STACKS ADDITIONAL READING

STACKS IN GENERAL

A ***stack*** is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the ***top*** of the **stack**. The definition of the **stack** provides for the insertion and deletion of items, so that a **stack** is a dynamic, constantly changing object. New items may be put on **top** of the **stack**, or items, which are at the top of the stack, may be removed. A **stack** is sometimes called a ***last-in first-out (LIFO) list***.

Primitive operations:

When an item is added to a **stack**, it is ***pushed*** onto the **stack**, and when an item is removed, it is ***popped*** from the stack. Given a **stack** ***s***, and an item ***i***, performing the operation ***push(s, i)*** adds the item ***i*** to the **top** of **stack** ***s***. Similarly, the operation ***pop(s)*** removes the **top** element and returns it as a function value. The following assignment operation removes the element at the **top** of ***s*** and assigns its value to ***i***;

i = ***pop***(s);

Because of the **push** operation, which adds elements to a **stack**, a **stack** is sometimes called ***pushdown list.*** There is no upper limit on the number of items that may be kept in a **stack**, since the definition does not specify how many items are allowed in the collection. **Pushing** another item onto a **stack** merely produces a larger collection of items. However, if a **stack** contains a single item and the **stack** is **popped**, the resulting **stack** contains no items and is called the **empty stack**. Although the **push** operation is applicable to any **stack**, the **pop** operation cannot be applied to the **empty stack** because such a **stack** has no elements to **pop**.

Therefore, before applying the **pop** operator to a **stack**, we must ensure that the **stack** is not **empty**. The operation ***empty(s)*** determines whether or not a **stack s** is **empty**. If the **stack** is **empty**, **empty(s)** returns the value ***TRUE***; otherwise it returns the value ***FALSE***. Another operation that can be performed on a **stack** is to determine what the **top** item is without removing it. This operation is written as ***stacktop(s)*** and returns the **top** element of **stack s**. The operation **stacktop(s*)*** is not really a new operation, since it can be decomposed through a **pop** and a **push**.

    i = ***stacktop***(s);

is equivalent to

    i = ***pop***(s);
    push (s, i);

Like the operation **pop**, **stacktop** is not defined for an **empty stack**. The result of an illegal attempt to **pop** or access an item from an **empty stack** is called ***underflow***. **Underflow** can be avoided by ensuring that **empty(s)** is **false** before attempting the operation **pop(s)** or **stacktop (s)**.

Stack example with scoping:

Scopes indicate beginning and ending of a section, the range of parameters. The types of **scopes** are indicated by ***parenthesis*** (**(** and **)**), ***brackets*** (**[** and **]**), and ***braces*** (**{** and **}**). A ***scope ender*** must be of the same type as its ***scope opener***. A **stack** may be used to keep track of the types of ***scopes*** encountered. Whenever a **scope opener** is encountered, it is pushed onto the **stack**. Whenever a **scope ender** is encountered, the **stack** is examined. If the **stack** is ***empty***, the scope ender does not have a matching opener and the string is therefore invalid. If, however, the **stack** is ***nonempty***, the **stack** is **popped** and checked whether the **popped** item corresponds to the **scope ender**. If a match occurs, we continue. If it does not, the string is invalid. When the end of the string is reached, the **stack** is **empty**; otherwise the string invalid.

THE STACK AS ABSTRACT DATA TYPE

eltype is used to denote the type of the **stack** element and parameterize the **stack** type with **eltype.**

/* value definition */
abstract typedef <<eltype>> STACK (eltype);

/* operator definition */
abstract *empty* (s)
STACK (eltype) s;
postcondition ***empty*** == (*len* (s) == 0);

abstract eltype ***pop*** (s)
STACK (eltype) s;
precondition ***empty*** (s) == FALSE;
postcondition ***pop*** == first (s');
s == *sub* (s', 1, *len*(s') – 1);

abstract *push* (s, elt);
STACK (eltype) s;
eltype elt;
postcondition s == <elt> + s';

In the abstract definition, the **underflow** condition is recognized. But an ***overflow*** is not a necessary condition. An **overflow** condition takes place with a limited availability of space to hold the **stack** elements. When items are pushed onto a **stack** and all the available space for the **stack** is being used by the elements in the **stack**. Where as the **underflow** is detected when a **pop** operation is performed on the **stack** where there are no elements.

If and when an **overflow** is detected in ***push***, execution halts immediately after an error message is printed. ***pop*** routine does not see the **overflow** condition. This routine allows the calling program to proceed after the call to ***pushandtest***, whether or not **overflow** was detected.

OTHER STACK FUNCTIONS

```
void popandtest (struct stack *ps, int *px, int *pund)
{
        if (empty (ps) )
        {
            *pund = TRUE;
            return;
        }  /* end of empty if */

        *pund = FALSE;
        *px = ps->items[ps->top--];
        return;

}  /* end of popandtest function */

popandtest (&s, &x, &und);
if (und)  /* take corrective action  */
else      /*  use value of x          */

void push (struct stack *ps, int x)
{
        if (ps->top == STACKSIZE-1)
        {
            printf ("%s",  "stack overflow");
            exit (1);
        }  /* end of if */
        else
             ps->items[++(ps->top) ] = x;
```

```
        return;
}  /* end of push function */

void pushandtest (struct stack *ps, int x, int *poverflow)
{
        if (ps->top == STACKSIZE-1)
        {
            *poverflow = TRUE;
            return;
        }  /* end of if */

        *poverflow = FALSE;
        ps->items[++(ps->top) ] = x;
        return;
}  /* end of pushandtest function */
```

If and when an **overflow** is detected in ***push***, execution halts immediately after an error message is printed. ***pop*** routine does not see the **overflow** condition. This routine allows the calling program to proceed after the call to ***pushandtest***, whether or not **overflow** was detected.

```
int stacktop (struct stack *ps)
{
        if (empty (ps))
        {
            printf (“%s”,  “stack underflow”);
            exit (1);
        }  /* end of empty if */
        else
            return (ps->items[ps->top] );
}  /* end of stacktop function */
```

Although the **overflow** and **underflow** conditions are treated similarly in *push* and *pop*, there is a fundamental difference between them. **Underflow** indicates that the pop operation cannot be performed on the **stack** and may indicate an error in the algorithm or the data. No other implementation or representation of the **stack** will cure the **underflow** condition. **Overflow**, however, is not a condition that is applicable to a **stack** as an abstract data structure. Abstractly, it is always possible to push an element onto a **stack**. A **stack** is just an ordered set, and there is no limit to the number of elements that such a set can contain. The possibility of **overflow** is introduced when a **stack** is implemented by an array with only a finite number of elements, there by prohibiting the growth of the **stack** beyond that number. The implementation of the algorithm did not anticipate that the **stack** would become so large. In some cases an **overflow** condition can be corrected by changing the value of the constant *STACKSIZE* so that the array field **item** contains more elements. There is no need to change the routines ***pop*** or ***push***, since they refer to whatever data structure was declared for the type **stack** in the program declarations.