5. Linked Lists

Oliver Diessel
odiessel@cse.unsw.edu.au

Last updated: 12:00 28 Aug 2008
Linked Lists

- Topics to be covered:
  - Self-referential structures
  - Linked Lists
  - List operations
  - Stacks
  - Ordered lists

Slide credits: Alan Blair
Self-Referential Structures

- We can define a structure containing within it a pointer to the same type of structure

```c
typedef struct lnode Lnode;
struct lnode {
    int data;
    Lnode *next;
};
```

- These “self-referential” pointers can be used to build larger “dynamic” data structures out of smaller building blocks
Linked Lists

- The most fundamental of these dynamic data structures is the Linked List
  - based on the idea of a sequence of data items or nodes
  - linked lists are more flexible than arrays
    - items don’t have to be located next to each other in memory
    - items can easily be rearranged by altering pointers
    - the number of items can change dynamically
    - items can be added or removed in any order

First element of list

Next data

Next data

Next data

Next data

NULL
The most fundamental of these dynamic data structures is the Linked List
- based on the idea of a sequence of data items or nodes
- linked lists are more flexible than arrays
  - items don’t have to be located next to each other in memory
  - items can easily be rearranged by altering pointers
  - the number of items can change dynamically
  - items can be added or removed in any order

We will look at how to create lists and some useful operations for manipulating them
Linked Lists

- a linked list is a sequence of items
- each item of the list contains data and a pointer to the next item
- also need to maintain a pointer to the first item or “head” of the list
- the last item in the list points to NULL
- need to distinguish between the node and the data; the node is like a “container” which holds the data inside it
Linked List Node

Examples of list nodes:

Address Book

<table>
<thead>
<tr>
<th>next</th>
<th>name</th>
<th>address</th>
<th>telephone</th>
<th>email</th>
</tr>
</thead>
</table>

| next | ID | component_type | num_of |

Bill of Materials
typedef struct addressNode AddressNode;

struct addressNode {
    AddressNode *next;
    char *name;
    char *address;
    char *telephone;
    char *email;
};

Address Book
Linked List Node Structure in C

typedef struct addressNode AddressNode;

struct addressNode {
    AddressNode *next;
    char *name;
    char *address;
    char *telephone;
    char *email;
};

**Note1:** the `typedef` must appear before the structure definition in order to be able to include a self-referential pointer within the `struct`.

**Note2:** this example illustrates another common “type” naming convention i.e. use of upper case for leading letters of descriptive terms.
Exercise

Write a suitable definition for a BOM

typedef struct addressNode AddressNode;

struct addressNode {
    AddressNode *next;
    char *name;
    char *address;
    char *telephone;
    char *email;
};
List Operations

- Fundamental List operations:
  - create a new node with specified data
  - search for a node with particular data
  - insert a new node into the list
  - remove a node from the list

- Other operations are possible and can be added as needed
- Lists also form the basis for useful data structures like stacks and queues
List Operations

Lnode * makeNode(int data); // create new node

Lnode * findNode(int data, Lnode *head);

Lnode * push(Lnode *new_node, Lnode *head); // to front

Lnode * pop(Lnode *head); // first item

void    printList(Lnode *head); // print all items
void    freeList(Lnode *head); // clear entire list

Lnode * insert(Lnode *new_node, Lnode *head); // in order

Lnode * excise(Lnode *old_node, Lnode *head);
Making a new node

/*
   Create a new node containing the specified data,
   and return a pointer to this newly-created node.
*/

Lnode * makeNode( int data ) {
    Lnode *new_node = (Lnode*)malloc(sizeof(Lnode));
    assert(new_node != NULL);
    new_node->data = data;
    new_node->next = NULL; // ensure next->NULL!
    return (new_node);
}
Finding a Node in a List

/* Search through list to find the first node with the specified data, and return a pointer to this node.
If no such node exists, return NULL. */

Lnode * findNode(int data, Lnode *head) {
    Lnode * node = head; // start at first node in list
    // keep searching until data found, or end of list
    while ((node != NULL) && (node->data != data)) {
        node = node->next;
    }
    return (node);
}

Note short-circuit check that node exists before testing data or assigning next to node
Push a Node onto the Front of a List

Pushing a new item involves two operations:
1. make the new node point to the current head of the list
2. make the new node become the new head of the list
Push a Node onto the Front of a List

/* Push new node to front of the list and return the resulting (longer) list */

Lnode * push( Lnode *new_node, Lnode *head ) {
    new_node->next = head;
    return (new_node);
}

- Since this function returns the new list, it should be called like this:

    list = push( makeNode(‘A’), list );
Pop the First Node from a List

/* Pop first item from list and return the remaining (shorter) list */

Lnode * pop( Lnode *head ) {
    Lnode *tmp;

    if (head != NULL) { // check list isn’t empty
        tmp = head;
        head = head->next;
        free(tmp);
    }
    return (head);
}
Printing a List

/*   Print all items in the list one by one */

void printList( Lnode *head ) {
    Lnode *node = head;

    // traverse the list printing each node in turn
    while (node != NULL) {
        printf( "->%c", node->data );
        node = node->next;
    }
    printf("\n");
}

Deleting all items from a List

/* Delete all items from a linked list */

void freeList( Lnode *head ) {
    Lnode *node = head;
    Lnode *tmp;

    while ( node != NULL ) {
        tmp = node;
        node = node->next;
        free(tmp);
    }
}
Example: stack.c

```c
int main( void ) {
    Lnode *list = NULL;
    int ch;

    while (( ch = getchar()) != EOF ) {
        if ( ch == '-' )
            list = pop(list);
        else if (ch = '
')
            printList(list);
        else
            list = push( makeNode(ch), list);
    }
    freeList(list);
}
```
Insert a node into an ordered list

Lnode * insert( Lnode *new_node, Lnode *head ) {
    Lnode *next_node = head;
    while (new_node->data > next_node->data) {
        prev_node = next_node;
        next_node = next_node->next;
    }
    new_node->next = next_node;
    return (head);
}

Problem: need to keep track of previous node!
insert() – version 2

Lnode * insert( Lnode *new_node, Lnode *head ) {
    Lnode *next_node = head, *prev_node;
    while (new_node->data > next_node->data) {
        prev_node = next_node;
        next_node = next_node->next;
    }

    prev_node->next = new_node;
    new_node->next = next_node;
    return (head);
}

Problem: what if new node goes at the end?
**insert() – version 3**

```c
Lnode * insert( Lnode *new_node, Lnode *head ) {
    Lnode *next_node = head, *prev_node;
    while (next_node
        && new_node->data > next_node->data) {
        prev_node = next_node;
        next_node = next_node->next;
    }
    prev_node->next = new_node;
    new_node->next = next_node;
    return (head);
}
```

**Problem:** what if new node goes at the beginning?
insert() – final version

Lnode * insert( Lnode *new_node, Lnode *head ) {
    Lnode *next_node = head, *prev_node = NULL;
    while (next_node && new_node->data > next_node->data) {
        prev_node = next_node;
        next_node = next_node->next;
    }
    if (prev_node == NULL)
        head = new_node;
    else
        prev_node->next = new_node;
    new_node->next = next_node;
    return (head);
}

Exercise0: what are all cases?
Exercise1: check this works in all cases.
Remove a Node from a Ordered List

Lnode * excise ( Lnode *node, Lnode *head ) {

  /*
   * Remove the node pointed to by "node"
   * from the ordered list pointed to by "head" and
   * return a pointer to the new head of the list
   */

}
Remove a Node from a Ordered List

```c
Lnode * excise ( Lnode *node, Lnode *head ) {
    if (node != NULL) {
        if (node == head)
            head = head->next; // remove first item
        else {
            Lnode *prev_node = head;
            while (prev_node
                && prev_node->next != node) {
                prev_node = prev_node->next;
            }
            if ( prev_node != NULL ) { // node found in list
                prev_node->next = node->next;
            }
        }
    }
    return (head);
}
```
Exercise

- Check that \texttt{excise()} behaves sensibly in all of these cases:
  - removing the first item
  - removing the last item
  - removing an interior item
  - node is not in list
  - node is NULL
  - list is empty
  - node is NULL and list is empty.
Example: `ordered.c`

```c
int main (void) {
    Lnode *list = NULL;
    Lnode *node;
    int ch;

    while ((ch = getchar()) != EOF) {
        if ( ch == '-' ) { // remove item from list
            ch = getchar();
            node = findNode( ch, list );
            if (node != NULL) {
                list = excise( node, list );
                free( node );
            }
        }
    }
}
```
... 

else if ( ch == '\n' ) { 
    printList(list);
} else {
    list = insert( makeNode(ch), list );
}

freeList( list );

return 0;
}
Where to from here?

- Take a look at Moffat Ch 10.2 on Linked Structures
- Queues