COSC 3319 Lab #2 Spring 2017 Burris

**“C” Option (best grade is a 75):**

1. Implement algorithms Reallocate and MoveStack from the hymnal to repack memory when overflow occurs out of stack I.
2. *Allocate the memory dynamically in your program as a single contiguously allocated array in the system stack prior to processing data for the runtime stacks, for the stack space allocated at runtime in the system stack!* no heap memory for this storage space allocation!
3. Algorithms Reallocate and MoveStack should be written as functions or procedures.
4. **No global variables to communicate information btw your main program and subprograms!**
5. Limited use of global variables is occasionally justified in OOP “class” definitions. Can’t use *to communicate information within a class, you will not receive credit for the lab*!
6. prompt the user for the total amount of memory (M), L0, and number of stacks (N).
7. Use memory locations L0 + 1 through L0 + M
8. Your program should divide the memory equally between the stacks prior to processing any transactions.
9. When overflow occurs, assume that 13% of the available memory is to be divided equally between the stacks and that 87% of the available memory is to be divided based on growth.
10. No penalty is to be suffered by stacks that occupy less space when overflow occurs than when the previous overflow occurred (they receive the equal allocation).
11. **Print the contents of each transaction as you process it**.
12. After overflow, print contents of BASE[J], TOP[J], and OLDTOP[J] before repacking for 1 <= j <= N.
13. after memory has been repacked(***MoveStack***), print the contents of BASE[J] and TOP[J] ***properly labeled with addresses (subscript locations prior to executing algorithm Reallocate)***
14. Clearly indicate the address of the base and top of each stack in your output.
15. An error message should be printed and processing terminated when the available amount of memory free for distribution falls below 5% of the total memory allocation (from Lo + 1 thru M) or all transactions have been processed.
16. Print an appropriate message if an attempt is made to pop a stack that is empty but continue to process transactions.

**“C/B” data:**

Print values as they are popped from the stack.

Allocate 4 stacks at runtime in an array with subscripts from -11 through 51. Use memory locations L0 := 3 and M := 24 for the 4 stacks. Your stacks will physically occupy locations 4 through 24. Assume locations -11 through 3 and 25 through 51 are in use by other portions of your software. You must prompt the user at run time for the lower and upper bounds of the array (-11, 51), L0 and M.

I1 Burris, I2 Zhou, I2 Shashidhar, I3 Deering, I2 An, I2 Deering, I3 Lester, I1 Yang, I3 Smith, I2 Wei, I2 Zhou, I2 Arcos, D2, I1 Wei, I2 Rabieh, D1, D1, I2 Song, I2 Cho, D3, I2 Varol, I3 Karabiyik, I1 Cooper, I1 Smith, I1 McGuire , I3 Najar, I2 An, I1 Zhou, D2, I2 Deering, I1 Burris, I2 Cho, I2 McGuire, I3 Hope, I3 Pray, I3 NoHope

**“B” Option (maximum grade is 85):**

Implement the “C” option. Minimize the amount of main memory utilized as overhead for algorithms Reallocate and MoveStack as described in the handout

I will specifically verify that you use only a single physical array to hold the contents of OLDTOP, GROWTH, and NEWBASE, i.e., OldTop[J] = Growth[J-1] = NewBase[J] for 1 <= J <= (N+1)!

Clearly indicate in your code with a magic marker where the memory optimizations have been made. *Algorithms Reallocate and MoveStack must be implemented as a package (or class)*.

You may add additional functionality (methods) if desired.

Process the “C” option data.

**“A” Option (maximum grade is 94):**

Algorithms Reallocate and MoveStack must be implemented as generics

Implement the “B” option.

Process the C option data first. Now process the “A” option data below.

Use 3 stacks with L0 = 4 and M = 13 (use memory locations 5 through M = 15) in an array with locations 0 through 50. You are only using a portion of the space in the array.

I2 January 15 1956; I2 February 14, 1957; I3 September 16, 1946;

I2 September 17, 1842; I2 April 1, 2015; I1 December 24, 1996, D1, I3 March 16, 1992;

D1; I2 January 15, 1956; I3 April 4, 1492; I3 November 7, 1776;

I3 June 12, 1994; I2 July 4, 1776; I2 January 15, 2012; I3 December 6, 1991;

I3 March 5, 1886; I1 October 24, 1996; I1 November 23, 1996; I1 November 2, 1990;

I3 September 14, 1998

**“A+” Option (maximum grade is 100):**

Management would really be impressed if the user has the ability to specify the subscript (index) used to access the data space

If you implement this option, mark it with a colored pen and brag on yourself so that I do not overlook it while grading.

Allocate memory space with L0 = ‘a’, M = ‘n’, and terminate processing if the amount of available memory drops below 7% using the “C” option data set with 4 stacks. You are really using array locations b, c, d, e, f, g, h, I, j, k, l, m, and n.

In Ada Pos(‘M’) – Pos(‘A’) is the number of locations between A and M. Note the “succ(‘a’)” is ‘b’ and the “pred(‘b’)” is ‘a’.

Management would be further impresses if you use class definitions allowing for inheritance. This is not however required to receive the maximum possible grade.

*There is more pride in this part of the lab than points. How much pride do you have?*

**Hints:**

Hint 1: Many languages have a library floor and ceiling operator, normally buried in a math function library that must be referenced for the compiler (with, import, include). They may be under a related name such as round and truncate or lower bound and upper bound. A good programmer however should not need a library, indeed the challenge is to write your own. Consider the following:

function floor(x: float) return integer is

temp: integer;

begin

temp := integer(x); -- force conversion to integer

if( float(temp) <= x)

return temp;

else

return temp – 1;

end;

Hint 2: There is more than one way to read and write strings in Ada. In most instances if you declare a string variable to be of length 10 (e.g., str: array(1..10) of character), Ada expects 10 characters in the input every time you execute “get(str).” You can use a publicly available variable length string library such as VString to ease reading and writing variable length strings. Look on the “T” drive for VString. When there are only a limited number of possible strings, such as in the lab for names, you can declare them as a programmer defined enumeration type and read / write them using I/O routines written by the compiler using generics. For example:

The most appropriate data type for the “names” below is string. You will learn the most by using string variables. You may however make the names an enumeration type if desired and utilize Ada’s ability to create I/O routines for programmer defined enumeration data types.

Variable length string libraries in the public domain are available. You may obtain and use one of these libraries if desired as long as you include it as part of your source and properly document its use in the code. VStrings.ada at T:\CSC\DSB\ada\Ada\StringHandling is such a library.

type NameType is (Joe, Betty, Sam, Bob, Mary);

package NameType\_IO is new Ada.Text\_IO.Enumeration\_IO(NameType);

use NameType\_IO;

N1: NameType;

get(N1);

put(N1); -- “put” defaults to printing uppercase. A simple parameter can fix this.

-- In the generic instantiation for enumeration types:

-- put(To: out String; Item: out Enum; Set: in Type\_Set := Default\_Setting);

-- where Default\_Setting: Type\_Set := Upper Case;

Hint 3: Do not forget the hint in the first lab, i.e.,

Lab2 << FileIn >> Fileout

When this command is entered at the command line, all input is taken from FileIn and all output is redirected to FileOut. This technique can save you considerable effort during debugging.

**Multiple Sequentially Allocated Lists Sharing Memory Locations**

Assume the case where multiple sequentially allocated list must coexist in a restricted amount of memory. We assume that the list will vary in size dynamically. If one list overflows into space currently occupied by another list, we choose to rearrange the lists by moving their contents to accommodate the space requirements for the list that overflowed. For convenience, we will assume initially that all lists are stacks. We further assume for convenience that the stacks occupy memory locations 1 through M known as “StackSpace.” All stacks are initially allocated equal amounts of space. The sized of each stack is then allowed to vary dynamically at run time. If Base[J] and Top[J] are used to track the space allocations for each stack, then initial space allocation is given by:

**Base[J] = Top[J] = floor((J - 1) / N \* M) + L0**

where N is the number of stacks, 1 <= J <= N, and the stacks share the common memory area consisting of all locations L with L0 < L <= M. The “floor” operator means to truncate the fractional part of the number, e.g., floor(7.89699) is 7.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |
| L0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 = M |
|  | Joe | Mary | Bob |  |  |  |  | Tom | Sue |  |
| Base[1]=0 |  |  |  |  | Base[2]=5 |  | Base[3]=7 |  |  |  |
|  |  |  | Top[1]=3 |  | Top[2]=5 |  |  |  | Top[3]=9 | Base[4]=10 |
|  |  |  |  |  |  |  |  |  |  |  |

The diagram represents 3 stacks utilizing 10 memory locations 1 through 10. Note that a fourth “Base” has been allocated to allow for a consistent means to check for overflow in the insertion algorithm below. In general, for N stacks, we will use N + 1 bases. L0 is treated as a dummy location just prior to the useful memory area.

Comments are enclosed in { } in the following algorithms.

The basic stack operations are as follows:

**Insertion in stack K:**

Top[K] := Top[K] + 1;

If Top[K] > Base[K + 1] then

report overflow; *{Algorithm Reallocate should be used to determine if*

*additional space can be made available by taking it from another stack.}*

Else

StackSpace[ Top[K] ] := Y;  *{Insert value of Y into the stack.}*

End If;

**Deletion from stack K:**

If Top[K] = Base[K] then

report underflow;

Else

Y := StackSpace[ Top[K] ]; *{Remove top item in stack and assign to Y.}*

Top[K] := Top[K] - 1;

End If;

When overflow occurs in algorithm Insertion, one of three possibilities exist:

**A)** Find the smallest J for which K < J <= N and Top[J] < Base[J+1]. If any such J exist, move contents up one or more locations.

**B)** If step “A” fails, find the largest J for which 1 <= J < K and Top[J] < Base[J+1]. If this condition exists, move items down one or more notches.

**C)** We find Top[J] = Base[J+1] for all J /= K. We are out of memory and no accommodation is possible for the overflow condition.

**Algorithm Reallocate (sequentially allocated tables).**

We assume that overflow has occurred in one of the N stacks pointed to by Top[K] in the space allocation “StackSpace.” Top[J] is the current top for stack “J.” Base[J] represents the current base for each stack. This algorithm will reallocate available memory between stacks. EqualAllocate is the percentage of available memory that should be allocated equally between all stacks. GrowthAllocate is the percent of available memory that should be allocated to stacks based on their growth since the last time memory overflowed. For example, if EqualAllocate = 0.6, then 60% of free memory should be shared equally by all stacks on overflow and 40% should be used to reflect growth. Three temporary arrays OldTop, Growth, and NewBase are required**. They may actually occupy the same physical memory locations (a single physical array) if mapped in the following manner: OldTop[J] = Growth[J -1] = NewBase[J]** for 1 <= J <= (N+1). OldTop[J] should initially be set to the initial value of the top of stack pointer for each stack when the main program is initiated. We assume the stacks occupy all memory locations L where 1 <= L <= M. MinSpace is the minimum amount of available space left to make reallocation worth while. For example if we assume MinSpace = 1, then we use up the last available unit of memory prior to reporting failure. It would be more realistic to set MinSpace to say 10% of M. If available memory drops below a reasonable level, overflow and re-packing will occur repeatedly. The resulting overhead would be unacceptable in many applications. While every unit of memory can be utilized by this algorithm, such use is normally unreasonable.

**ReA1:** *{Find the amount of available space for reallocation. TotalInc is the total growth since the last time memory overflowed.}*

AvailSpace := M - L0 ( or Base[N+1]– Base[0] ); TotalInc := 0; J := N;

While J > 0 Loop

AvailSpace := AvailSpace - (Top[J] - Base[J]);

If Top[J] > OldTop[J] then

Growth[J] := Top[J] - OldTop[J];

TotalInc := TotalInc + Growth[J];

Else

Growth[J] := 0;

End If;

J := J - 1;

End Loop;

**ReA2**: If AvailSpace < (MinSpace - 1) then report insufficient memory for re-packing to occur and terminate. *{Please note that when memory overflowed, space for the new request has already been reserved at the location pointed to by Top[K]. If AvailSpace = 0, then there is exactly one unit of space available. If AvailSpace < 0, we are truly out of memory.}*

**ReA3:** GrowthAllocate := 1 - EqualAllocate; *{EqualAllocate must be represented as a decimal fraction, e.g., 0.15 would imply 15% of available memory to be allocated equally between all N stacks}*

Alpha := EqualAllocate \* AvailSpace / N; *{EqualAllocate \* AvailSpace is the amount of memory to be divided equally between the stacks. Each stack gets 1/N of this space. Alpha is a real number and must be computed to a reasonable number of digits to the right of the decimal point.}*

**ReA4:** Beta := GrowthAllocate \* AvailSpace / TotalInc; *{TotalGrowthSpace := GrowthAllocate \* AvailSpace is the amount of memory to be allocated based on growth. Beta := TotalGrowthSpace / TotalInc is the amount of space to allocate a stack for each unit it has increased in size since the last time memory overflowed. This algorithm does not penalize for a stack shrinking in size. There is simply no growth allocation, only the equal allocation represented by Alpha. Beta is a real number and must be computed to reasonable accuracy}*

**ReA5:** NewBase[1] := Base[1]; Sigma := 0;

For J := 2..N Loop *{Increase J from 2 through N in increments of 1.}*

Tau := Sigma + Alpha + Growth[J-1]\*Beta;

NewBase[J] := NewBase[J-1] + (Top[J-1] - Base[J-1]) + floor(Tau) - floor(Sigma);

Sigma := Tau;

End Loop;

*{NewBase[J] is the sum of the preceding base location (Base[j-1]) plus the size of the preceding stack (Top[J-1] - Base[J-1]) plus the growth allocation for the preceding stack (floor(Tau)) minus floor(Sigma)). The “floor” operator means to take the integer part of the number without rounding. For example: floor(0.345) = 0; floor(8.43) = 8; and floor(8.999999) = 8. When this step is complete, NewBase[J] represents the location of the new base for each stack J after re-packing.}*

**ReA6:** {It is time to re-pack the stacks. Recall that space for a new element was allocated in a stack pointed to by Top[K] when overflow occurred. We must adjust for this prior to re-packing by subtracting one from K. After re-packing is complete, we must add one back to K prior to returning to the program requesting the reallocation. The program that requested reallocation of memory may then store the desired datum in the stack location pointed to by Top[K].}

Top[K] := Top[K] - 1;

Perform Algorithm MoveStack;

Top[K] := Top[K] + 1;

Insert item causing the over flow at location Top[K];

For J in 1..N Loop //get ready for next potential overflow.

OldTop[J] := Top[J];

End Loop;

**Algorithm MoveStack:**

This algorithm moves the contents of each stack. Stacks are moved downward without overlapping stacks to be moved upward or stacks that are to remain in their current location. Note Base[1] for stack one should never move. It cannot move below location 1. If moved to location C, where C > 1, the locations 1 to (C-1) would be inaccessible (wasted).

**This is a classic space-time tradeoff.** The algorithm is well behaved when memory is only 50% full. That is, very little time is spent in re-packing. When possible, memory utilization should probably not exceed 75%. Note the largest stack if known should be the first stack as it is never moved. This will reduce overhead.

Space Optimization: The arrays OldTop, Growth, and Newbase can occupy the same physical space (OneArray): **OldTop[J] = Growth{J-1] = Newbase[J]** for 1 <= J <= N + 1.

Once values from OldTop and Growth are used to calculate a newbase the oldtop value is no longer required.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | OldTop |  |  | Growth |  |  | Newbase |  |  | OneArray |
| 1 |  |  |  |  |  | 1 |  |  | 1 |  |
| 2 |  |  | 1 |  |  | 2 |  |  | 2 |  |
| 3 |  |  | 2 |  |  | 3 |  | => | 3 |  |
| 4 |  |  | 3 |  |  | 4 |  |  | 4 |  |
| … |  |  | 4 |  |  | … |  |  | 5 |  |
| N |  |  | … |  |  | N |  |  |  |  |
|  |  |  | N |  |  |  |  |  | N+1 |  |
|  |  |  |  |  |  |  |  |  |  |  |

Assume the following StackSpace using locations 1 through 100, M= 100.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | A | B | C | D | E |  |  |  |  |  | A | B | C | D | E | F | A | B | … |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 100 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

In the beginning:

|  |  |  |  |
| --- | --- | --- | --- |
| Original | OldTop(1) = 0 | Base(1) = 0 | Top(1) = 0 |
| Values | OldTop(2) = 10 | Base(2) = 10 | Top(2) = 10 |
|  | OldTop(3) = 16 | Base(3) = 16 | Top(3) = 16 |
|  |  | Base(4) = 100 |  |

The stack space appears as above after processing the following operations. Note the operation **I2G** causes stack 2 to overflow into stack 3 which is currently empty. "I1A" should be interpreted as insert an "A" in stack 1. "D3" should be interpreted as delete the top item in stack 3.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| I1A | I1B | I1C | I1D | I1E | I3A | I3B | I2A | I2B | I2C |
| I2D | I2E | I2F | D3 | D3 | **I2G\*** |  |  |  |  |

At overflow:

|  |  |
| --- | --- |
| Base(1) = 0 | Top(1) = 0, 1, 2, 3, 4, 5 |
| Base(2) = 10 | Top(2) = 10, 11, 12, 13, 14, 15, 16, 17 |
| Base(3) = 16 | Top(3) = 16, 17, 18, 17, 16 |
| Base(4) = 100 |  |

**ReA1:**

L0 < L <= M => 0 < L <= 100

AvailSpace := M - L0 := M – 0 := 100.

TotalInc := 0

The number of stacks is N := 3.

Hence: AvailSpace := AvailSpace - (5-0) - (17 - 10) - (16 - 16) = 88.

Growth[1] := 5 - 0 := 5.

Growth[2] := 17 - 10 := 7.

Growth[3] := 16 - 16 := 0.

TotalInc := 5 + 7 + 0 := 12.

**ReA2:**

Note AvailSpace = 88 implies there are 88 spaces to allocate not including the space required to store "G" into stack 2. That space was reserved at the time overflow occurred.

**ReA3:**

Assume EqualAllocate := 0.1 or 10% equal space allocation. Then GrowthAllocate := 1.0 - 0.1 := 0.9 or 90% of space allocated based on dynamic growth.

**Alpha** := 0.1\*(AvailSpace/N) := 0.1 \* (88.0/3.0) := **2.9333**.

**ReA4:**

**Beta** := 0.9\*(AvailSpace/TotalInc) := 0.9 \* (88.0/12.0) := **6.5999**.

**ReA5:**

**{J := 1}**

Do nothing. The base of stack 1 never moves.

***NewBase[1] : = 0;***

Sigma := 0;

++++++++++++++++++++++++++++++++++++++++++++++++++

**{J := 2}**

Tau := Sigma + Alpha + Growth[1]\*Beta := 0 + 2.9333 + 5\*6.5999 := 35.933;

**NewBase[2]** := NewBase[1] + Top[1] - Base[1] + floor(Tau) - floor(Sigma)

:= 0 + 5 - 0 + 35 - 0 := **40**;

***Hence NewBase[2] := 40;***

Sigma := Tau := 35.9333

+++++++++++++++++++++++++++++++++++++++++++++++++++

**{J := 3}**

Tau := Sigma + Alpha + Growth[2]\*Beta := 35.9333 + 2.9333 + 7\*6.5999 := 85.0666;

**NewBase[3]** := NewBase[2] + Top[2] - Base[2] + floor(Tau) - floor(Sigma)

:= 40 + 17 - 10 + 85 - 35 := **97**;

***Hence NewBase[3] := 97;***

Sigma := Tau := 85.9333;

++++++++++++++++++++++++++++++++++++++++++++++++++++

We are finished with algorithm ReAllocate and ready to repack memory!