ESAT INNOVATION SCHOOL | ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

ALGORITHMS AND ARTIFICIAL INTELLIGENCE

ALGORITHMS AND
ARTIFICIAL INTELLIGENCE
HND
Iván Sancho
2020/2021
Borja Serrano García

Index

1.- ADT

1.1 Memory Node

The memory node is an ADT that serves as the basis for the others. The node contains data, a size of the node and a pointer to memory manager in order to make use of the functions.

Later a pointer is added to the previous node and the next one to be able to access them in the form of a list or stack.

In relation to data structures, a node usually means a single basic unit of data. Nodes can form more complex structures where any single node may be connected to any number of other nodes.

1.2 Vector

Vector is stored in memory contiguously. Vectors are data structures that grow and decrease dynamically, as needed.

The vector data structure represents a set of objects. The set of objects is variable in size.

Objects are incorporated up to the full capacity of the vector. When you need to incorporate a new object into a filled vector, the vector is automatically expanded, according to the capacity you want.
When the vector capacity is full, the vector resizes.

1.3 List

A list is a dynamic data structure that contains objects of the same type in such a way that an order is established between them.

Every item but the first has a predecessor, and every item but the last has a successor. The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list.

The simple list has a pointer that points to the next node until the end. The list can be readjusted if you have full capacity or if the length does not cover the capacity and you want to decrease.

The list ADT is an object-oriented extension of the concrete linked list data structure. It provides accessor and update functions based on an object-oriented encapsulation of the node objects used by linked lists.

1.4 DLlist

A doubly linked dllist is a data structure consisting of a set of sequentially linked nodes.

Each node contains three fields, two for so-called links, which are references to the next and previous node in the sequence of nodes, and another for storing the information.

The link to the previous node of the first node and the link to the next node of the last node, point to a type of node that marks the end of the dllist to facilitate the navigation of the dllist.

1.5 Stack

A stack is an ordered list or data structure that allows data to be stored and retrieved, being the mode of access to its elements of type LIFO (Last In, First Out).

For data handling it has two basic operations: push, which places an object on the stack, and its inverse operation, and pop, which removes the last stacked element.

You only have access to the last stacked object (TOS, Top of Stack). pop allows this element to be obtained, which is removed from the stack allowing access to the previous one (previously stacked), which becomes the last one, the new TOS. The head node and the data nodes are encapsulated in the ADT.

1.- Memory Stack

1.1 Storing data in a program

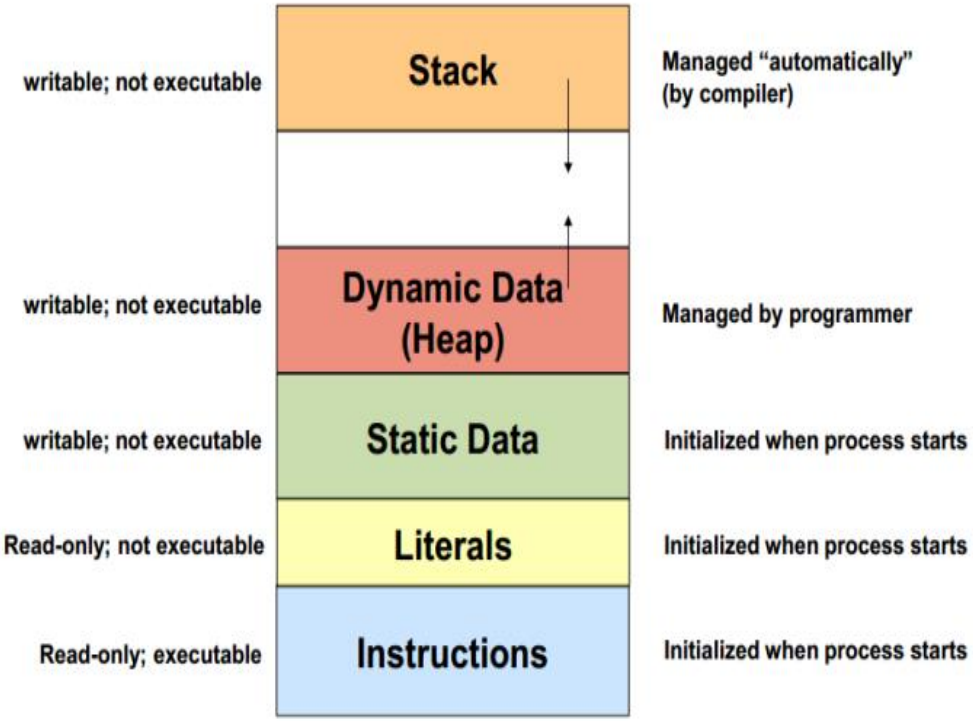
C program data is stored in one of three memory areas that the programmer has at his disposal: static memory, heap and stack:

• The static memory area is for data that does not change in size, it allows store global variables that are persistent during the execution of a Program.

• The heap allows you to store dynamically acquired variables (malloc or calloc) during the execution of a program.

• The stack allows to store arguments and local variables during the execution of the functions in which they are defined.

The compiler allocates a certain space for the variables and generates the references to access the variables in the stack and in the static zone. The size of the stack variables and the static zone cannot be changed during the execution of the program, it is assigned statically.



1.2 Stack operation

Here are stored the arguments passed to the program, the strings of the environment where it is executed, arguments passed to the functions, the local variables that do not yet contain any content, and it is also where the IP record is stored when a function is called (values return).

Before we dive into explaining the stack of a program, we are going to explain how the data is stored in this part of the segment.

A stack or stack is an ordered list or data structure in which the mode of access to its elements is of type LIFO (Last In First Out) that allows to store and retrieve data.

For data management there are two basic operations: stack (push), which places an object on the stack, and its inverse operation, remove or unstack (pop), which removes the last stacked item.

At any given time, you only have access to the top of the stack, that is, the last stacked object (TOS, Top Of Stack). The remove operation allows this element to be obtained, which is removed from the stack allowing access to the next one (previously stacked), which becomes the new TOS. There can be multiple execution stacks in a program when it is multithreaded; Each thread has its own stack with which to keep track of your function calls.

1.3 Stack Frame

Every time a program executes a function in memory, a data structure is generated in the STACK following the LIFO pattern. This section is where the data necessary for the correct execution of the functions of a program are stored. This structure that is built when a function is executed is called a stack frame. After the function has been executed, the created stack frame can be overwritten by the stack frames generated by other functions.

A stack frame is made up of:

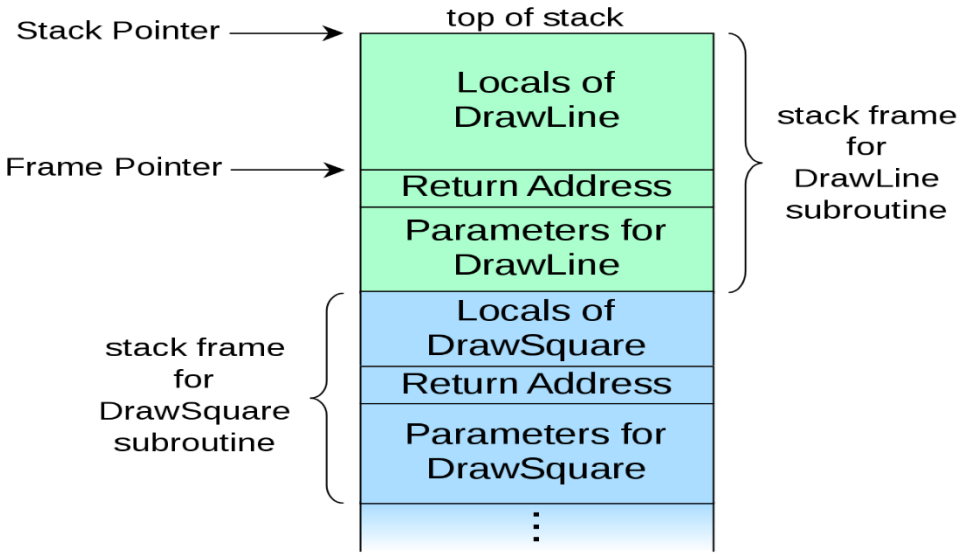
- Previous Stack Frame: What would be the other stack frames of other functions stored in the stack.

- Arguments: The arguments passed to the functions are stored here.

- Return address [EIP or RET]: The return memory address is stored here. The memory address that points to the next instruction through which the flow of the program should continue after the function has finished.

- Previous Stack Frame Address [EBP]: Memory address that points to the previous stack frame. The memory address previously pointed to by the EBP register is stored here. This will be better understood in later examples.

- Space reserved for local variables: Here the space is reserved to store the local variables of the function.



4.- Comparative

4.1 Vector

The vector is a contiguously linked data structure so I hope it's pretty fast.

Insert first has to be quite fast when the vector is empty, although it can be very slow if the vector has a lot of data since it has to resize all of them.

Insert last has to be faster since it goes directly to the end and does not have to resize the previous elements.

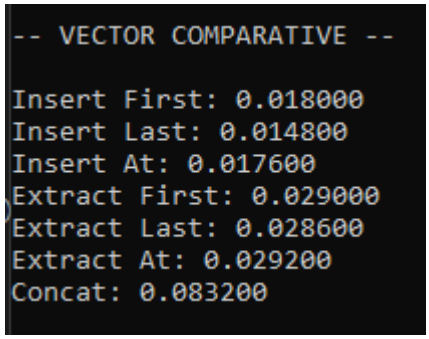
Insert At is going to be the slowest because it has to go through the vector until it reaches the position and then resize the objects that are behind.

Extract First will be the slowest because to extract the first object you have to resize all the others and delete both the information of the first and the last.

Extract Last is going to be the fastest because you don't have to resize anything other than change the queue and delete that object.

Extract At will be a bit slow depending on which position it is in, the closer to the queue the faster it will be.

Concat is going to be slow because it has to join two vectors that may not be complete and therefore has to erase the empty objects and then join the new vector into another vector.



The results in the vector have been as expected, although it is surprising how insert first requires so much time to execute. The concat method is also surprising because it duplicates almost any other method.

4.2 List

The list is a linked data structure but not contiguous so I hope it is a bit faster than the vector.

Insert first has to be quite fast when the list is empty and fast also if it is full because you only have to add a pointer to the beginning.

Insert last will be similar to insert first because you just have to add the object pointer to the end.

Insert At is going to be the slowest because it has to go through the list until it reaches the position and then have to change the pointer of the next node from the previous one to the new object and the pointer of the previous node with the new object.

Extract First is going to be very fast because you only have to remove the first object.
Extract Last is going to be the slowest because it has to go through the entire list to remove the last object.

Extract At is going to be slow because it has to go through the list until it reaches the required position and then have to change where they point to remove the object

Concat is going to be fast because it can access the last object of the first list and the first object of the second list to save it in a new list.

```
-- LIST COMPARATIVE --

Insert First: 0.017600
Insert Last: 0.017600
Insert At: 0.017600
Extract First: 0.016600
Extract Last: 0.068200
Extract At: 0.029400
Concat: 0.018400
```

The results in the list have been more or less as expected with very similar times between insert first and insert last and with a large time difference between insert at.

The extractions were as expected although the difference between extract first and the others is enormous.

4.3 Double List

The double list is a linked data structure but not contiguously but bilaterally so I hope it is a little faster than the vector and a little slower than the simple list.

Insert first has to be quite fast when the double list is empty and fast also if it is full because you have to add a pointer from the beginning to the previous first node and backwards.

Insert last will be similar to insert first because you just have to add the object pointer to the end and backwards for it to have double meaning.

Insert At is going to be the slowest because it has to go through the double list until reaching the position and then having to change from the previous and next nodes the one to be added and from the one to be added also put the double address, both for next and previous, of the object.

Extract First will be very fast because you only have to remove the first object and from the new first object add the last object as the previous one.

Extract Last is going to be the slowest because it has to go through the entire list to remove the last object.

Extract At is going to be very slow because it has to go through the list until it reaches the required position and then have to change where they point in both directions to remove the required object.

Concat will be almost as fast as in the list because it can access the last object of the first list and the first object of the second list to save it in a new list, you just have to add the double meaning between the first and the last of the double list with the first and the first and last of the list to be concatenated.

```
-- DLLIST COMPARATIVE --

Insert First: 0.048400
Insert Last: 0.018000
Insert At: 0.016800
Extract First: 0.017200
Extract Last: 0.223000
Extract At: 0.029400
Concat: 0.019200
```

The results of the double list are quite similar to those of the simple list. They improve in the inserts and in the extracts. This result was not what was expected since more operations have to be done on paper.

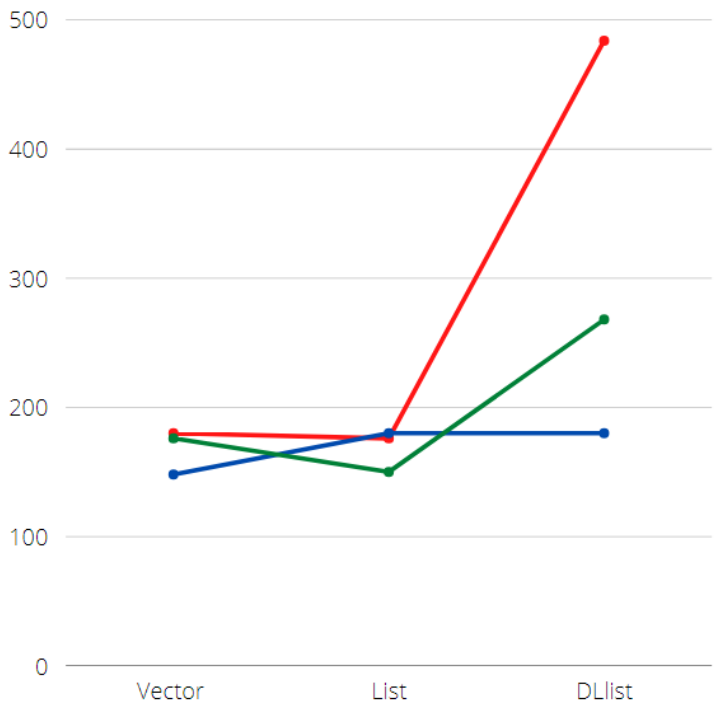
Even so, the concat method is practically the same in time in the list as in the double list.

4.4 Efficiency

There are many ways to measure the amount of resources used by an algorithm: the two most common measures are temporal and spatial. Here we are going to measure the efficiency of InsertFirst, InsertLast and InsertAt.

Through the space that each one occupies, the same, we can know the real differences in time in the Vector, List and DLlist structures

Each data structure occupies the same space so there are no differences in operations due to size. The time difference is due to the complexity and structure difference already mentioned, such as, for example, the speed of adding an element in a position of each given data structure.




```
Vector bytes 16

List bytes 16

DLlist bytes 16
```

5. Bibliography

1. https://es.wikipedia.org/wiki/Pila_de_llamadas#/media/Archivo:Call_stack_layout.svg

2. https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/

3. http://profesores.elo.utfsm.cl/~tarredondo/info/datos-algoritmos/ELO-320-Memoria.pdf

4. https://netting.wordpress.com/2016/10/01/segmentacion-de-memoria-de-un-programa-y-el-stack/

5. https://codingornot.com/diferencias-entre-heap-y-stack