NULL) {

**printf** ("%d --> ", currentPtr->data);

currentPtr = currentPtr->nextPtr;

}

**puts** ("NULL");

}

**else** {

**puts** ("List is empty");

} // linked list is empty

} // printList

**Delete Example #1 (deleting last node)**

* Let’s look at the case of deleting the last node in the list (easiest case)
* Assume we have code from insertion examples above

**struct** node \*previousPtr = NULL;

**struct** node \*currentPtr = headPtr;

While haven't reached the **LAST NODE** in the list

**while** (currentPtr->nextPtr != NULL) {

previousPtr = currentPtr;

currentPtr = currentPtr->nextPtr

}

currentPtr is pointing to LAST node in list

previousPtr is pointing to node BEFORE LASTnode

Now disconnect the last node and free memory

previousPtr->nextPtr = currentPtr->nextPtr;

**free**(currentPtr);

currentPtr = NULL;

// Display the new list with node deleted

printList(headPtr);

**Displays**

The list is: 2 --> 4 --> NULL

**Visually** (deleting node with data = 6)

**Step1: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr = NULL currentPtr = head**Ptr**

**Step 2: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr currentPtr

**Step 3: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr currentPtr

**Step 4: headPtr**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | Null |

previousPtr currentPtr

**Delete Example #2 (deleting node in middle)**

* Changes need highlighted in code **1) Need to add in comparison!**

While haven’t reached end of list and

**struct** node \*previousPtr = NULL; current node's number != numberToDelete

**struct** node \*currentPtr = headPtr;

**int** numberToDelete = 4;

**while** (currentPtr != NULL && currentPtr->data != numberToDelete) {

previousPtr = currentPtr;

currentPtr = currentPtr->nextPtr; **2) Need to check if found node!**

} **currentPtr** is pointing to node to delete

but if NULL then node is not in list.

**previousPtr** is pointing to node BEFORE

**if** (currentPtr != NULL) { node to delete

previousPtr->nextPtr = currentPtr->nextPtr;

**free**(currentPtr);

currentPtr = NULL; Code in if-stmt is the same, just added

} test to see if found node

**else** {

**puts** ("Node to delete not found!");

}

printList(headPtr);

* Delete node with data = 4

**Step 1: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr = NULL currentPtr = head**Ptr**

**Step 2: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr currentPtr

**Step 3: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr currentPtr

**Step 4: headPtr**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 6 |  |  | Null |

previousPtr currentPtr = NULL

**Delete Example #3 (deleting 1st node)**

* Handle deletion at **front of list**
  + Say numberToDelete = 2
  + Current code we are building case by case crashes!
  + Here is the code from above:

// Prompt user for number to be deleted from the linked list

**int** numberToDelete;

**printf** ("Enter number of last node in list ");

**scanf** ("%d", &numberToDelete);

**struct** node \*previousPtr = NULL;

**struct** node \*currentPtr = headPtr;

numberToDelete = 2 so skip while

**while** (currentPtr != NULL && currentPtr->data != numberToDelete ) {

previousPtr = currentPtr; // walk to...

currentPtr = currentPtr->nextPtr; // ... next node

}

// currentPtr is pointing to node to delete - if NULL, node is not in list!

// previousPtr is pointing to node BEFORE node to delete

**if** (currentPtr != NULL) { previousPtr is **NULL** so this

previousPtr->nextPtr = currentPtr->nextPtr; code crashes! Attempting

**free**(currentPtr); to access NULL pointer!

currentPtr = NULL;

}

**else** {

**puts** ("Node to delete not found!");

}

// Display the new list with node deleted

printList(headPtr);

* + **To fix, add to above code the following:**

Check if node to delete is 1st node!

**if** (headPtr->data == numberToDelete) {

headPtr = headPtr->nextPtr;

**free** (currentPtr);

currentPtr = NULL;

}

**else** {

Code from while loop on goes here

} // not deleting 1st node

**Visually** (deleting node with data = 2)

**Step 1: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr = NULL currentPtr = head**Ptr**

**Step 2: headPtr**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 |  |  | 4 |  |  | 6 |  |  | Null |

previousPtr = NULL currentPtr = head**Ptr**

**Step 3: headPtr**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 |  |  | 6 |  |  | Null |

previousPtr = NULL currentPtr = NULL

**Delete Example #4: Code to delete any node in the list**

**struct** node {

**int** data;

**struct** node \*nextPtr;

};

**int** **main**(**void**) {

// Prompt user for number to be deleted from the linked list

**printf** ("Enter number of ANY node in list ");

**scanf** ("%d", &numberToDelete);

// The list's address is necessary only in the case when a node is added as the 1st node

// Why? Because providing the address enables the headPtr to be modified (call by reference)

deleteNode(&headPtr, numberToDelete);

printList (headPtr);

} // main

// Delete a node in the linked list

**void** **deleteNode** (**struct** node \*\*headPtr, **int** numberToDelete) {

// Figure out where node to delete is in linked list, create two pointers to help

**struct** node \*previousPtr = NULL;

**struct** node \*currentPtr = \*headPtr;

// Check if node to delete is 1st node!

**if** ((\*headPtr)->data == numberToDelete) {

\*headPtr = (\*headPtr)->nextPtr;

**free** (currentPtr);

currentPtr = NULL;

}

**else** {

// While haven't reached end of list and current node's number != numberToDelete

**while** (currentPtr != NULL && currentPtr->data != numberToDelete ) {

previousPtr = currentPtr; // walk to...

currentPtr = currentPtr->nextPtr; // ... next node

}

// currentPtr is pointing to node to delete - if NULL, node is not in list!

// previousPtr is pointing to node BEFORE node to delete - if NULL, node is 1st node

**if** (currentPtr != NULL) {

previousPtr->nextPtr = currentPtr->nextPtr;

**free**(currentPtr);

currentPtr = NULL;

}

**else** {

**puts** ("Node to delete not found!");

}

} // not 1st node

} //deleteNode

// Walk down list printing the value in each node

**void** **printList** (**struct** node \*listPtr) {

// Walk list and print values

**if** (listPtr != NULL) {

**printf** ("The list is: ");

**struct** node \*currentPtr = listPtr;

**while** (currentPtr != NULL) {

**printf** ("%d --> ", currentPtr->data);

currentPtr = currentPtr->nextPtr;

}

**puts** ("NULL");

}

**else** {

**puts** ("List is empty");

} // linked list is empty

} // printList