Data Structures in C (MCA371)

CAC1: Data Structures Applications and Algorithm Efficiency

1. INTRODUCTION

The organized arrangement of a collection of data in a computer's memory and the set of operations performed on that data can be defined as a data structure. It is basically an algorithm that follows certain rules and operates on abstract datatypes.

There are two main types of data structures, namely

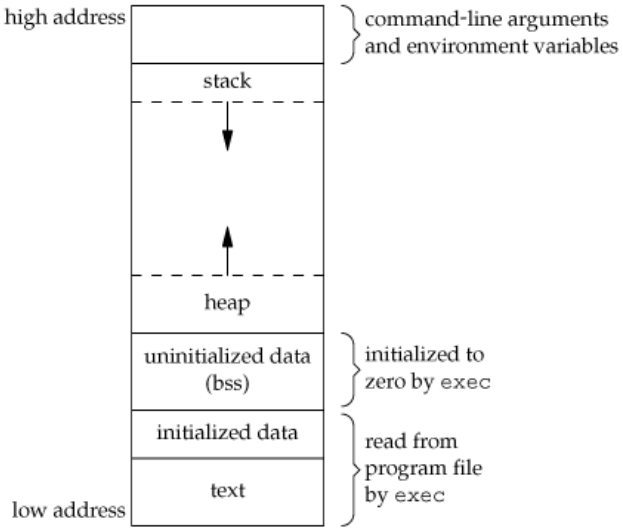
- Primitive data structures
  These are primitive datatypes such as int, float, char, etc.

- Non-primitive data structures
  These are derived from primitive data structures such as arrays, queues, linked lists, etc. Non-primitive data structures are also classified based on how data is stored in the memory, namely
  i) Linear data structure
     If the arrangement of data in memory is in a sequential manner.
  ii) Non-linear data structure
     If the arrangement of data in memory is not in a sequential manner.

Data structures increase modularity and code efficiency when used properly in the right applications. They make it easy to organize and operate on real-time data based on the application or the type of data they are operating on. They are also structured based on the frequency of the most common operations performed on the data.

2. APPLICATIONS OF DATA STRUCTURES

2.1 Stack in the memory management of a computer program:



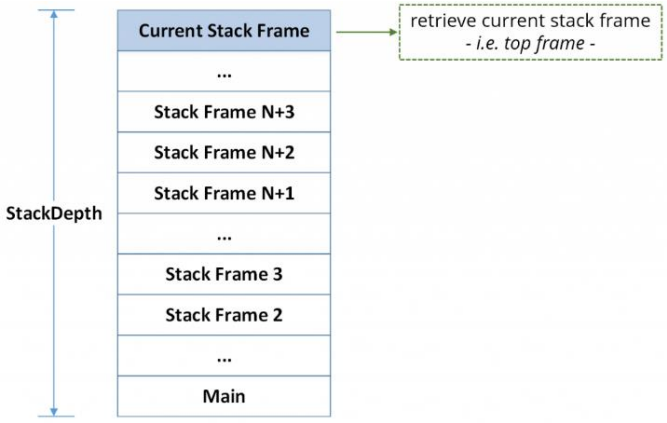
The stack data structure is used in the memory layout architecture of a computer program and holds stack frames for every function call in the program execution. Each stack frame holds the return address, local variables, and passed arguments for each function call. From the figure, we can see that the stack has dynamic size growing and shrinking in need of how many function calls are made. New stack frames are added to the top of the stack and when the function is destroyed the stack frame is removed from the top of the stack.

Instead of a stack, we cannot try to use linked lists or trees, or any other non-linear data structure as stack frames manage memory better by contiguous memory allocation and remove the need for compaction.

Stacks cannot be substituted with queues here as the most recent stack frame needs to be deleted first not the stack frame that was created first, in this case, the main function stack frame which would crash the whole execution.

Also holding the most recent stack frame at the stack top and implementing a cascading destroy operation holds a consistent flow and integrity of the program execution when a function call returns back to the called function, especially in cases of recursion.

By controlling the position of insertion and deletion of stack frames, stack data structures are the most suited data structure for this application.



2.2 Linked List in memory management of a computer (First Fit Algorithm):

First Fit Algorithm for Memory Management is when the first memory partition which is sufficient to accommodate the process is allocated. The idea is to use the memory block with a unique tag id. Each process of different sizes are given block id, which signifies to which memory block they belong to, and unique tag id to delete particular process to free up space.

Code in C:

```
#include<stdio.h>

void main()

{

int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

for(i = 0; i < 10; i++)

{flags[i] = 0;

allocation[i] = -1;}

printf("Enter no. of blocks: ");

scanf("%d", &bno);

printf("\nEnter size of each block: ");

for(i = 0; i < bno; i++)

scanf("%d", &bsize[i]);

printf("\nEnter no. of processes: ");

scanf("%d", &pno);

printf("\nEnter size of each process: ");

for(i = 0; i < pno; i++)

scanf("%d", &psize[i]);

for(i = 0; i < pno; i++)        //allocation as per first fit

for(j = 0; j < bno; j++)

if(flags[j] == 0 && bsize[j] >= psize[i])

{allocation[j] = i;

flags[j] = 1;

break;}

//display allocation details

printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");

for(i = 0; i < bno; i++)

{printf("\n%d\t\t%d\t\t", i+1, bsize[i]);

if(flags[i] == 1)

printf("%d\t\t\t%d",allocation[i]+1,psize[allocation[i]]);

else

printf("Not allocated");}}
```

Time Complexity:

Search: O(n) Insert: O(1) Deletion: O(1)

Space Complexity: dynamic in size

Why this data structure?

Since linked list does not allocate memory contiguously, it is perfect to use up empty spaces between processes in memory reducing internal fragmentation while still maintaining the flow.

2.3 String reversal using stack

Since a string is an array of characters, considering the array to be implemented as a stack we will be able to pop each character from the back (last in first out) in reverse order and push it into a new stack to get the reverse of the original string.

Code in C:

```
#include <stdio.h>
#include <string.h>
#define max 100
int top,stack[max];
void push(char x){
    // Push(Inserting Element in stack) operation
    if(top == max-1){
        printf("stack overflow");
    } else {
        stack[++top]=x;
    } }
void pop(){
    // Pop (Removing element from stack)
    printf("%c",stack[top--]);
}
main(){
  char str[]="sri lanka";
  int len = strlen(str);
  int i;
  for(i=0;i<len;i++)
      push(str[i]);
  for(i=0;i<len;i++)
    pop();
}
```

Time complexity: O(n)

Space complexity: O(n)

Why this data structure?

Since stack offers last in first out insertion and deletion strategy, it is easier to pop characters from the back onto a new stack.