**Lecture 29 - Chapter 12: C Data Structures – Mon Dec 4 or Tues Dec 5**

**Announcements**

Reading:

* Chapter 12

Assignments:

* Due: Assignment #11

**Today’s Goals**

1. FCQ
2. Stacks
3. Queues

**FCQ**

Visit this link [colorado.campuslabs.com/courseeval](http://colorado.campuslabs.com/courseeval)

Authenticate using your campus user ID

Your input is anonymous

Please include anything that will help me improve my teaching as well as anything you felt I did well! Thank you!

**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
* Stacks
  + Linear data structure - Last in First Out (LIFO)
  + Insertions and deletions are made only at the top of the stack
* Queues
  + Linear data structure - First in First Out (FIFO)
  + Insertions are made at the back (tail)
  + Deletions are made at the front (head)
  + Represents a waiting line
* Binary Trees
  + Nonlinear, two-dimensional data structure

**Stacks**

**Linked Lists**

* Collection of linked structures
* Can insert values anywhere into the list easily
* Can delete items from anywhere in the list easily

**Stack**

* Constrained version of linked list
* Insertions and deletions are made **only at the top** of the stack
* Linear data structure - Last in First Out (**LIFO**)

**Stack Operations**

* Push – inserting new node on top of stack
* Pop – deleting node on top of stack
* Peek – examining the node on the top of the stack

**Stack Example #1**

* Create a ***simple stack*** with numbers => {2, 4, 6}
* Result:

topPtr

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

**struct** stackNode {

**int** data; Same as a linked list node

**struct** stackNode \*nextPtr; Just named it differently

};

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

// Build a simple stack

// Create two node pointer variables

**struct** stackNode \*topPtr = NULL;

**struct** stackNode \*stackNodePtr = NULL;

// Allocate memory in the heap for 1st stack node

stackNodePtr = **malloc**(**sizeof**(**struct** stackNode));

// Setup the 1st stack node

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

stackNodePtr->data = 2;

stackNodePtr->nextPtr = NULL;

}

// Set top pointer to this 1st stack node

topPtr = stackNodePtr;

// Allocate memory in the heap for 2nd stack node

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

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

stackNodePtr->data = 4;

stackNodePtr->nextPtr = NULL;

}

// Insert 2nd node at top of stack!

stackNodePtr->nextPtr = topPtr;

topPtr = stackNodePtr;

// Allocate memory in the heap for 3rd stack node

stackNodePtr = **malloc**(**sizeof**(**struct** stackNode));

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

stackNodePtr->data = 6;

stackNodePtr->nextPtr = NULL;

}

// Insert at top of stack! This is different from linked list!

stackNodePtr->nextPtr = topPtr;

topPtr = stackNodePtr;

// Walk list and print values – SAME AS LINKED LISTS JUST RENAMED!

printStack (topPtr);

} // main

// Walk stack printing the value in each node

**void** **printStack** (**struct** stackNode \*stackPtr) {

// Walk stack and print values

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

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

**struct** stackNode \*currentPtr = stackPtr;

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

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

currentPtr = currentPtr->nextPtr;

}

**puts** ("NULL");

}

**else** {

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

} // stack is empty

} // printStack

**Displays**

The stack is: 6 --> 4 --> 2 --> NULL

**Stack Push Example:**

* Move above code into a function called “push”

**struct** stackNode {

**int** data;

**struct** stackNode \*nextPtr;

};

**void** **push** (**struct** stackNode \*\*topPtr, **int** number);

**void** **printStack** (**struct** stackNode \*stackPtr);

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

//Assume we have the stack we just created!

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

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

// Stack's address is necessary since node is always added as 1st node.

// Providing address enables value stored in topPtr (address of 1st node) to be

// modified.

push(&topPtr, number);

printStack (topPtr);

} // main

// Insert a new node on the top of the stack

**void** **push** (**struct** stackNode \*\*topPtr, **int** number) {

// Allocate memory for stack node in the heap

**struct** stackNode \*stackNodePtr = **malloc**(**sizeof**(**struct** stackNode));

// Setup the stack node and insert as 1st node

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

stackNodePtr->data = number;

stackNodePtr->nextPtr = \*topPtr;

\*topPtr = stackNodePtr;

}

**else** {

**puts** ("Could not allocate memory for node");

}

} // push

**Displays**

Enter a value for node to add to the stack **1**

The stack is: 1 --> 6 --> 4 --> 2 --> NULL

**Stack Pop Example:**

* Before we attempt to remove an item from the stack we should make sure stack is not empty!
* Write a method called isEmpty

Returns true or false based on if stack is empty or not

bool **isEmpty** (**struct** stackNode \*topPtr) {

**return** topPtr == NULL;

}

* Code to remove top node

**struct** stackNode {

**int** data;

**struct** stackNode \*nextPtr;

};

**void** **pop** (**struct** stackNode \*\*topPtr);

**void** **printStack** (**struct** stackNode \*stackPtr);

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

// Stack's address is necessary since node is always added as 1st node.

// Providing address enables value stored in topPtr (address of 1st node) to be

// modified.

**if** (!isEmpty(topPtr)) {

pop(&topPtr);

}

printStack (topPtr);

} // main

// Delete node from the top of the stack

**void** **pop** (**struct** stackNode \*\*topPtr) {

**struct** stackNode \*tempPtr = \*topPtr;

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

**free** (tempPtr);

tempPtr = NULL;

} // pop

**Displays**

The stack is: 6 --> 4 --> 2 --> NULL

**Visually**

**Step 1: topPtr**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| tempPtr |  | 1 |  |  | 6 |  |  | 4 |  |  | 2 |  |  | Null |

**Step 2: topPtr**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| tempPtr |  | 1 |  |  | 6 |  |  | 4 |  |  | 2 |  |  | Null |

**Step 3: topPtr**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| tempPtr |  | 1 |  |  | 6 |  |  | 4 |  |  | 2 |  |  | Null |

**Queues**

**Queue**

* Constrained version of linked list
* Insertions are made at the back (tail)
* Deletions are made at the front (head)
* Represents a waiting line
* Linear data structure - First in, First Out (**FIFO**)

**Queue Operations**

* enqueue – inserting new node into back of queue
* dequeue – deleting node from front of queue

**Queue Example #1**

* Create a ***simple queue*** with numbers => {2, 4, 6}
* Result:

headPtr tailPtr

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

**struct** queueNode {

**int** data;

**struct** queueNode \*nextPtr;

};

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

// Build a simple queue -> {2, 4, 6}

// Create two node pointer variables

**struct** queueNode \*headPtr = NULL;

**struct** queueNode \*tailPtr = NULL;

**struct** queueNode \*queueNodePtr = NULL;

// Allocate memory in heap for a queue node

queueNodePtr = **malloc**(**sizeof**(**struct** queueNode));

// Setup the 1st queue node

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

queueNodePtr->data = 2;

queueNodePtr->nextPtr = NULL;

}

// Set the head and tail to point to this first queue node

headPtr = queueNodePtr;

tailPtr = queueNodePtr;

// Allocate memory in heap for 2nd queue node

queueNodePtr = **malloc**(**sizeof**(**struct** queueNode));

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

queueNodePtr->data = 4;

queueNodePtr->nextPtr = NULL;

}

// Insert at end of queue! THIS IS DIFFERENT FROM LINKED LIST

tailPtr->nextPtr = queueNodePtr;

tailPtr = queueNodePtr;

// Allocate memory in heap for 3rd queue node

queueNodePtr = **malloc**(**sizeof**(**struct** queueNode));

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

queueNodePtr->data = 6;

queueNodePtr->nextPtr = NULL;

}

// Insert at end of queue! THIS IS WHERE THINGS WILL CHANGE

tailPtr->nextPtr = queueNodePtr;

tailPtr = queueNodePtr;

// Walk list and print values – SAME AS LINKED LIST AND STACK

printQueue (headPtr);

} // main

// Walk queue printing the value in each node

**void** **printQueue** (**struct** queueNode \*queuePtr) {

// Walk queue and print values

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

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

**struct** queueNode \*currentPtr = queuePtr;

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

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

currentPtr = currentPtr->nextPtr;

}

**puts** ("NULL");

}

**else** {

**puts** ("Queue is empty - nothing to print");

} // queue is empty

} // printQueue

**Displays**

The queue is: 2 --> 4 --> 6 --> NULL

**Enqueue Example:**

* Assume queue from above
* Note enqueue method takes ***address of headPtr and tailPtr***!
* Need address to modify value stored in ***headPtr – address of 1st node*** and ***tailPtr – address of last node***

**struct** queueNode {

**int** data;

**struct** queueNode \*nextPtr;

};

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

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

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

headPtr's address is necessary **ONLY** when node is

1st node. tailPtr is always need since always

Changing value stored in tailPtr. Providing address

Enables value in headPtr and tailPtr to be modified

enqueue(&headPtr, &tailPtr, number);

**printf** ("tailPtr->data should be %d AND it is %d\n", number, tailPtr->data);

printQueue (queueHeadPtr);

} // main

// Insert node at end of queue

**void** **enqueue** (**struct** queueNode \*\*headPtr, **struct** queueNode \*\*tailPtr, **int** number) {

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

**struct** queueNode \*queueNodePtr = **malloc**(**sizeof**(**struct** queueNode));

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

queueNodePtr->data = number;

queueNodePtr->nextPtr = NULL;

// Is this the 1st node in the queue?

**if** (\*headPtr == NULL) {

\*headPtr = queueNodePtr;

}

**else** {

// Connect node into queue - set last node to point to new node

(\*tailPtr)->nextPtr = queueNodePtr;

}

// Move tailPtr over to new node

\*tailPtr = queueNodePtr;

}

**else** {

**printf** ("No memory available to insert %d\n", number);

}

} //enqueue

**Displays -- Assume queue 2 --> 4 --> 6 --> NULL**

Enter a value for node to add to the queue **3**

Value pointed at by queueTailPtr should be 3 AND it is 3

The queue is: 2 --> 4 --> 6 --> 3 --> NULL

**Dequeue Example:**

* Before attempt to remove an item make sure queue it is not empty!
* Write a method called isEmpty

Returns true or false based on if queue is empty or not. Same as stack function!

bool **isEmpty** (**struct** queueNode \*headPtr) {

**return** headPtr == NULL;

}

* Assume queue from above
* Code to remove 1st node in queue

**struct** queueNode {

**int** data;

**struct** queueNode \*nextPtr;

};

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

// The queue's headPtr address is necessary if node deleting is ONLY ONE in queue

// Providing address enables value stored in headPtr to be modified

// Same situation if queue contains nodes and deleting last node. Need to change

// value stored in tailPtr

dequeue(&queueHeadPtr, &queueTailPtr);

printQueue (queueHeadPtr);

}

// Remove a node from front of queue

**void** **dequeue** (**struct** queueNode \*\*headPtr, **struct** queueNode \*\*tailPtr) {

// Set tempPtr to point to 1st node in queue

// Move head to 2nd node in queue

**struct** queueNode \*tempPtr = \*headPtr;

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

// Is there only one node in the list

**if** (\*headPtr == NULL) {

\*tailPtr = NULL;

}

**free** (tempPtr);

tempPtr = NULL;

} //dequeue

**Displays -- Assume queue 2 --> 4 --> 6** --> 3 **--> NULL**

The queue is: 4 --> 6 --> 3 --> NULL