Self-Referential Structures

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

```c
typedef struct list_node list_node;

struct list_node {
    int data;
    list_node *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

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

Linked List

- 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

Example of a list node:

```
| next | name | address | telephone | email |
```

Linked List Node Structure in C

```
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 to 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

```
list_node *makeNode(int data);
list_node *findNode(int data, list_node *head);
list_node *push(list_node *newNode, list_node *head);
list_node *pop(list_node *head);
void printList(list_node *head);
void freeList(list_node *head);
list_node *insert(list_node *newNode, list_node *head);
list_node *excise(list_node *oldNode, list_node *head);
```
Making a New Node

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

list_node *makeNode(int data)
{
    list_node *newNode = (list_node *)malloc(sizeof(list_node);
    if (newNode == NULL) {
        printf(stderr, "Out of memory\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

Finding a Node in a List

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

list_node *findNode(int data, list_node *head)
{
    // start at first node
    list_node *node = head;

    // keep searching until data or end found
    while (node != NULL && node->data != data) {
        node = node->next;
    }
    return node;
}

Recursive version of findNode()

// First check the head. Then check the rest // by (recursively) calling the function itself!

list_node *findNode(int data, list_node *head)
{
    if (head == NULL || head->data == data) {
        return head;
    } else {
        return findNode(data, head->next);
    }
}

Question: Will this function keep calling itself, to infinity? Why not?

Pushing a Node onto the Front of a List

Pushing a new item involves two operations:
- make the new node point to the current head of the list
- 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 list and
// return the resulting (longer) list

list_node *push(list_node *newNode, list_node *head) {
    newNode->next = head;
    return newNode;
}
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

list_node *pop(list_node *head) {
    list_node *tmp = head;
    if (head != NULL) {
        head = head->next;
        free(tmp);
    }
    return head;
}

Printing a List

// Print all items in the list one by one

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

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

Recursive version of printList()

// First print the head, then print the rest

void printList(list_node *head) {
    if (head != NULL) {
        // avoid "infinite descent"
        printf("-%c", head->data);
        printList(head->next);
    } else {
        printf("\n");
    }
}
Deleting all items from a List

```c
// Delete all the items from a linked list.

void freeList(list_node *head) {
    list_node *node = head;
    list_node *tmp;
    while (node != NULL) {
        tmp = node;
        node = node->next;
        free(tmp);
    }
}
```

Example: stack.c

```c
int main(void) {
    list_node *list = NULL;
    int ch;
    while (((ch = getchar()) != EOF) {
        if (ch == '-') {
            list = pop(list);
        } else if (ch == '\n') {
            printList(list);
        } else {
            list = push(makeNode(ch), list);
        }
    }
    freeList(list);
}
```

Insert a Node into an Ordered List

```c
list_node *insert(list_node *newNode, list_node *head) {
    list_node *nextNode = head;
    // find correct position
    while (newNode->data > nextNode->data) {
        nextNode = nextNode->next;
    }
    // link new node into list
    newNode->next = nextNode;
    return head;
}
```

Problem: need to keep track of previous node!

insert() - version 2

```c
list_node *insert(list_node *newNode, list_node *head) {
    list_node *nextNode = head, *prev_node;
    // find correct position
    while (newNode->data > nextNode->data) {
        prev_node = nextNode;
        nextNode = nextNode->next;
    }
    // link new node into list
    prev_node->next = newNode;
    newNode->next = nextNode;
    return head;
}
```

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

```
list_node *insert(list_node *newNode, list_node *head) {
    list_node *nextNode = head, *prev_node;

    // find correct position
    while((nextNode && newNode->data > nextNode->data) {
        prev_node = nextNode;
        nextNode = nextNode->next; // find correct position
    }
    // link new node into list
    prev_node->next = newNode;
    newNode->next = nextNode;
    return head;
}
```

Problem: what if new node goes at the beginning?

**insert() - final version**

```
list_node *insert(list_node *newNode, list_node *head) {
    list_node *nextNode = head, *prev_node;

    // find correct position
    while((nextNode && newNode->data > nextNode->data) {
        prev_node = nextNode;
        nextNode = nextNode->next;
    }
    // link new node into list
    if (prev_node == NULL) {
        head = newNode;
    } else {
        prev_node->next = newNode;
    }
    newNode->next = nextNode;
    return head;
}
```

Exercise: check this works in all cases.

**Remove a Node from a List**

```
list_node *excise(list_node *node, list_node *head) {
    if (node != NULL) {
        if (node == head) {
            head = head->next;  // remove first item
        } else {
            list_node *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 `excise()` behaves sensibly in all of these cases:
- removing first item
- removing last item
- removing interior item
- node is not in list
- node is NULL
- list is empty
- node is NULL AND list is empty.
```c
int main(void) {
    list_node *list = NULL;
    list_node *node;
    int ch;
    while ((ch = getchar()) != EOF) {
        if (ch == '-') { // remove item from list
            ch = getchar();
            node = findNode(ch, list);
            if (node != NULL) {
                list = exercise(node, list);
                free(node);
            }
        } else if (ch == '\n') {
            printList(list);
        } else {
            list = insert(makeNode(ch), list);
        }
    }
    freeList(list);
    return 0;
}
```