Linked List Library Documentation

// file name: linked\_list\_all\_functions\_20240416.c

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <time.h>

typedef struct node {

int value;

struct node \*next;

} Node;

// Print functions:

void print\_list(Node \*head);

// Insert and Delete functions:

Node \*insert\_at\_head(Node \*head, int new\_value);

Node \*insert\_at\_tail(Node \*head, int new\_value);

Node \*delete\_at\_head(Node \*head);

Node \*delete\_at\_tail(Node \*head);

Node \*delete\_first\_match(Node \*head, int delete\_value, bool \*was\_deleted);

Node \*efficient\_delete\_match(Node \*head, int delete\_value, int \*num\_deleted);

void delete\_duplicates(Node \*head); // NEW

Node \*insert\_after(Node \*head, int new\_value, int after\_value); // NEW

Node \*delete\_list(Node \*node); // NEW

// Append functions:

Node \*append\_list(Node \*head1, Node \*head2);

// Reverse Functions:

Node \*reverse\_list(Node \*head);

// Sort Functions:

void sort\_list(Node \*head); // "Bubble" sort method - classic array sorting method.

// Length, Search, Count, and Replace functions:

int length(Node \*head);

int recursive\_length(Node \*node);

bool is\_member\_r(Node \*node, int find\_value);

int count\_matches\_r(Node \*node, int find\_value);

void replace\_matches\_r(Node \*node, int find\_value, int replace\_value);

int main(void)

{

Node \*list1\_head = NULL;

Node \*list2\_head = NULL;

// Load list1 from the head and tail.

list1\_head = insert\_at\_head(list1\_head, 10);

list1\_head = insert\_at\_head(list1\_head, 5);

list1\_head = insert\_at\_head(list1\_head, 3);

list1\_head = insert\_at\_tail(list1\_head, 10);

list1\_head = insert\_at\_tail(list1\_head, 12);

list1\_head = insert\_at\_tail(list1\_head, 15);

list1\_head = insert\_at\_tail(list1\_head, 16);

list1\_head = insert\_at\_tail(list1\_head, 30);

list1\_head = insert\_at\_tail(list1\_head, 30);

// Load list2 from the head and tail.

list2\_head = insert\_at\_tail(list2\_head, 14);

list2\_head = insert\_at\_tail(list2\_head, 30);

list2\_head = insert\_at\_tail(list2\_head, 30);

list2\_head = insert\_at\_tail(list2\_head, 14);

list2\_head = insert\_at\_tail(list2\_head, 30);

printf("List before deletions of 5 or 50...\n");

print\_list(list1\_head);

bool deleted;

// Test value of the boolean variable 'deleted.'

if (deleted)

printf("bool deleted value is true.\n");

else if (deleted)

printf("bool deleted value is false.\n");

else

printf("bool deleted value has no value.\n");

list1\_head = delete\_first\_match(list1\_head, 5, &deleted); // The memory address of the

if (deleted) // empty boolean 'deleted' is

printf("A node with value 5 was deleted.\n\n"); // passed to the delete...

else // and is returned set to true

printf("A node with value 5 was not deleted.\n\n"); // or false.

printf("List after deleting 5...\n\n");

print\_list(list1\_head);

list1\_head =

delete\_first\_match(list1\_head, 50, &deleted);

if (deleted)

printf("A node with value 50 was deleted.\n\n"); // If deleted is true...

else printf("A node with value 50 was NOT deleted.\n\n"); // ...otherwise

printf("List after deletions of 5 and/or 50...\n");

print\_list(list1\_head);

printf("\nList before head deletion.\n");

print\_list(list1\_head);

list1\_head = delete\_at\_head(list1\_head);

printf("\nList after deleting head.\n");

print\_list(list1\_head);

list1\_head = delete\_at\_tail(list1\_head);

printf("\nList after deleting head and tail.\n");

print\_list(list1\_head);

printf("\nLength of list: %d\n", length(list1\_head));

printf("Length of list (found with a recursive function): %d\n\n",

recursive\_length(list1\_head));

if (is\_member\_r(list1\_head, 10))

printf("10 found in list\n");

else

printf("(10 note found in list)\n");

if (is\_member\_r(list1\_head, 22))

printf("22 found in list\n");

else

printf("22 not found in list\n\n");

printf("The value 10 was found %d time(s).\n\n", count\_matches\_r(list1\_head, 10));

replace\_matches\_r(list1\_head, 10, 30);

printf("List after replacing 10 with 30.\n");

print\_list(list1\_head);

int num\_deleted = 0;

list1\_head = efficient\_delete\_match(list1\_head, 30, &num\_deleted);

printf("\nList after deleting all 30s.\n");

print\_list(list1\_head);

printf("Number of deleted 30s: %d\n", num\_deleted);

// Test performance.

// Create two large lists:

Node \*listX = NULL, \*listY = NULL; // Declare two empty lists.

for (int i = 0; i < 50000; i++) // Insert 50,000 nodes...

listX = insert\_at\_head(listX, i % 10); // looping in values from 0 to 9.

for (int i = 0; i < 100000; i++)

listY = insert\_at\_head(listY, i % 10);

clock\_t tic, toc;

tic = clock();

listX = efficient\_delete\_match(listX, 4, &num\_deleted);

toc = clock();

printf("\nefficient\_delete\_matches in 50,000 nodes: %fs\n", (double)(toc - tic) / CLOCKS\_PER\_SEC);

printf("elements deleted: %d\n", num\_deleted);

tic = clock();

listY = efficient\_delete\_match(listY, 4, &num\_deleted);

toc = clock();

printf("\nefficient\_delete\_matches in 100,000 nodes: %fs\n", (double)(toc - tic) / CLOCKS\_PER\_SEC);

printf("elements deleted: %d\n", num\_deleted);

list1\_head = append\_list(list1\_head, list2\_head);

printf("\nList #1 after appending list #2:\n");

print\_list(list1\_head);

list1\_head = reverse\_list(list1\_head);

printf("\nPrint list after reversal:\n");

print\_list(list1\_head);

sort\_list(list1\_head); // No "list1\_head =" because this is "void" function like print\_list.

printf("\nPrint list after sort:\n");

print\_list(list1\_head);

delete\_duplicates(list1\_head); // Delete duplicates.

printf("\nPrint list after deleting duplicates:\n");

print\_list(list1\_head);

list1\_head = insert\_after(list1\_head, 34, 15);

printf("\nPrint list after inserting 34 after 15:\n");

print\_list(list1\_head);

list1\_head = insert\_after(list1\_head, 35, 23);

printf("\nPrint list after inserting 35 after 23 (which is not in the list, so it goes to the tail):\n");

print\_list(list1\_head);

list1\_head = delete\_list(list1\_head);

printf("\nPrint list after deleting the whole list:\n");

if (list1\_head == NULL)

{

printf("(No list to print.)\n");

}

else

{

printf("(There's still something in the list.)\n");

}

print\_list(list1\_head);

/\*

printf("\nPrint second list if anything is there:\n");

if (list2\_head == NULL)

{

printf("(No list to print.)\n");

}

else

{

printf("There's still something in the second list even though it was appended to list 1 "

"and is throwing a segmentation error.\n");

}

print\_list(list2\_head);

\*/

return 0;

}

// Print Functions:

void print\_list(Node \*head)

{

Node \*current;

current = head;

int i = 0;

while (current != NULL)

{

printf("(Node%3d: %3d)\n", i, current->value );

i++; // Increment the virtual index.

current = current->next;

}

printf("\n");

}

// Insert Functions:

Node \*insert\_at\_head(Node \*head, int new\_value)

{

Node \*new\_node = calloc(1, sizeof(Node));

new\_node->value = new\_value;

if (head == NULL) return new\_node;

else

{

new\_node->next = head;

return new\_node;

}

}

Node \*insert\_at\_tail(Node \*head, int new\_value)

{

Node \*new\_node = calloc(1, sizeof(Node));

new\_node->value = new\_value;

if (head == NULL) return new\_node;

else

{

Node \*current = head;

while (current->next != NULL) current = current->next;

current->next = new\_node;

return head;

}

}

// Delete Functions:

Node \*delete\_at\_head(Node \*head)

{

if (head == NULL) return NULL;

else

{

Node \*to\_return = head->next;

free(head);

return to\_return;

}

}

Node \*delete\_at\_tail(Node \*head)

{

if (head == NULL) return NULL;

else if (head->next == NULL) // If head->next is null, there's only one Node.

{

free(head); // Delete the one Node.

return NULL;

}

else

{

Node \*current = head;

Node \*prev = NULL;

while (current->next != NULL)

{

prev = current;

current = current->next;

}

prev->next = NULL;

free(current);

return head;

}

}

Node \*efficient\_delete\_match(Node \*head, int delete\_value, int \*num\_deleted)

{

\*num\_deleted = 0; // Why does \*num\_deleted continue to be referenced with an \*

// but, say head, which was also passed with an \* is not?

if (head == NULL) return NULL; // If list is empty, return NULL.

Node \*current, \*temp, \*new\_head;

current = head;

\*num\_deleted = 0; // Pass by reference (using the memory address);

while (current->value == delete\_value)

{

temp = current;

current = current->next;

free(temp);

\*num\_deleted = \*num\_deleted + 1;

if (current == NULL) return NULL;

}

new\_head = current;

while (current->next != NULL)

{

if (current->next->value == delete\_value)

{

temp = current->next;

current->next = current->next->next;

free(temp);

\*num\_deleted = \*num\_deleted + 1;

}

else current = current->next;

}

return new\_head;

}

// This function deletes that first Node that matches the value passed to it.

// It handles four cases: (1) the list is empty, (2) the head has the value, so it must

// be deleted and the second Node's pointer must be sent back as the new head, (3) the

// value is found in a later position, so that the ->next pointer in the previos Node

// must be pointed to the Node after the Node that's deleted, or (4) the value is not

// found in the list at all.

Node \*delete\_first\_match(Node \*head, int delete\_value, bool \*was\_deleted)

{

if (head == NULL)

{

\*was\_deleted = false;

return NULL;

printf("No nodes with value X found.\n");

}

if (head->value == delete\_value)

{

Node \*temp = head->next;

free(head);

\*was\_deleted = true;

return temp; // When the head is deleted, we want to report that whatever Node it

// was pointing to is now the new head.

}

Node \*current = head->next;

Node \*prev = head;

while (current != NULL)

{

if (current->value == delete\_value)

{

prev->next = current->next;

free(current);

\*was\_deleted = true;

return head;

}

prev = current;

current = current->next;

}

\*was\_deleted = false;

return head;

}

// Append Functions:

Node \*append\_list(Node \*head1, Node \*head2)

{

if (head1 == NULL) return head2;

Node \*current = head1;

while (current->next != NULL) current = current->next; // Find the end of list 1.

current->next = head2; // Point the tail of list 1 to the head of list 2 to connect them.

return head1;

}

// Reverse Functions:

Node \*reverse\_list(Node \*head)

{

if (head == NULL) return NULL; // If list is empty, there's nothing to reverse.

if (head->next == NULL) return head; // If list has one node, return the old

// head as still in the head position.

Node \*current = head; // Create current pointer to traverse the list.

Node \*next\_node = head->next; // Create a holder for the pointer next in the list.

current->next = NULL; // Set the head node's next point to NULL, making it the tail.

while (next\_node != NULL)

{

Node \*tmp = next\_node->next;

next\_node->next = current; // The next node's next value gets pointed to the former head.

current = next\_node; // The current (index holder) gets moved to the node's address.

next\_node = tmp; // The next\_node pointer gets moved forward like current just did.

}

return current; // At the end, current has moved to the former tail, and is now the head.

}

// Sort Functions:

// Bubble Sort:

void sort\_list(Node \*head)

{

if (head == NULL) return ; // If there's nothing to sort, return nothing (nothing sorted).

if (head->next == NULL) return ; // If there's only one node, return nothing (nothing sorted).

bool swapped = false;

do

{

swapped = false;

Node \*current = head;

Node \*prev = NULL;

while (current->next != NULL)

{

prev = current;

current = current->next;

if (current != NULL)

{

if (current->value < prev->value)

{

int temp;

temp = prev->value;

prev->value = current->value;

current->value = temp;

swapped = true;

}

}

}

} while (swapped);

}

// Find List Length, Count Matches, Replace Matches

int length(Node \*head)

{

Node \*current;

current = head;

int length = 0;

while (current != NULL)

{

current = current->next;

length++;

}

return (length);

}

int recursive\_length(Node \*node)

{

if (node == NULL) return 0;

else return 1 + recursive\_length(node->next);

}

bool is\_member\_r(Node \*node, int find\_value)

{

if (node == NULL) return false; // Value not found in an empty list.

else if (node->value == find\_value) return true; // If value is found, return

// true. Run until true.

else return is\_member\_r(node->next, find\_value); // If value never found,

} // tail node with .next =

// NULL will trigger first

// line to return false.

int count\_matches\_r(Node \*node, int find\_value) // Find the number of times a value

{ // is found in the list. (Recursive.)

if (node == NULL) return 0;

else if (node->value == find\_value)

return 1 + count\_matches\_r(node->next, find\_value);

else return count\_matches\_r(node->next, find\_value);

}

void replace\_matches\_r(Node \*node, int find\_value, int replace\_value)

{

if (node != NULL) // If the head node is NULL (indicating an empty list) do nothing.

{

if (node->value == find\_value) node->value = replace\_value;

replace\_matches\_r(node->next, find\_value, replace\_value);

}

}

// Added Delete function:

void delete\_duplicates(Node \*head)

{

if (head == NULL) return ; // If the list is emtpy, do nothing.

if (head->next == NULL) return ; // If there's only one node, do nothing.

// There are two loops below. The outside loop tranverses the list once, stopping

// on each Node and checking (with the inside loop) to see if any of the other

// Nodes has the same value.

Node \*current1, \*current2, \*duplicate; // Declare pointers to track the nodes.

current1 = head; // Set the current1 Node pointer to the first Node.

while (current1 != NULL && current1->next != NULL)

{

current2 = current1;

while (current2->next != NULL) // Restart when inner loop reaches end of list.

{

if (current1->value == current2->next->value)

{

duplicate = current2-> next; // Store the pointer of the Node to be deleted.

current2->next = current2->next->next; // Connect the Node to the next, next

// Node before deleting the one in between.

free(duplicate); // Delete the duplicate Node.

}

else current2 = current2->next; // Ratchet the inner loop forward.

}

current1 = current1->next; // Ratchet the out loop forward.

}

}

// Added Insert function:

// This function is designed to insert the new value as the head if the list was

// previously empty, and insert the new value at the tail if none of the Nodes has

// the value specified to insert after.

Node \*insert\_after(Node \*head, int new\_value, int after\_value)

{

Node \*new\_node = calloc(1, sizeof(Node)); // Create the new Node on the heap.

new\_node->value = new\_value; // Set the value of the new Node to the value passed to the function.

if(head == NULL) return new\_node; // If the list is empty, return the memmory address of the new

// Node as the head address.

else

{

Node \*current = head; // Instantiate the current node tracker, set equal to the head Node.

while (current->next != NULL) // While we're not at the end of the list...

{

if (current->value == after\_value) // If the current Node's value is the value we want to insert

{ // the new Node after...

new\_node->next = current->next; // set the new Node's next pointer to the current Node's next pointer...

current->next = new\_node; // and aim the current Node's next pointer at the new Node.

return head; // Return the old (unchanged) head memory address.

}

else current = current->next; // Otherwise, move on the next Node to search for the "after value."

}

new\_node->next = current->next; // If we've gotten all the way to the end of the list (without finding

// the "after value"...

current->next = new\_node; // aim the last Node next pointer to the new Node.

return head; // Return the old (unchanged) head memory address.

}

}

// Add Delete function (that deletes entire list):

Node \*delete\_list(Node \*node)

{

if (node != NULL) // Traverse the list...

{

delete\_list(node->next); // Save the next Node's memory address.

free(node); // Delete the current address.

}

return NULL;

}