Lecture 8.2

Topics:

1. Stack – Introduction
2. Stack Implementation – Array Based

---

1. Stack – Introduction

One of the popular data structures is stack. Stack is a natural description for many applications. Let's see what a stack is!

1.1 Definition

A stack is a collection of objects, where objects are of the same type (and there may be a known maximum number of objects in a stack). A stack is also referred to as a **Last In First Out (LIFO)** structure and generally depicted as in **Figure 1**.

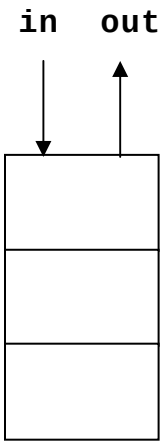

Figure 1 A stack representation

1.2 Memory Stack – An Application

Stacks are used very commonly in computer software and applications. For example, space for parameters and local variables is created internally within the computer by using stack.

For each function call, a memory stack is incremented to create space for local variables and parameters associated with the method (This space is known as the "activation record" for the function call).

1.3 Implementation

There are two basic operations that one can do with a stack:

(1) Pushing, and

(2) Popping.

These two operations must apply to the only one location, which is the top of the stack. Clearly, one cannot push into a stack if it is full and one should not pop an empty stack.

There are two approaches for implementations of stacks:

(1) List, and

(2) Array (or vector) based.

One fundamental difference between these two approaches is about the use of indices in array-based implementation to indicate roughly how many members (memory blocks) would be initially and no such a priori information is needed in list approach.

1.3.1 List Implementation

Let's assume that the data to be stored in a list is of type `DataType`.

The two functions of `insert()` and `remove()` are now restricted to deal with the first element of the list. That means `insert()` will add (`push` in) one node at the top of the list and `remove()` will remove (`pop` out) the first node of the list.

There are several helper methods that most stacks would have to check out the behavior of the stack. For example, a general stack ADT with a linked list based is given below.

```
DataList dataList;          /*A list of values (nodes)
                              of type DataType*/

void push(DataList, DataType node);  /*DataType is the
                                       element type*/

void pop(DataList);         /*May want to have the value
                             returned instead of void*/

void empty(DataList);       /*Remove all values*/

int isEmpty(DataList);      /*Is the stack empty?*/

DataType top(DataList);     /*Look at value at the top*/
```

1.3.2 Array-Based Implementation

The array implementation uses a **`struct`** to hold the stack information and location of the actual data array.

Consider a structure below.

```
struct StackHead {
  int stackSize;
  int topOfStack; /*Empty stack will have topOfStack == -1*/
  DataType* stackArray;
};

typedef struct StackHead* Stack;

void push(Stack stk, DataType data);
void pop(Stack stk);

void empty(Stack stk);
int isEmpty(Stack stk);
DataType top(Stack stk);
int isFull(Stack stk);
```

The implementation for the above methods should be similar to those in the list based structure. One can add a new method `isFull()` to check out if the stack is full. To add (push) to a stack, one may

want to call in `isFull()` if it is full then additional memory must be allocated before the next push can take place.

2. Stack Implementation – Array Based

2.1 Example

An array-based implementation is given below where the data are of type **`int`**.

```
/**
 * Program:    cis27L0821.c
 * Discussion: Stack -- Array Implementation
 */
#include <stdio.h>
#include <stdlib.h>

const int EMPTYSTACK = -1;
const int MAXSTACK = 5;

struct StackHead {
  int stackSize;
  int topOfStack;
  int *stackArray;
};

typedef struct StackHead* Stack;

void push(Stack stk, int a);
void pop(Stack stk);
int top(Stack stk);

void empty(Stack stk);
int isEmpty(Stack stk);
int isFull(Stack stk);

int main() {
  int ch;
  Stack myStack = (Stack) malloc(sizeof(Stack));

  myStack->stackArray = (int*) malloc(sizeof(int) * MAXSTACK);

  printf("\nsizeof( int ) * MAXSTACK : %d\n", sizeof(int) * MAXSTACK);

  myStack->topOfStack = EMPTYSTACK;

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  push(myStack, 5);

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  push(myStack, 6);

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  pop(myStack);
```

```
  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  pop(myStack);

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  push(myStack, 1);

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  push(myStack, 2);

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  push(myStack, 3);

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  push(myStack, 4);

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  push(myStack, 5);

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  push(myStack, 6);

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  push(myStack, 7);

  printf("\ntopOfStack : %d\n", myStack->topOfStack);
  printf("\n\tValue of top of stack : %d\n", top(myStack));

  printf("\nEnter a character + ENTER to QUIT.\n");
  scanf("&d", &ch);

  return 0;
}

void push(Stack stk, int a) {
  if (isFull(stk)) {
    printf("\nStack is full!\n");
  } else {
    stk->topOfStack++;
    stk->stackArray[stk->topOfStack] = a;
  }
  return;
}
```

```
void pop(Stack stk) {
  if (isEmpty(stk)) {
    printf("\nStack is empty!\n");
  } else {
    stk->topOfStack--;
  }
  return;
}

int top(Stack stk) {
  int temp;
  if (!isEmpty(stk)) {
    temp = stk->stackArray[stk->topOfStack];
  } else {
    printf("\nStack is empty!\n");
    temp = -9999;
  }
  return temp;
}

void empty(Stack stk) {
  stk->topOfStack = EMPTYSTACK;
  return;
}

int isEmpty(Stack stk) {
  return (stk->topOfStack == EMPTYSTACK);
}

int isFull(Stack stk) {
  return (stk->topOfStack == (MAXSTACK - 1));
}

/*PROGRAM OUTPUT

sizeof( int ) * MAXSTACK : 20

topOfStack : -1

Stack is empty!

        Value of top of stack : -9999

topOfStack : 0

        Value of top of stack : 5

topOfStack : 1

        Value of top of stack : 6

topOfStack : 0

        Value of top of stack : 5

topOfStack : -1

Stack is empty!
```

```
        Value of top of stack : -9999
topOfStack : 0
        Value of top of stack : 1
topOfStack : 1
        Value of top of stack : 2
topOfStack : 2
        Value of top of stack : 3
topOfStack : 3
        Value of top of stack : 4
topOfStack : 4
        Value of top of stack : 5
Stack is full!
topOfStack : 4
        Value of top of stack : 5
Stack is full!
topOfStack : 4
        Value of top of stack : 5
Enter a character + ENTER to QUIT.
s
*/
```

2.2 Implementation Issues

In the above implementation, the stack size is fixed by declaring a value of **MAXSTACK**. What if we would like to create a stack of some other size?

What do we need to do if stack is full? Increasing its size? If so, how?