The ADT stacks

The term abstract data type (ADT) is used in computer science to mean a data structure together with its operations, without specifying an implementation. Suppose we want to define a new data type for complex numbers of the form $a + ib$, where a and b are real. The new data type, together with its arithmetic operations, is an ADT. It is up to each system to determine how the values of the new data type are represented and manipulated computationally. Native types such as `char`, `int`, and `double` are implemented by the C compiler.

Here, we develop and implement the ADT stack, one of the most useful standard data structures. A stack is a data structure that allows insertion and deletion of data to occur only at a single restricted element, the top of the stack. This is the last-in-first-out (LIFO) discipline. This means the most recently added element is the first to be removed, like a pile of plates, where you always pick the top one. The typical operations that can be used with a stack are

- push: adds an element to the top of the stack.
- pop: removes and returns the top element.
- top: returns the top element without removing it.
- empty: checks if the stack has no elements.
- full: checks if the stack has reached its maximum capacity (only for fixed-size implementations).
- reset: clears the stack, or initializes it.

The stack, along with these operations, is a typical ADT. A stack can be represented by a fixed-length array or a linked list.

Stack using an array: First, we describe the implementation of a stack using a fixed-length array. The top of the stack will be denoted by an integer-valued member named top. The various stack operations will be implemented as functions, each of whose parameter lists includes a parameter of type pointer to stack. Using a pointer avoids copying a potentially large stack to perform a simple operation.

To make the program more general, we denote each element of the stack by `data` with the help of `typedef`. Suppose that the members of the stack are integers. In this case, we declare

```
typedef int data;
```

On the other hand, if the members of the stack are plates identified by brand name and manufacturing year, then we declare

```
typedef struct {
char brand[50];
int mfgy;
} data;
```

Suppose that we are given a character string. We can write the string in reverse order quite easily. However, suppose the task is to do the same using a stack. Let `MAXSIZE` be the dimension of the array. We first reset the stack. Since the integer variable component `top` points to the top element and an array starts with index zero, reset is equivalent to setting the `top` component to `-1`. If the stack is not full (i.e. `top` is not `MAXSIZE-1`), then increment the `top` by one and push the first character of the string onto the stack in the `top` position. Then do the same with the second character and so on until the character just before the string-terminating null character. Then, we pop and print the characters one by one from the stack until the stack becomes empty. Note that a stack is empty when `top` component becomes `-1`. The code is self-explanatory.

```
#include <stdio.h>
#define MAXSIZE 1000
#define EMPTY -1
typedef char data;
typedef struct
{
 data item[MAXSIZE];
 int top;
} stack;
typedef enum {false,true} boolean;
boolean empty(stack *);
boolean full(stack *);
void reset(stack *);
void push(stack *, data);
data pop(stack *);
int main()
{
int i;
char prov[]="Life is not a bed of roses.";
stack s;
reset(&s);           // reset stack, i.e. top=-1
printf("Original string: %s\n",prov);
printf("Reverse string using stack: ");
```

```
for(i=0;prov[i] != '\0';i++)  //loop over string
{
 if(!full(&s))        // equiv. if(full(&s)==false)
 {
  push(&s,prov[i]);
 }
}
while (!empty(&s))        // equiv while(empty(&s)==false)
{
 putchar(pop(&s));  // take out (char) data and print using putchar()
}
putchar('\n');
return 0;
}
data pop(stack *stk)
{
 data c;
 c=stk->item[stk->top];
 stk->top -=1;
 return c;
}
boolean empty(stack *stk)
{
 return (boolean) (stk->top==-1);
}
boolean full(stack *stk)
{
 return (boolean) (stk->top==MAXSIZE-1);
}
void push(stack *stk,data c)
{
 stk->top +=1;
 stk->item[stk->top]=c;
}
void reset(stack *stk)
{
 stk->top=-1;
}
```

Listing 76: C Program "arstk.c"

The following is the output of the code:

```
Original string: Life is not a bed of roses.
Reverse string using stack: .sesor fo deb a ton si efiL
```

We have not used the top function in the above program. However, we can easily write it as follows:

```
data top(stack *stk)
{
 return stk->item[stk->top];
}
```

Stack using a linear linked list: Here, the array elements have been replaced by nodes. Each node has two components: an item that holds the data, and a self-referential pointer. The stack is defined as a structure with a single component `top` that points to a node. Initially, the `top` pointer is set to `NULL`. We also check whether `top` equals `NULL` to determine whether the stack is empty or not. When we get new data, we create a space for a new node. The new data is stored in the item component of the new node, and the pointer of the new node points to the node pointed by the `top` pointer component of the stack. Next, the `top` pointer is made to point to the new node. Similarly, to pop an element from the stack, we point the `top` pointer to the second most top node. The data from the top node is returned, and the space for the top node is freed.

```
#include <stdio.h>
#include <stdlib.h>
typedef char data;
typedef struct node
{
 data item;
 struct node *next;      // self referential node
} node;
typedef struct
{
 node *top;              // stack consists of single pointer to node
} stack;
typedef enum {false,true} boolean;
boolean empty(stack *);
void reset(stack *);
```

```
void push(stack *, data);
data pop(stack *);
int main()
{
int i;
char prov[]="Life is not a bed of roses.";
stack s;
reset(&s);           // reset stack, i.e. top=NULL
printf("Original string: %s\n",prov);
printf("Reverse string using stack: ");
for(i=0;prov[i] != '\0';i++)  //loop over string
{
  push(&s,prov[i]);     // made a new node & assign
}
while (!empty(&s))        // equiv while(empty(&s)==false)
{
 putchar(pop(&s));  // take out (char) data and print using putchar()
}
putchar('\n');
return 0;
}
data pop(stack *stk)
{
 data c;
 node *p;
 p=stk->top;         // p points to the top node
 stk->top=p->next;   // top points to next node
 c=p->item;          // assign data to c
 free(p);     // free the space of the top node
 return c;
}
void push(stack *stk,data c)
{
 node *p;
 p=(node *)calloc(1,sizeof(node));    // create a new node
 p->item=c;                           // assign data to node
 p->next=stk->top;       // new node points to that pointed by stack->top
 stk->top=p;                          // stack->top points to new node
}
```

```
boolean empty(stack *stk)
{
        return (boolean) (stk->top==NULL);   // check if top == null
}

void reset(stack *stk)
{
 stk->top=NULL;                          // assign top to null
}
```

Listing 77: C Program "lnstk.c"

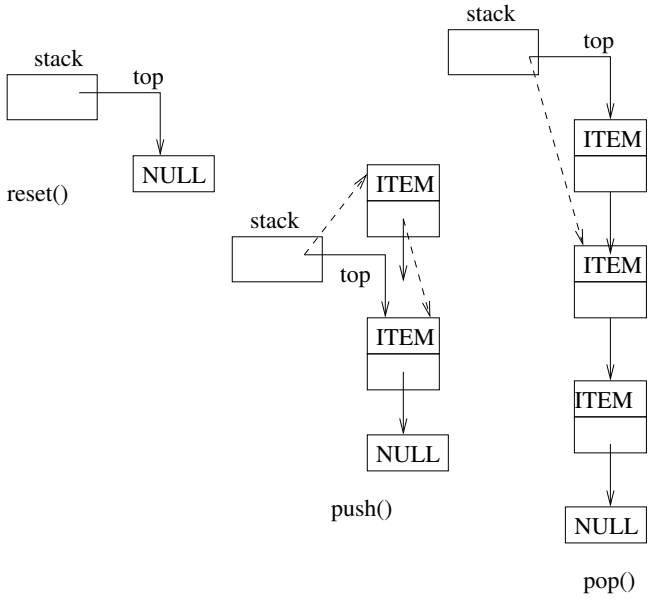
The following is the output of the code:

```
Original string: Life is not a bed of roses.
Reverse string using stack: .sesor fo deb a ton si efiL
```

Here, we can also write the top function as follows:

```
data top(stack *stk)
{
 return stk->top->item;
}
```

A pictorial diagram illustrating the main operations of a stack is shown below.



Note that in the linear linked-list implementation, the stack is defined to be a structure with a single component `top` that points to a node. We can also define a stack to be a structure with an additional integer counter `cnt` that counts the number of nodes in the stack:

```
typedef struct
{
 node *top;              // pointer to a node
 int cnt;                // node counter
} stack;
```

We just set `stk->cnt=0` in the reset function. Similarly, we check `stk->cnt==0` to test whether the stack is empty.

Return to Lecture Topics