QUASI-RANDOM ACCESS DYNAMIC ARRAY

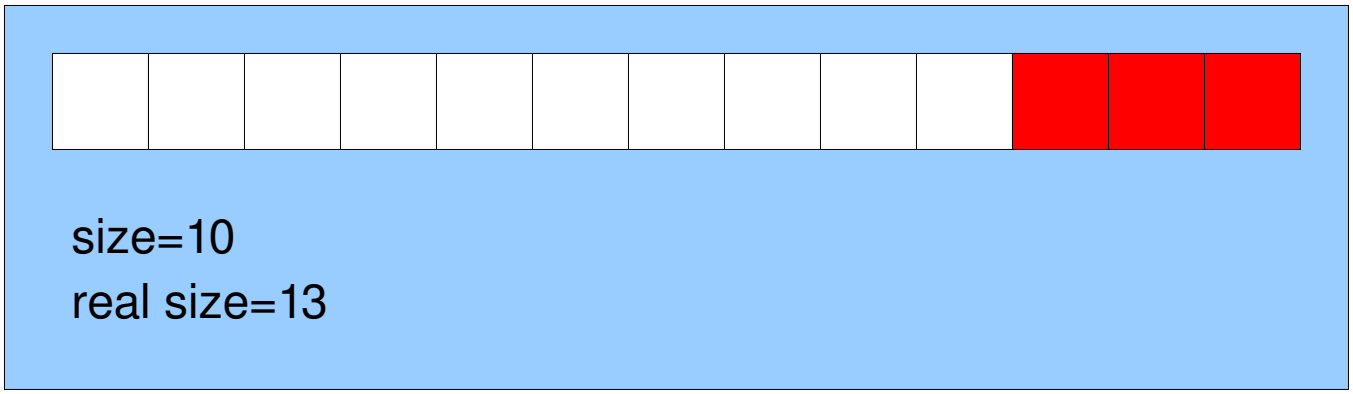
Giovanni Santostefano
(11-18-2008) ver. 1.0
http://santostefanogiovanni.blogspot.com
email: idmgiovanni@libero.it

The dynamic array is a linked list of static arrays.
This can give a semi-random access based on how many times array is extended out of bounds.

For a list of n static array we have an access complexity of O(n+1)
The dynamic array contains m elements, with m>n because every subarray had a surplus space of 1/3 of his declared dimension.
So if we declare 3 items array the real array is large 4 items.
If we extend the array of 1 item, no extra node is built.

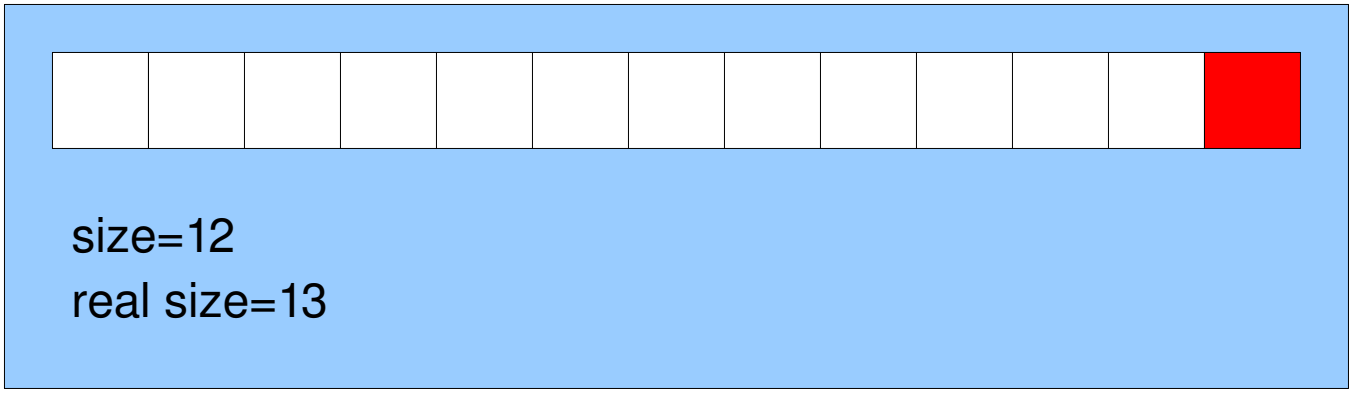
Lets see next how the dinamic array is built step by step.

STEP 1
Declaration of a dynamic array of 10 elements



subarrays=1 size=10 real size=13

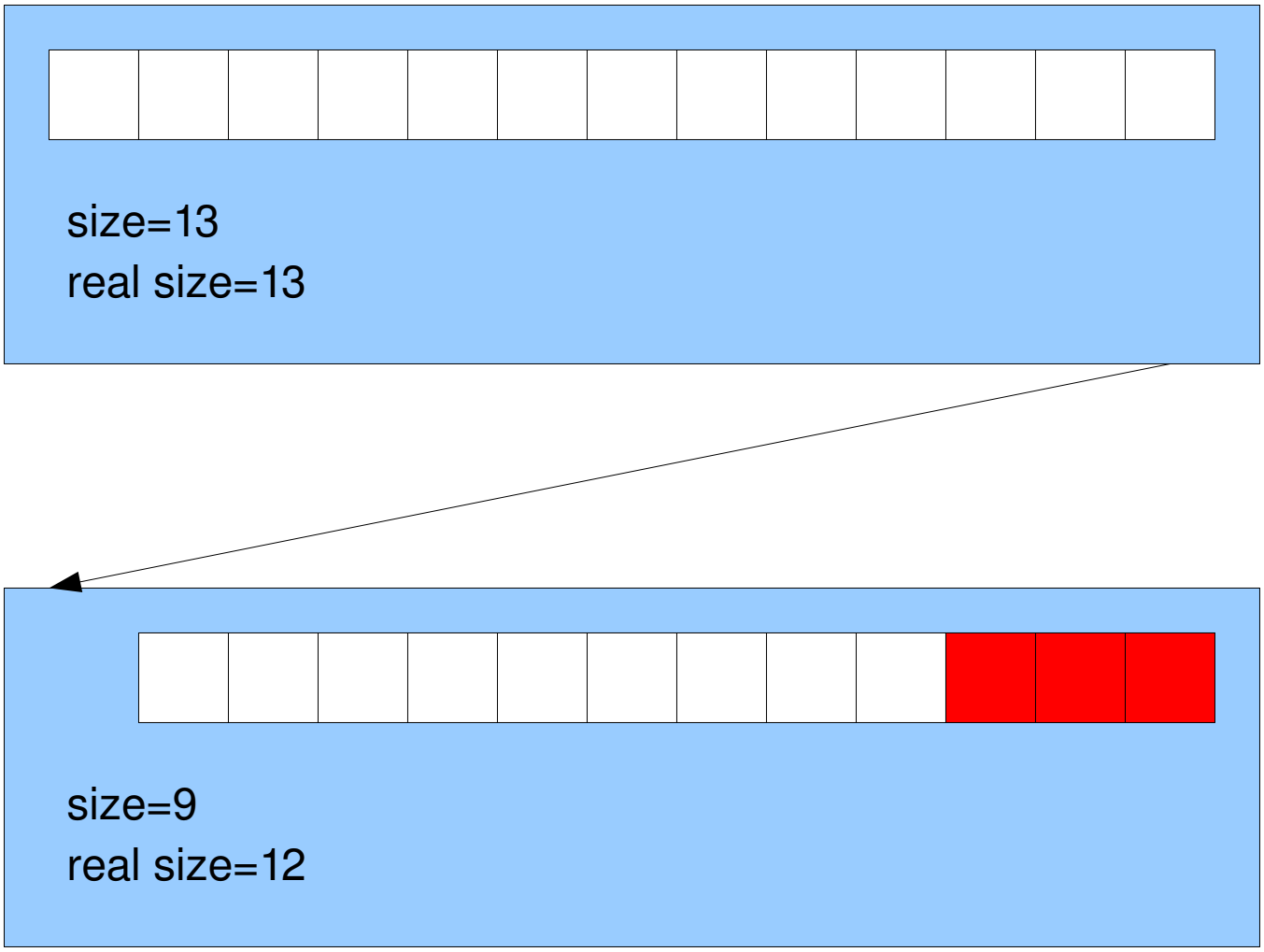
STEP 2
Enlarging the array by 2 elements



subarrays=1 size=12 real size=13

STEP 3
Enlarging the array by 10 elements

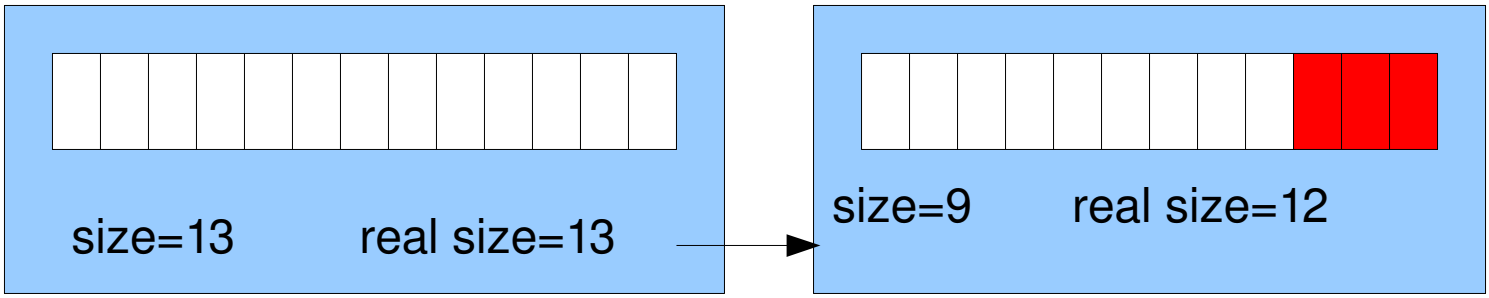
subarrays=2 size=22 real size=25



And so on!
When we extend an array over the extraspace another node is created with the static array of the proper dimension plus the new extra space

ACCESS POLITICS

To access to the array items we pass through the list and find the node that contains the array with the indexed item.
Take the previous example.



subarrays=2 size=22 real size=25

If it's supposed to access to the 5th element, we start reading the first node. We know that the node size is 13 > 5. Then we returns the data[5] element of the actual node. This is also the best case O(1).

Let's take for example the 20th node. We set shift=0.
We start by the first node and we see that the size see 13 < 20 then we set shift=shift+node.size (13) and proceed with the next node. The next node has shift+size that is 22 > 20. Then we have to return the 20-shift item -> data[20-shift].
So if n is the number of subarrays (nodes of the list) we can access to a generic item in O(n+1) => O(n).

ADVANTAGES / DISADVANTAGES

DISADVANTAGES
QUASI-random access
The disadvantage of this system compared to a generic vector is that we doesn't access to the elements in O(1) time but in O(n) with n the number of subarrays.

ADVANTAGES
This system avoid the copy of large datas in the case of array extension.
In the normal dynamic array when we extend over the extra size, we have to allocate a larger new array and copy all the datas inside the new one.
With this system we simply add a new node with the space required for the new items.

This dynamic array system is only good for lots of “big” array extensions

TECHNIQUES

ALLOCATION
To extend the array we must add an item on the tail of the list.
We have to also maintain a pointer to the list last node.
When we receive an access dynarr[x] with x greater than the dynamic array size we first control if the extra space (present only in the last node) can resolve our request.
If so we change the size parameter of the last node and return the control on the proper item.
Otherwise we have to change the size of the last node to his real size and allocate a new node with

new_node_items=x-(prev. array real size)+ (1/3*x-(prev. array real size))

new_node_sizeparam=x-(prev. array real size)

new_node_realsizeparam=x-(prev. array real size)+ (1/3*x-(prev. array real size))

Then we update the dynamic array descriptor and return the access to the requested item.

If we doesn't use the 1/3 extra space we can't prevent small allocations by creating new nodes. Without extra-space each allocation request must take one physical allocation.
So we can have an array of m items built into n nodes. With m=n we have the list same access complexity.

We count to have n<m

DEALLOCATION

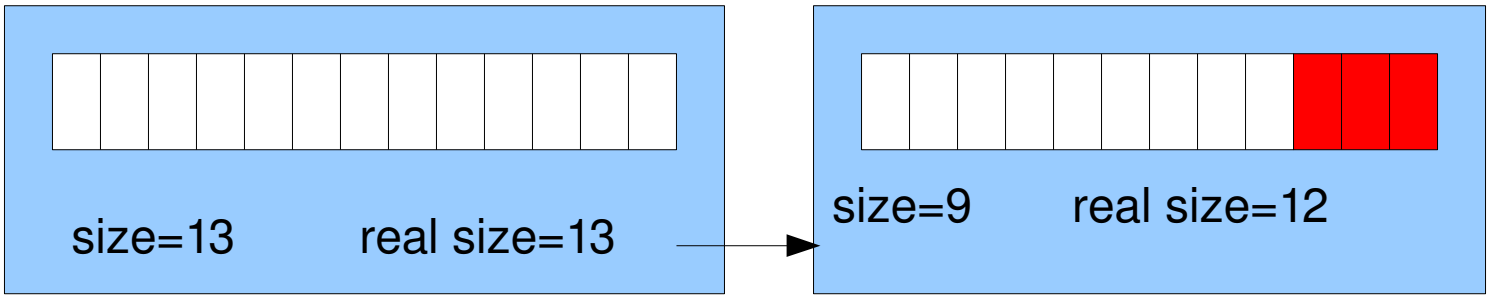
To reduce the size of the array we proceed removing nodes if necessary.

First we take a look the last node.

If (new_size > dynarray.size – lastnode.size) then

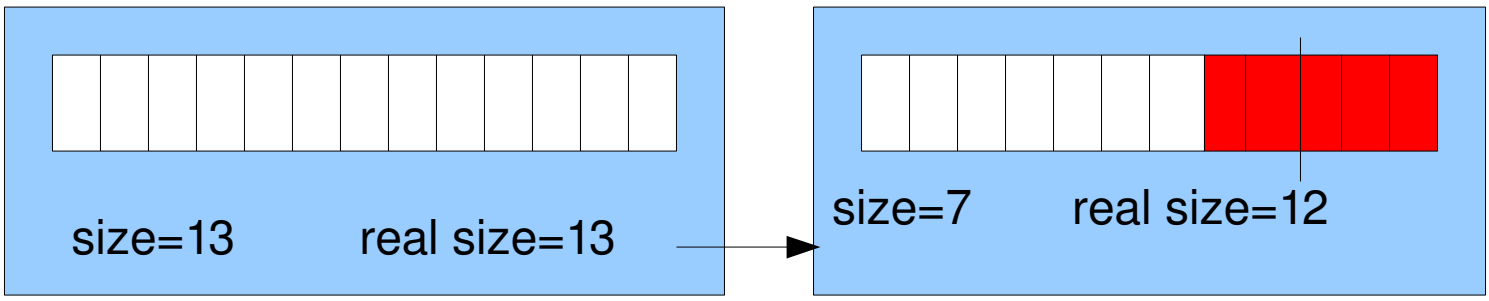
lastnode.size=lastnode.size-(new_size - (dynarray.size – lastnode.size))

So we have that



subarrays=2 size=22 real size=25

REDUCE TO SIZE 20

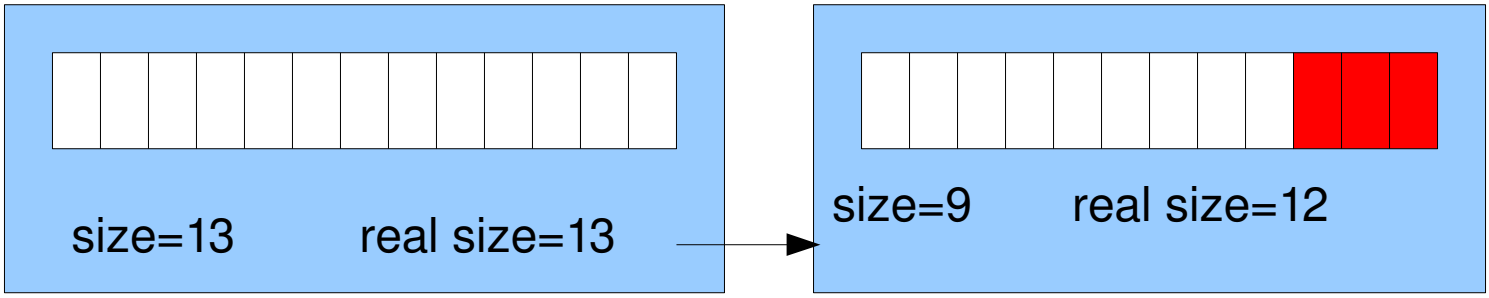


subarrays=2 size=20 real size=25

No deallocation is performed in this case. We also have more extra-space.

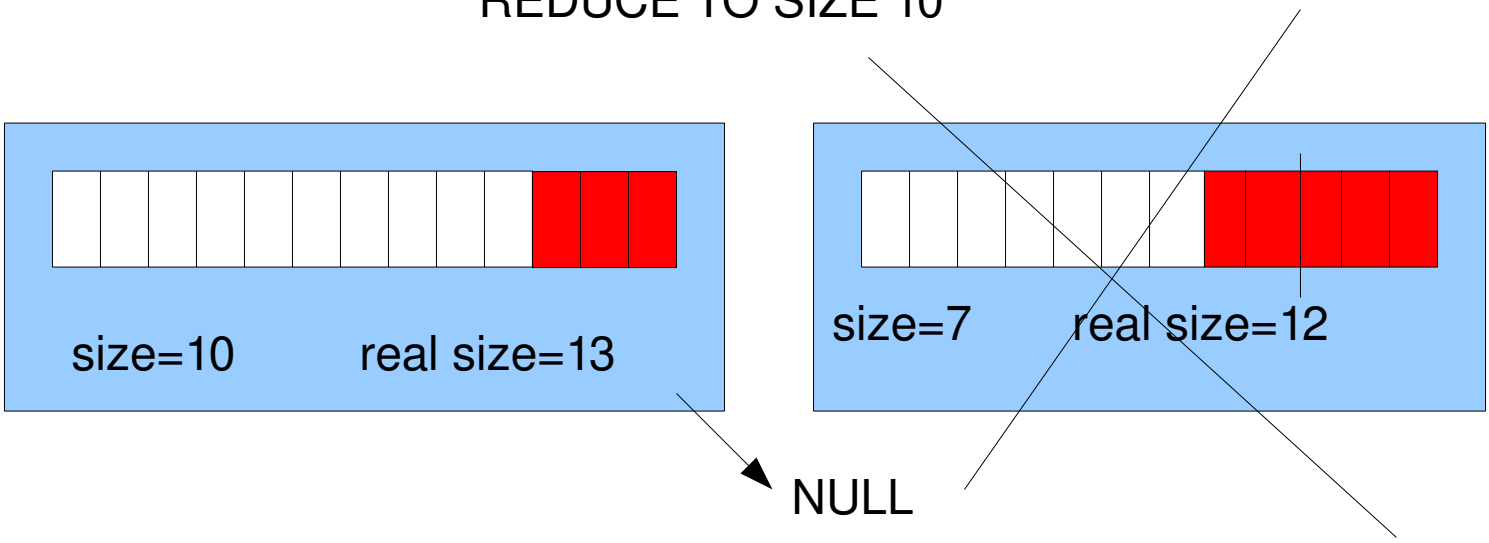
If (new_size < dynarray.size – lastnode.size) then
deallocate last node and
lastnode.size=lastnode.size-(new_size - (dynarray.size – lastnode.size))

So we have that



subarrays=2 size=22 real size=25

REDUCE TO SIZE 10



subarrays=1 size=10 real size=13

The deallocation is performed and we have the new extra-space. But this code is not really good because we have to show the real pseudo-work to remove many nodes.

So take a look to the pseudocode in the next page.

PSEUDOCODE FOR THE REMOVAL

```
while (new_size < dynarray.size – lastnode.size)
{
        dynarray.realsize=dynarray.size-lastnode.size;
        dynarray.size=dynarray.realsize;
        dynarray.subarrays=dynarray.subarrays-1;
        dynarray.last=lastnode.previous;
        dynarray.last.next=NULL;
        delete last node;
}

lastnode.size=lastnode.size-(new_size - (dynarray.size – lastnode.size));
```

CONCLUSIONS

This is the quasi-random access dynamic array.
As you can see in certain conditions is good to lose complete random access to elements to obtain fast array size enlargement without recopy lots of datas in a bigger array.
Good also for memory critical systems.

LICENCE