**Modules in C**

- A **module** is a set of type, variable, and function definitions that are only accessible through a defined **interface**

- The programming language C has **no module system**

- Conventions allow emulation of a module system to a certain extent (Textbook 14.5)

- A module **Name** is represented by two files:
  - The **header file** `Name.h` contains type definitions, prototypes of exported functions, and *(only if really necessary)* `extern` declarations of exported global variables.
  - The **implementation file** `Name.c` contains the definitions of the exported functions (and exported global variables), and **static** definitions of non-exported functions and global variables.

**Compilation of Multi-File Programs**

- You **edit** `myprogram.c`

- You **compile**: `cc -o myprogram myprogram.c`
  - **Preprocessor** generates **preprocessed source** (`myprogram.i`, `utils.i`), typically each including `utils.h`
  - **Compiler proper** generates **assembly modules** (`myprogram.s`, `utils.s`)
  - **Assembler** generates **object code** (`myprogram.o`, `utils.o`)

- You **run** it: `/myprogram`
  - **Operating system** generates a new process
  - **Dynamic linker** resolves references to shared libraries
  - **Loader** generates **executable in-memory image**
  - **CPU** runs machine code

**How Does a Computer Run Your Program?**

- You **edit** `myprogram.c`

- You **compile**: `cc -c myprogram.c` `cc -c utils.c`
  - Preprocessor generates **preprocessed source** (`myprogram.i`, `utils.i`), typically each including `utils.h`
  - Compiler proper generates **assembly modules** (`myprogram.s`, `utils.s`)
  - Assembler generates **object code** (`myprogram.o`, `utils.o`)

- You **run** it: `/myprogram`
  - **Preprocessor** generates **preprocessed source** (`myprogram.i`, `utils.i`), typically each including `utils.h`
  - Compiler proper generates **assembly modules** (`myprogram.s`, `utils.s`)
  - Assembler generates **object code** (`myprogram.o`, `utils.o`)

**Linked Lists — Header File**

```c
/* CharList.h --- Linked lists containing character data */

typedef struct CharListNodeStruct {
    char data;
    struct CharListNodeStruct *nextPtr; /* struct label necessary! */
} CharListNode;

typedef CharListNode *CharList;

/* Access functions */
void printCharList(CharList p); /* printing as raw character sequence */
void insert(CharList *p, char val); /* insertion: recursive implementation */
void insertIter(CharList *p, char val); /* iterative implementation */
void delete(CharList *p, char val); /* deletion: recursive implementation */
void deleteIter(CharList *p, char val); /* iterative implementation */
```
Linked Lists — main()

```c
#include <stdio.h> /* CharListTest.c */
#include <stdlib.h>
#include <stdbool.h>
#include "CharList.h"

int main() {
    CharList list = NULL;
    char c;
    bool inserting = 1;
    while((c = getchar()) != EOF) {
        if (c == '
') continue;
        if (c == '	') {
            inserting = !inserting; continue;
        }
        if (inserting) {
            printf("Inserting \"%c\"\n", c);
            fflush(stdout);
            insert(&list, c);
        } else {
            printf("Deleting \"%c\"\n", c);
            delete(&list, c);
            printf("Current list: ");
            printCharList(list);
            printf("\n");
        } } return 0;
}
```

Queues — Implementation File

```c
#include <stdio.h> /* queue.c */
#include <stdlib.h>
#include <stdbool.h>
#include "queue.h"

void printQueue( Queue * q ) {
    QueueNode * n = q->head;
    while( n != NULL ) {
        printf("%c", n->data);
        n = n->nextPtr;
    }
}
```

Queues — Initialisation and Construction

```c
void initQueue(Queue * q) { /* q is assumed to be non-NULL */
    q->head = NULL;
    q->tail = NULL;
} 
```

This can be used to initialise local Queue variables.

Dynamically allocated Queues are initialised by the constructor:

```c
Queue * newQueue() { 
    Queue * q = malloc(sizeof(Queue));
    if ( q == NULL ) { fprintf(stderr, "newQueue: out of memory\n");
        return q;
    }
    initQueue(q);
    return q;
}
```
Queues — Adding an Item at the End

/* pass by reference — \texttt{q} is assumed to be non-NULL! */
bool enqueue( Queue * q, char c ) {
    QueueNode * n = malloc( sizeof(QueueNode) );
    if ( n == NULL ) {
        fprintf(stderr, "enqueue: out of memory!\n");
        return false;
    } else {
        n->data = c;
        n->nextPtr = NULL;
        if ( q->head == NULL ) { q->head = n; }
        else { q->tail->nextPtr = n; }
        q->tail = n;
        return true;
    }
}

Queues — Removing an Item from the Head

bool isEmpty( Queue * q ) { return q->head == NULL; }

/* pass by reference — \texttt{q} is assumed to be non-NULL and non-empty! */
char dequeue( Queue * q ) {
    QueueNode * oldHead = q->head;
    char c = oldHead->data;
    q->head = oldHead->nextPtr;
    if ( q->head == NULL ) {
        q->tail = NULL;
    }
    free(oldHead);
    return c;
}

List Operations, Complexity, and List Datatypes

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Singly-linked list with tail pointer</th>
<th>Doubly-linked list with tail pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert at head</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete at head</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insert at tail</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Concatenation</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete at tail</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Search</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Ordered insertion</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Typical datatype</td>
<td>Stack</td>
<td>Queue</td>
</tr>
</tbody>
</table>

Trees

- In the context of \textit{undirected graphs}, a \textit{tree} is a graph where for \textit{any two nodes} there is \textbf{exactly one path} connecting them.
- In the context of \textit{directed graphs}, a \textit{tree} is a connected graph where every node has indegree at most one.
- As \textit{inductively defined data structure}, a \textit{tree} is
  - either the \textbf{empty tree}
  - or a \textit{node} containing some data and having some number of \textbf{successor trees}

An important example are \textbf{terms}: A term is
  - either a variable $v$
  - or a constant $c$
  - or the application $f(t_1, \ldots, t_k)$ of a $k$-ary function symbol to $k$ terms
  \Rightarrow structural representations of programs, formulae, sentences, …
- Important as implementation of sets and partial functions: \textbf{search trees}
**Binary Trees: Type Definition**

A **binary tree** with *int* data is
- either the empty tree (NULL)
- or (pointer to) a node with an *int* data field and two successor trees.

```c
typedef struct TreeNodeStruct {
    struct TreeNodeStruct *leftPtr;  // pointer to left subtree
    int data;                      // node value
    struct TreeNodeStruct *rightPtr; // pointer to right subtree
} TreeNode;
```

typedef TreeNode * Tree;

**Binary Trees: Construction**

```c
Tree MkBranch(Tree left, int value, Tree right) {
    TreeNode * result = malloc( sizeof( TreeNode ) );
    if ( result == NULL )
        fprintf(stderr, "MkBranch(%d): no memory available.\n", value);
    else {
        result→data = value;
        result→leftPtr = left;
        result→rightPtr = right;
    }
    return result;
}
```

```c
int main() {
    Tree t1 = MkBranch( MkBranch( MkBranch( NULL, 7, NULL), 3, NULL)
                        , 6
                        , MkBranch( MkBranch( NULL, 4, NULL), 5
                                     , MkBranch( NULL, 2, NULL))
    );
    ...
}
```

**Binary Trees: Ordered Insertion**

In a **binary search tree** without duplicates, for every node *n* with data *d*,
- the data of all nodes in the *left* subtree of *n* are **less than** *d*;
- the data of all nodes in the *right* subtree of *n* are **greater than** *d*.

```c
void treeInsert( Tree * t, int value )
{
    if ( *t == NULL ) { *t = MkBranch( NULL, value, NULL );}
    else {
        if ( value < (*t)→data )
            treeInsert( &( (*t)→leftPtr ), value );
        else if ( value > (*t)→data )
            treeInsert( &( (*t)→rightPtr ), value );
        else {} /* value == (*t)→data — ignore duplicate values */
    }
}
```
Binary Trees: Ordered Insertion, Iterative

```c
void treeInsertIter( Tree * t, int value )
{
    while ( *t != NULL && value != (*t)->data ) {
        if ( value < (*t)->data ) {
            t = &( (*t)->leftPtr );
        } else /* value > (*t)->data */ {
            t = &( (*t)->rightPtr );
        }
    }
    if ( *t == NULL ) { *t = MkBranch( NULL, value, NULL ); } else {} /* value == (*t)->data — ignore duplicate values */
}
```

Traversals

```c
void inOrder( Tree t ) {
    if ( t != NULL ) { inOrder( t->leftPtr );
        printf("%3d", t->data );
        inOrder( t->rightPtr );
    }
}

void preOrder( Tree t ) {
    if ( t != NULL ) { printf("%3d", t->data );
        preOrder( t->leftPtr );
        preOrder( t->rightPtr );
    }
}

void postOrder( Tree t ) {
    if ( t != NULL ) { postOrder( t->leftPtr );
        postOrder( t->rightPtr );
        printf("%3d", t->data );
    }
}
```

Tree Size

Simple recursive function:

```c
int treeSize( Tree t ) {
    if ( t == NULL ) return 0;
    else return treeSize( t->left ) + 1 + treeSize ( t->right );
}
```

More complicated approach: updating a counter as side-effect:

```c
void treeAddSizeToCounter( Tree t, int * count ) {
    if ( t == NULL ) return;
    else { (*count)++;
    treeAddSizeToCounter( t->left, count );
    treeAddSizeToCounter( t->right, count );
    }
}

int treeSize2( Tree t ) {
    int count = 0;
    treeAddSizeToCounter( t, &count );
    return count;
}
```

Tree Membership

Recursive version:

```c
bool treeMember( Tree t, int q ) {
    if ( t == NULL ) return false;
    if ( q < t->data ) return treeMember ( t->left, q);
    if ( q > t->data ) return treeMember ( t->right, q);
    return true;  // because here q == t->data
}
```

Transforming tail recursion into iteration:

```c
bool treeMemberIter( Tree t, int q ) {
    while ( t != NULL && q != t->data ) {
        if ( q < t->data ) t = t->left;
        else t = t->right;
    }
    return t != NULL;  // because then q == t->data
}
```
Preparing Deletion: Cutting Off the Largest Node

Specification:
• The result is a one-node tree consisting of the node containing the largest value in \( t \) before the call
• \( t \) loses only that node, and is still an ordered binary tree after the call

```c
Tree cutOffLargestNode(Tree * t) {
    Tree result;
    if ( *t == NULL ) return NULL; /* refine specification! */
    else if ( (t)->rightPtr == NULL ) /* not yet found */
        return cutOffLargestNode( &( ( *t )->rightPtr ) );
    else if ( (t)->rightPtr == NULL ) — found largest node */
        result = *t; /* remembered largest node */
        *t = result->leftPtr; /* redirected in-edge */
        result->leftPtr = NULL; /* deleted out-edge */
        return result;
}
```

Ordered Binary Trees: Deleting

```c
void treeDelete( Tree * t, int value ) {
    Tree delNode;
    while ( *t != NULL && value != ( *t )->data )
        if ( value < ( *t )->data ) t = &( ( *t )->leftPtr );
        else t = &( ( *t )->rightPtr );
    if ( *t == NULL ) return;
    else /* value == ( *t )->data */
        delNode = *t;
        if ( ( *t )->leftPtr == NULL ) *t = ( *t )->rightPtr;
        else if ( ( *t )->rightPtr == NULL ) *t = ( *t )->leftPtr;
        else /* two successors */
            delNode = cutOffLargestNode( &( ( *t )->leftPtr ) );
        ( *t )->data = delNode->data;
    free( delNode );
}
```

Stacks

• Also called LIFO stacks — Last In, First Out
• Model-theoretically equivalent to lists
• Usual implementation: (singly) linked lists
• Different Interface: Insertion, inspection, and deletion all happen only at one end — the “top” of the stack
  - Test for empty stacks:
    ```
    bool isEmpty( Stack s );
    ```
  - Pushing an item on the top of the stack — void does not report out-of-memory errors!
    ```
    void push( Stack * s, int n );
    ```
  - “Popping” the top element from a non-empty stack — precondition!
    ```
    int pop( Stack * s );
    ```

Stacks — Implementation

```c
typedef struct StackNodeStruct {
    int data;
    struct StackNodeStruct * nextPtr; /* struct label necessary! */
} StackNode;
```

typedef StackNode * Stack;

Stack Interface: Insertion, inspection, and deletion all happen at the “top”:
  - Test for empty stacks:
    ```
    bool isEmpty( Stack s );
    ```
  - Pushing an item on the top of the stack — false reports out-of-memory errors.
    ```
    void push( Stack * s, int n );
    ```
  - “Inspecting” the top element from a non-empty stack — precondition!
    ```
    int top( Stack s );
    ```
  - “Popping” (side-effect!) the top element from a non-empty stack
    ```
    int pop( Stack * s );
    ```
Stacks — Exercise

typedef struct StackNodeStruct {  /* stack.h */
    int data;
    struct StackNodeStruct * nextPtr;  /* struct label necessary! */
} StackNode;

typedef StackNode * Stack;

bool isEmpty( Stack s );
int top( Stack s );
bool push( Stack * s, int n );
int pop( Stack * s );

• What is missing from the interface?

• Provide an implementation file.

• Design and implement a “reverse Polish notation” calculator using a separate stack module.