GLOBAL JOURNAL OF PURE AND APPLIED SCIENCES, VOL.7, NO. 4, SEPTEMBER 2001: 755-759
COPYRIGHT (C) BACHUDO SCIENCE CO. LTD. PRINTED IN NIGERIA. ISSN 1118-0579

STACK : THE DATA STRUCTURE OF ALL PROGRAMMERS

CLEMENT B. NDEEKOR

(Received 3 August 1998 ; Revision accepted 2 March, 2001)

ABSTRACT

The skilled Programmer is at liberty to choose whatever data structure (that will contribute to an eventual program that is relatively efficient) for the storage of data items to be manipulated by his program. STACK is one of the data structures that can be chosen to accomplish the above task and aim. However, no matter the choice of data structure, a programmer directly or indirectly makes use of the stack. The aim of this paper is to point out how a programmer must make use of a stack before obtaining a result from his program. Illustrations are made of how stack can be explicitly defined and used as a data structure using two programming languages (PASCAL and Assembly Language on IBM microcomputers) and the implicit use of stack at the program translation and execution stage.

KEY WORDS: STACK, DATA, DATA STRUCTURE, MICROCOMPUTER, PROGRAM.

INTRODUCTION

The concept of data structures is a very important one in the aspect of computer programming. Its importance is so great that when combined with another concept (algorithms) computer scientists come to the conclusion that all about programming is covered. This can be attested to in the following definitions by renowned computer scientists:

(a) Computer Science is the study of data, its representation and transformation by a digital computer - Horowitz and Sahni, 1990.

(b) ALGORITHMS + DATA STRUCTURES = PROGRAMS – Niklause Wirth, 1976

A data structure is a particular organization of data items defining the relationship among them, the way they are viewed to be stored in computer memory and the types of operation that could be performed on them.

When writing a program, a programmer thinks of what tools to employ to obtain a program that will run fast, make maximum use of memory and other computer resources, give very accurate results etc. One of such tools are data structures. The data structures that can be adopted to store data items in a program are ARRAYS, QUEUES, LINKED LISTS, TREES, STACKS etc.

The stack may not have been explicitly chosen but the programmer must make use of it at least at the translation and/or execution stage of his program. How this is done is explained below. But first what is a stack?

DESCRIPTION OF A STACK

A stack is a data structure in which all insertions and deletions of entries are made at one end, called the TOP of the stack (Robert L. Kruse, 1994). The above definition implies that a stack has limited access. The figure below illustrates the structure of a stack.

CLEMENT B. NDEEKOR, Mathematics/ Computer Science Department, University of Port-Harcourt, Port-Harcourt. Nigeria

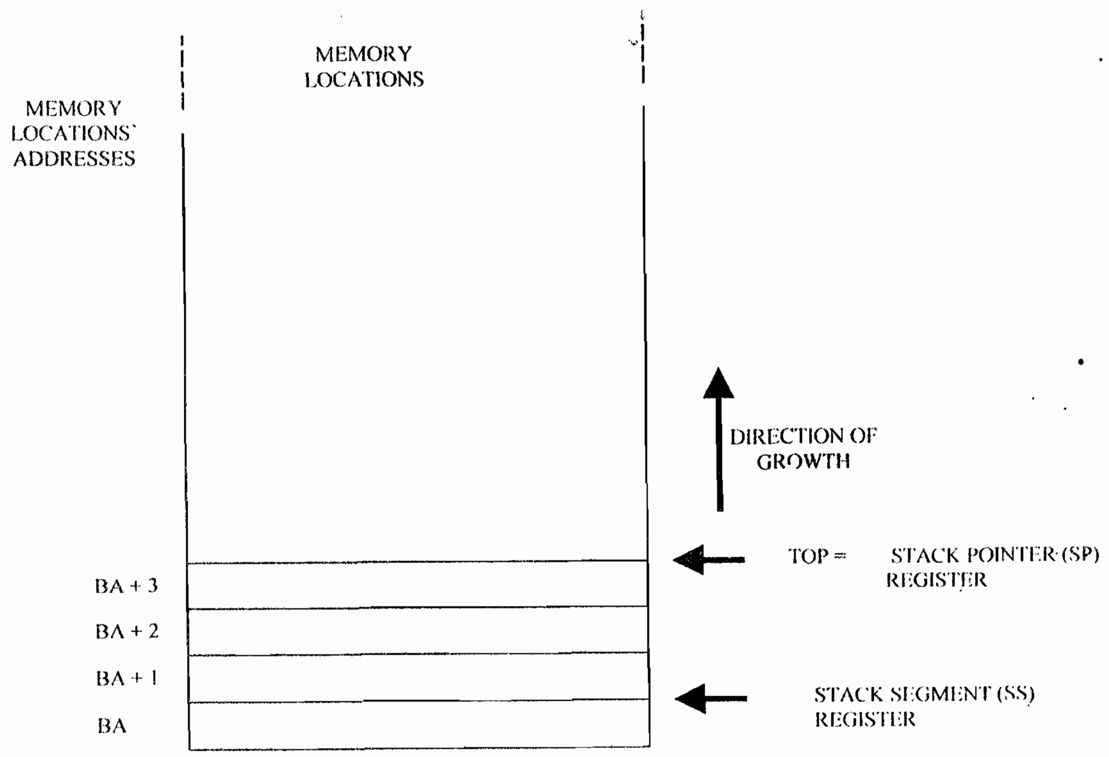
MEMORY LOCATIONS
MEMORY LOCATIONS' ADDRESSES
DIRECTION OF GROWTH
TOP = STACK POINTER (SP) REGISTER
BA + 3
BA + 2
BA + 1
BA
STACK SEGMENT (SS) REGISTER

FIG. 1 STRUCTURE OF A STACK

In the above diagram, BA represents the address of the first of the memory locations that form the stack.

Usually when a stack is defined, a group of consecutive memory locations are chosen to form the stack. This is done by some internal mechanisms and procedures. Two of the basic storage structures within the Central Processing Unit (CPU) called registers play paramount roles in stack definition and manipulation. They are the Stack Segment and Stack Pointer registers. The Stack Segment (SS) register points to the beginning address (in our case, BA) of the stack and never changes except a different stack from the current one is referenced. The Stack Pointer (SP) register points to the top of and infact the last element that was stored on the stack.

There are only two operations that are possible with the stack data structure. These are the PUSH and POP operations. With PUSH operation, a data item is stored on the stack and with the POP operation, a data item is retrieved from the stack. PUSH increments the value of SP by one while POP decrements its value by one. The last item stored on the stack is the first one that is always retrieved and hence, a stack is said to be a First In Last Out (FILO) or Last In First Out (LIFO) structure.

WAYS IN WHICH STACK IS USED

Programmers use the stack explicitly and implicitly. The skilled Programmer defines and uses a stack when it enhances the efficiency of his program. The novice Programmer who is not familiar with the stack as a data structure implicitly uses it when he eventually translates his program to object code and runs it. These two modes of stack usage are examined below.

EXPLICIT USE OF STACK

As stated earlier, a Programmer can define and use a stack as a data structure. How this is done is illustrated with two levels of programming languages viz. High level programming languages with

PASCAL as case study and Low level programming languages with Assembly language on the IBM Intel 8086 microcomputer as a case study.

In PASCAL, the Programmer must explicitly state that he wants to store and manipulate the data items involved in his program as a stack by declaring the data structure to be used as a stack. The declaration (for contiguous storage allocation) takes the form:

```
Type
        Stack   =   record
                    top : 0..maxstack;
                    entry: array [1..maxstack] of stackentry;
                    end;
```

In the above declaration, the underlined are keywords that must appear as they are in the declaration. A data structure of type stack has been declared with the name 'Stack'. Its size is going to be from 0 to the value of a system constant (maxstack). The entries of the stack are going to be in the form of an array that is to contain data items of the form stackentry (always determined by the Programmer). When the declaration has been done, suitable PUSH and POP procedures can then be used to manipulate the stack.

Still in PASCAL, a linked stack can also be defined. In this case, pointers are used to link one entry in the stack to the other. The declaration takes the form:

```
Type
            Stackpointer  =  ↑ stacknode;
            Stacknode     =    record

                          entry:   stackentry;

                          nextnode: stackpointer

                          end;
```

Again in the declaration, the underlined are keywords that must appear as they are. The pointer is called stackpointer and nextnode is like stackpointer and points from the previous entry to the next. The entries in the stack are called stacknode and are of the form stackentry (to be determined by the Programmer). Then appropriate procedures are used to manipulate the stack.

In Assembly Language on IBM Intel 8086 microcomputer and compatibles, every program is expected to declare at least one stack segment. The declaration takes the form:

```
Stack1          SEGMENT          stack
                DW       100H    DUP(?)

Stack1          ENDS
```

in which case a stack segment called stack1 has been declared to occupy 256 (100H) words of memory. Again, the underlined are keywords that must appear as they are.
After the declaration has been made, either the Programmer manipulates the stack as desired or the assembler uses it during assembly process.

IMPLICIT USE OF STACK

If a Programmer does not explicitly declare and manipulate a stack, he definitely uses one unconsciously if in his program he has used a sub-program (user-defined or/and in-built). These sub-programs are in the form of procedures, subroutines, functions or interrupts.

Use of stack in procedures, subroutines and functions are similar. During program translation and execution , main programs invoke subprograms. Taking procedures as a case study, the main program and each of the procedures it invokes have what is called an activation record which contains the following pieces of information:

(1) storage for simple names and pointers to arrays and other data structures local to the procedure,

(2) temporary variables for expression evaluation and parameter passing,

(3) information regarding attributes for local names and formal parameters when these cannot be determined at compile time,

(4) the return address,

(5) a pointer to the activation record of the caller.

These pieces of information must not be lost if the program is to execute correctly. So they are stored on a translator - created or run-time stack in the order in which the procedures are invoked.

As an illustration, if a procedure A invokes a procedure B which in turn invokes a procedures C; then their activation records will be stored on the stack in the order indicated in figure 2 below:

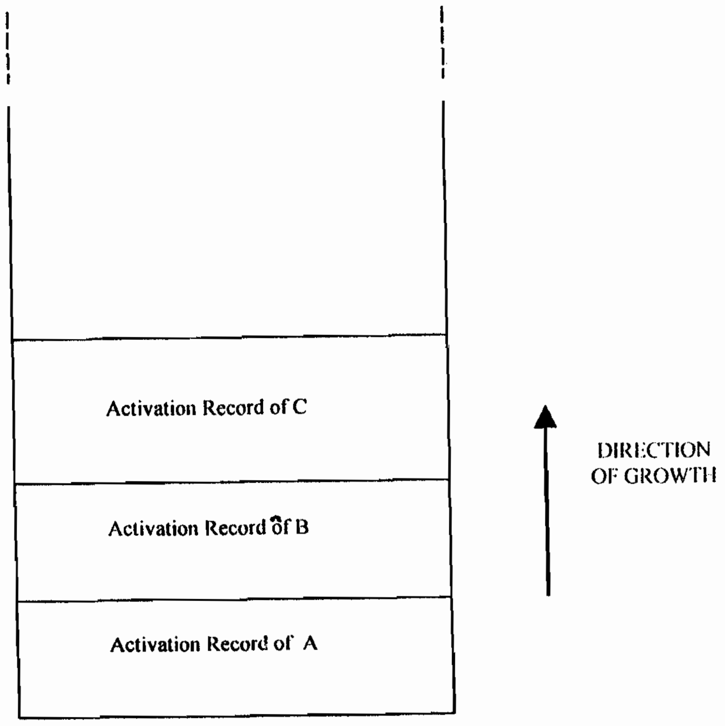

FIG. 2: STORAGE OF ACTIVATION RECORDS ON TRANSLATOR – CREATED OR RUN-TIME STACK.

Programming languages like C, C++ and assembly language allow Programmers to explicitly use an interrupt. For program execution to be transferred to the interrupt routines, information about the program invoking the interrupt at least at the point where the interrupt was invoked has to be stored on a translator-created stack so that execution can return at the appropriate point in the program when the interrupt has been serviced.

Also, programming languages like FORTRAN and BASIC which do not explicitly invoke interrupts cause so to be done when they are translated and executed. Some operations that are performed in high level languages and others invoke interrupts when they are translated and executed. Some of such operations are input/output of any type of data item. They atleast cause the service functions of interrupt 21H of DOS to be invoked and its servicing generates a translator-created stack.

CONCLUSION

Some programming languages allow Programmers declare and manipulate stack as a data structure within their programs. For users of such programming languages who are familiar with stack usage, they are at freedom to decide whether to use a stack or not.

However, at the translation and execution stage of programs and during interrupt processing, the programmer has no choice but to find out (if he cares) that the translator has employed the stack to be able to translate and execute his source code. Therefore a Programmer must always explicitly or/and implicitly make use of the stack data structure between when he writes his program and when he runs same to obtain desired result(s). Stack is thus every programmer's data structure.

REFERENCES

Detmer, R. C., 1990. Fundamentals of assembly language using IBM PCs and Compatibles. D.C. Heath and Company, Massachusetts, 530 pp.

Dunfemann, J., 1992. Assembly language (step-by-step). John Wiley & Sons Inc. USA, 432 pp.

Horowitz, E. and Sahni, S., 1990. Fundamentals of data structures in Pascal. Computer Science press, New York.

Kruse, R. C., 1994. Data structures and program design, 3rd Ed. Prentice-Hall, New Jersey, 689 pp.

Norton, P., 1985. Programmer's guide to the IBM PC. Microsoft Press, USA, 426 pp.

Sanchez, J., 1990. Assembly language tools and techniques for the IBM Microcomputers. Prentice-Hall, New Jersey, 447 pp.

Welsh, J. and Elder, J., 1982. Introduction to Pascal 2nd Ed. Prentice-Hall, India, 307 pp.

Wilson, L. B. and Clark R. G., 1988. Comparative programming language. Addison-Weley Publishing Company Inc., 379 pp.

Wirth N., 1976. Algorithms and data structures Prentice-Hall, New Jersey.