Data Structures in C

Prof. Georg Feil

Linked Lists

Winter 2018

Acknowledgement

- These lecture slides are partly based on slides by Rachel Jiang and Simon Hood
- Additional sources are cited separately

Reading Assignment (required)

- Data Structures (recommended textbook)
  - Chapter 4

(Note the textbook does a few things we might consider poor style, for example one-letter variable names and using int for Boolean values.)

So far…

- We've seen two important data structures, stacks and queues, implemented using arrays
  - Data items in an array are stored contiguously in memory
  - To access any item in an array we simply use its index number
- Data items are easy and quick to access, but arrays are not very flexible
- *Can you think of some limitations of arrays?*
- Size is fixed, cannot "grow" to add more data
- Not easy to insert items in the middle
- Not easy to delete items in the middle

Need to "shift" other items to make room or close gaps

Advantages of Linked Lists

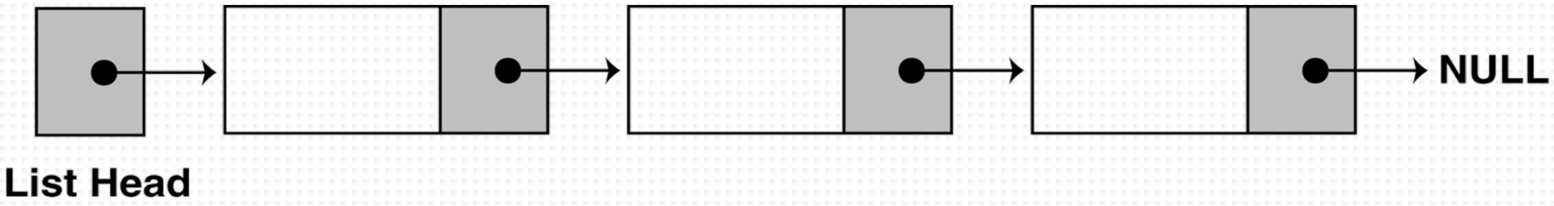
- For a big array resizing, inserting or deleting items is expensive in terms of CPU time – lots of data must often be moved

- **The linked list solves all of these problems!**

- Size can easily “grow” to add more data
  (no need to know or guess how big the list might get in advance)
- Easy and quick to insert items in the middle
- Easy and quick to delete items in the middle

What is a Linked List?

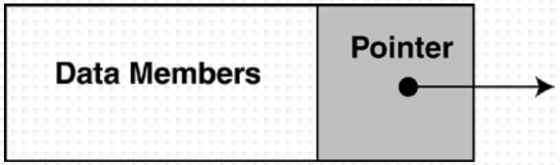
- A linked list is a data structure that stores data items, where each item has a pointer to the next item in the list
  - Each item is called a node
- The node's data can be of any type – int, double, a struct, several of these, or totally flexible by using void*
- The "next" pointer stores the next node's location in memory (*not* contiguous)
  - If we can access the first node, we can get to the next node



The Linked List Node

- The node is defined using a struct, so it can hold data (maybe more than one kind of data) and the next pointer

```
typedef struct node {
    int number;             // The data in this node
    struct node* next;      // Pointer to the next node
} Node;
```



- If the next pointer is NULL (zero) there are no more nodes after this one – this is the end of the list

Other node structure examples

- The linked list node can store any data, examples…

```
typedef struct node {   // Node for list of floating point data
    double number;
    struct node* next;  // Pointer to the next node
} Node;
```

```
typedef struct node {   // Node for list of players in a game
    char name[30];
    int score;
    int level;
    struct node* next;  // Pointer to the next node
} Node;
```

```
typedef struct node {   // Same as previous, using a 'nested' struct
    Player plr;
    struct node* next;  // Pointer to the next node
} Node;
```

```
typedef struct node {   // Node with arbitrary data (true ADT)
    void* data;
    struct node* next;  // Pointer to the next node
} Node;
```

Creating a Node

- To create a node we need to allocate memory for it using malloc, then initialize the fields
- Since we'll need to do this often let's make a function for it:

```
Node* createNode(int num) {
    Node* nd = (Node*)malloc(sizeof(Node));
    nd->number = num; // Put the value in the node
    nd->next = NULL;  // Next pointer is null for now
    return nd;     // Return a pointer to the new node
}
```

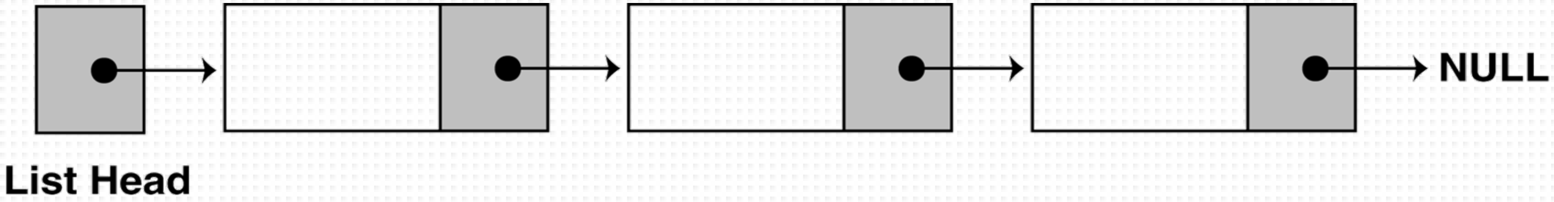
(Study this carefully to understand how it works!)

Creating a Linked List

- To create a linked list we need to
  1. Declare a pointer that points to the head (front) of the list
  2. Allocate a node using createNode (first node) and link it to the head
  3. Allocate other nodes and link them to the list

- Creating the list head pointer is easy – make it global/module var

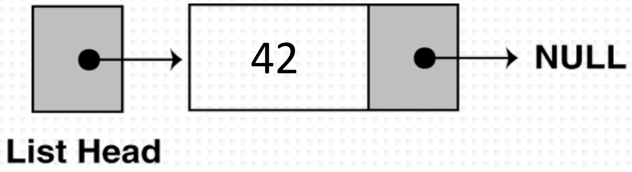
```
Node* head = NULL;
```



Creating a Linked List: First Node

- Here's how to add the first node to an empty list
  - We need to create a new node, then link it to the head

```
Node* nd = createNode(42);
head = nd;
```



- Now 'head' is not null anymore
  - What is `head->number` ?
  - What is `head->next` ?

Creating a Linked List: Adding More Nodes

- Adding nodes when the list is not empty takes a couple of steps
- Here's how to insert a new node at the head (front)

```
Node* nd = createNode(12);
nd->next = head;    // Link in the new node
head = nd;  // New node becomes the new head
```

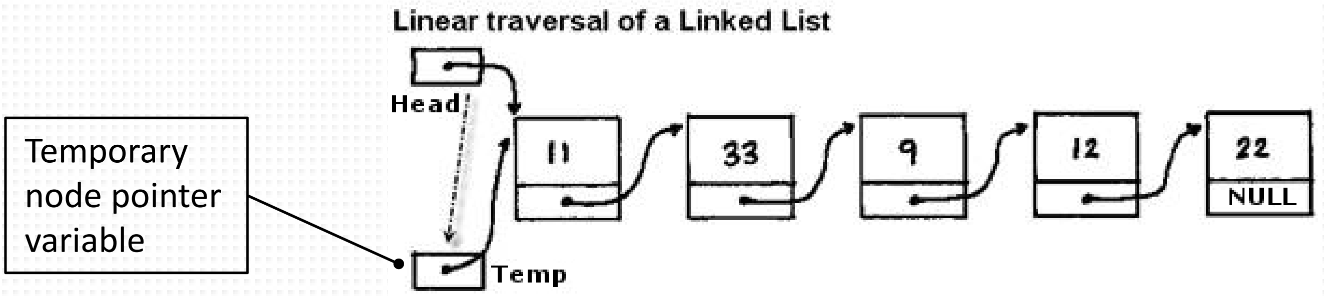
- Draw what the list looks like now!
- Note: This code works to add the first node as well

Traversing a Linked List

Just follow the pointers…

Traversing a Linked List

- Traversing a linked list means "visiting" all the nodes in order
- We need to traverse whenever we want to do something with all (or several of) the nodes, e.g.
  - Print out all the items in the list
  - Search through the list to find a specific item
  - Process or change each item in the list
- To traverse we normally start at the head and keep following "next" pointers until we reach a NULL



Traversing a Linked List

- For example, here's a function that traverses a linked list containing integers and prints all the data values
  - Assumes the starting node 'head' is a global or module variable

```
void printList(void) {
    Node* temp = head;  // Start at head
    // Loop as long as there are more nodes
    while (temp != NULL) {
        printf("%d\n", temp->number);
        temp = temp->next; // Go on to the next node
    }
}
```

(Study this carefully to understand how it works!)

Exercise 1: Add at head then traverse

- Write a C program that prompts the user to enter 5 numbers
- Add each number to the head of a linked list
  - When done, your list should contain 5 nodes
- Now traverse the list to print out all the numbers
  - What can you say about the order the numbers appear?

… Use the code given on the preceding slides!

Are there disadvantages? Yes.

- A linked list is not always better than an array when you want to store a list
- The code is more complex
  - Greater chance of bugs
  - If you use an existing, proven library instead of writing your own this is much less of a problem… ADT! (or a class in OO)
- Some operations are slower
  - Accessing data at a specific index number is slower than for an array because the data is not contiguous (*need to traverse*)
  - The CPU cache doesn't work as well for linked lists because "adjacent" items may actually be stored far apart in memory
  - Traversing backwards?

Inserting anywhere in a linked list

- We've seen how to insert new items at the head
  - The last item inserted appears first in the list
- Here's how to insert in the middle of a list or at the end
  - Assume the node pointer **prev** points to the node we want to insert after – *we have to know or find prev somehow*

```
Node* nd = createNode(1234);
nd->next = prev->next; // Link in the new node
prev->next = nd;
```

- Note: This code won't work to add a node at the head, or to add the first node

Exercise 2: Add at tail then traverse

- Write a C program that prompts the user to enter 5 numbers
- Add each number to the tail (end) of a linked list
  - Keep track of the 'prev' (last added) node!
  - When done, your list should contain 5 nodes
  - Remember you can't add the first node the same way you add the other nodes
- Now traverse the list to print out all the numbers
  - The numbers should appear in order (not reversed)!

Searching a Linked List

- Often we need to find a particular data item in a list
- This function searches for ‘value’ starting at node ‘nd’
  - It returns a pointer to the node containing the first match
  - If the value was not found it returns NULL

```
Node* searchList(Node* nd, int value) {
    Node* result = NULL;
    Node* temp = nd;  // Start at given node
    while (temp != NULL) {
        if (temp->number == value) {  // Found a match?
            result = temp;   // Return the current node
            break;
        }
        temp = temp->next; // Go on to the next node
    }
    return result;
}
```

Exercise 3: Search

- Extend your program from Exercise 2 … (save a copy first)
- After printing the numbers in the list, use the searchList function to search for the number 42
  - If the number is found, print "found!"
  - If the number isn't found, print "not found!"

- Hint: Remember the searchList function returns NULL if the number was ***not*** found, or some pointer that's not null if the number ***was*** found

Deleting Items From a Linked List

- Here's how to delete a particular item from the list
  - As with inserting items, deleting an item at the head is a bit different from deleting items elsewhere in the list
  - Since each node was allocated using malloc, we must remember to free the node when we delete it
- Suppose 'nd' points to the node to delete

```
if (nd == head) {  // Is this the first node?
    head = nd->next;
} else {
    prev->next = nd->next;
}
free(nd);  // Deallocated the deleted node
```

We have to know 'prev', the previous node, somehow

Exercise 4: Delete

- Extend your program from Exercise 2 … (save a copy first)
- After printing the numbers in the list, delete the 4th number in the list
  - Traverse to find the 4th node
  - Delete that node
  - Print the list again to be sure the 4th item is gone
- Hint: Remember while traversing that you'll need both a pointer to the node to delete and a pointer to the previous node!
- Variation: Delete all the numbers in the list

Additional Exercise A: Maintaining a Sorted List

This exercise is more challenging …

- Write a program that inputs numbers from the user
- Insert each number into a linked list, but keep the numbers sorted from smallest to largest
  - You'll have to find the right place to insert each number
- Stop your input loop when the user enters 999 (the sentinel value)
- Print the entire list so you can check if it's sorted

Additional Exercise B: Linked List Ops

Write a program that implements a linked list of strings (each node contains one string). Then write functions to do the following linked list operations:

- A function named print that prints the items in the list.
- A function named size that returns the size (number of nodes) in the linked list.
- A function lookup that checks if a given string is contained in the linked list.
- An add function that adds a string at the end of the list if it is not already contained in the linked list.
- A remove function that removes a given string from the list.

Also write a main function to test the other functions.

Doubly Linked Lists

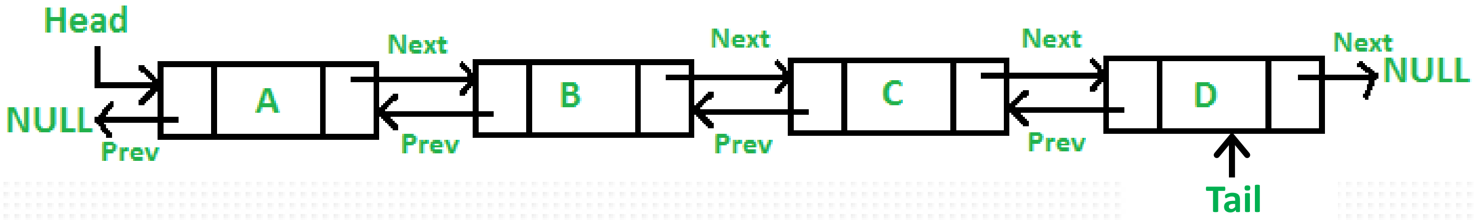
Traverse in both directions

Limitations of a single 'next' pointer

- The linked list we've studied so far is a singly linked list
  - Each node has exactly one 'next' pointer, pointing to the next item towards the tail of the list
- Did you notice it was sometimes tricky to know the "previous" node?
- How would you traverse this linked list *backwards*?
  - From tail to head
- Let's add another 'next' pointer that points in the other direction – to the previous node

Doubly Linked List

- With a doubly linked list we can traverse backwards or forwards
- Each node has two pointers: next, previous
  - Operations which required some indirect way to know the previous node (like delete) are now simpler
- In addition to the head pointer which tells where the list starts, we'll also keep a tail pointer



Doubly Linked List Node

- Node definition for a doubly linked list containing integers:

```
typedef struct node {
    int number;           // The data in this node
    struct node* next;    // Pointer to the next node
    struct node* prev;    // Pointer to the previous node
} Node;
```

- If either pointer is NULL (zero) there are no more nodes in that direction

Creating a Node

- Creating a node is similar to before
- Both pointers will be null so it's not "linked in" yet

```
Node* createNode(int num) {
    Node* nd = (Node*)malloc(sizeof(Node));
    nd->number = num; // Put the value in the node
    nd->next = NULL;  // Next pointer is null for now
    nd->prev = NULL;  // Prev pointer is null for now
    return nd;     // Return a pointer to the new node
}
```

Creating a Doubly Linked List

- To create a doubly linked list we need to
  1. Declare a pointer that points to the head (front) of the list
  2. Declare a pointer that points to the tail (end) of the list
  3. Allocate nodes as needed and add them to the list. If a node gets added at the head, the head pointer changes. If a node gets added at the tail, the tail pointer changes

- Now create the head and tail pointers
  - Make them global or module variables

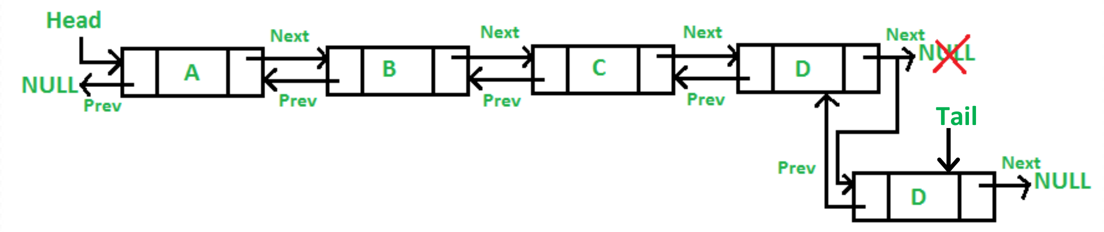
```
Node* head = NULL;

Node* tail = NULL;
```

Adding Nodes to a Doubly Linked List

- This function appends a new node at the end of the list
  - You need to pass it a data value to put in the new node

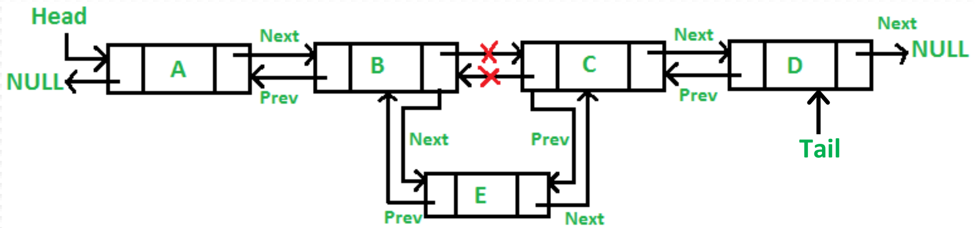
```
void appendNode(int num) {
    Node* nd = createNode(num);  // Allocate new node 'nd'
    if (head == NULL) {   // If list is empty
        head = nd;   // New node is the head (and tail)
    } else {
        tail->next = nd; // Make old tail node point at new one
        nd->prev = tail; // Make new node point back at old tail
    }
    tail = nd;   // New node becomes the new tail
}
```



Inserting Nodes in a Doubly Linked List

- This function inserts a new node after any given node
  - You need to pass it an existing node, and a value
  - Note: ‘loc’, ‘head’, and ‘tail’ must not be null

```
void insertNode(Node* loc, int num) {
    Node* nd = createNode(num);  // Allocate new node 'nd'
    if (loc->next != NULL) {
        loc->next->prev = nd; // Node after loc points back to nd
    }
    nd->next = loc->next; // New node points to node after loc
    loc->next = nd;  // loc now points to the new node
    nd->prev = loc;  // New node points back to loc
    if (loc == tail) { // If at end, new node becomes new tail
        tail = nd;
    }
}
```



Traversing a Doubly Linked List

- Here's a function that traverses a linked list containing integers in either direction and prints the data values
  - You need to pass it the starting node and direction

```
void printList(Node* nd, bool forward) {
    // Loop as long as there are more nodes
    Node* temp = nd;
    while (temp != NULL) {
        printf("%d\n", temp->number);
        if (forward)
            temp = temp->next; // Go to the next node
        else
            temp = temp->prev; // Go to the previous node
    }
}
```

Deleting Items From a Doubly Linked List

- Here's a function to delete a particular node
  - Note: 'nd', 'head', and 'tail' must not be null

```
void deleteNode(Node* nd) {
    if (nd == head) {   // Deleting head node?
        head = nd->next;
    }
    if (nd == tail) {   // Deleting tail node?
        tail = nd->prev;
    }
    if (nd->next != NULL) {   // Is there a next node?
        nd->next->prev = nd->prev;
    }
    if (nd->prev != NULL) {   // Is there a previous node?
        nd->prev->next = nd->next;
    }
    free(nd);  // Deallocate the deleted node
}
```

Exercise 5: Doubly Linked List

Write a C program that does the following:

- Prompt the user to enter 5 numbers and append them to a doubly linked list
- Insert the number 42 after the head of the list
  - It should then be the second item in the list
- Print the entire list both forward and backward
- Delete the items at the head and the tail of the list
- Print the list again – should be only 4 items left

… Use the code on the preceding slides!