Dynamic Vector. Memory reallocation

automatic: if more than capacity() - size() elements are inserted into the vector

manual: if you know the capacity to which your vector must eventually grow, then it is usually more efficient to allocate that memory all at once

Strategies usually increase capacity it by a factor of two

- proportional to the current capacity

  inserting a series of elements into a vector is a ***linear*** time operation

- a fixed constant

  inserting a series of elements into a vector is a ***quadratic*** time operation

Reallocation

- Iterators that keep pointer to elements inside the vector

  any such iterator became invalid (pointer –address in memory)

- increase capacity : does not change size
- does not change the values of any elements of the vector

Capacity management (Java style)

Java – Vector

capacityIncrement
elementCount
elementData

Vector: deprecated , historical reasons

Vector() *an initial capacity of ten*
Vector(Collection c)
Vector(int initialCapacity)
Vector(int initialCapacity, int capacityIncrement)

- capacity()
- ensureCapacity(int minCapacity)
- trimToSize()

Capacity management (Java style)

Java –Array List

ArrayList: *The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.*
for example: newCapacity = (oldCapacity * 3)/2 + 1;

- ArrayList() : *an initial capacity of ten*.
- ArrayList(Collection c)
- ArrayList(int initialCapacity)

- void ensureCapacity(int minCapacity)
  before adding a large number of elements,
  using the ensureCapacity may reduce the amount of incremental reallocation
- void trimToSize()

Capacity management (C++ STL Vector)

empty

size

max_size

the maximum potential size the container can reach due to known system or library implementation limitations, but the container is by no means guaranteed to be able to reach that size

resize

---

capacity

reserve

Requests that the vector capacity be at least enough to contain n elements

shrink_to_fit [2011]

Requests the container to reduce its capacity to fit its size.

Container implementation is free to optimize otherwise and leave the vector with a capacity greater than its size.

Expansion and contraction

Load factor

$\alpha(...) = number\_of\_stored\_elements \ / \ number\_of\_allocated\_slots$

By convention

$\alpha(...) = 0 / 0 = 1$

Expansion and contraction

DELETE operation: it is enough to remove the specified item

- It is often desirable to contract the table, so that the wasted space is not exorbitant.
- Table contraction is analogous to table expansion
  allocate a new, smaller table and then copy the items

A natural strategy (**?!**)

- Expansion: double the table capacity when an item is inserted into a full table
- Contraction: halve the capacity when a deletion would cause the table to become less than half full.

➔ the load factor of the table never drops below 1/2, **but**:

Scenario. let ***n*** be an exact power of 2.

- perform ***n/2*** insertions, which by our previous analysis cost a total of (n). At the end of this sequence of insertions, *capacity* = *size* = n/2.
- perform ***n/2*** operations by following the sequence:
  I, D, D, I, I, D, D, I, I,... ,
  where I stands for an insertion and D stands for a deletion

the total cost of the n operations is (n^2), and the amortized cost of an operation is (n).

Expansion and contraction

Strategy (**improved**)

- halve the table size when a deletion causes the table to become less than 1/4 full, rather than 1/2 full as before

→ allow the load factor of the table to drop below 1/2

Contraction:

- occur when the load factor would fall below ¼
- after a contraction, the load factor of the table is also 1/2

(*Cormen*)

With this strartegy

the actual time for any sequence of n operations on a dynamic table is O(n)

Regular **asymptotic analysis** looks at the performance of an individual operation.

Amortized analysis deals with the total cost over a number of runs of the routine

- is a worst-case analysis
- gives the average performance of an operation
  - a sequence of invocations of the operation

<u>Example:</u>

- A dynamic array that doubles in size when needed
- Subalg. addLast(v, el)

Regular **asymptotic analysis**
Subalg. addLast(v, el) costs O(n)
*because it **might** need to grow and copy all elements to the new array.*

Amortized analysis
adding an item really costs O(1) on average
takes into account that in order to have to grow, n/2 items must have been added without causing a grow since the previous grow

Amortized analysis on the next code:

Convention:

```
        v.els : 0-based array

Subalg. createEmpty()
        v.n=0;
        v.cap=0;
        v.els=NIL
end_createEmpty

Subalg. addLastWithRealloc1(v,el)          // double capacity
        If v.cap = 0 then
                v.cap=1;
                v.els = new TElement[1]
        Else
                If v.n = v.cap then
                        newEls = new TElement[2*v.cap]
                        for i=0, v.n-1 do              // copy els
                                newEls [i]=v.els[i]
                        endfor
                        delete [] v.els
                        v.els= newEls
                        v.cap = 2 * v.cap
                endif
        endif
        v.els[v.n] = el
        v.n=v.n+1
end_addLastWithRealloc1


Subalg. nxaddLast(v)
        createEmpty(v)
        for i:=1, n do
                @read el
                addLastWithRealloc1(v,el)
        endfor
End_nxaddLast
```

```
Subalg. addLastWithRealloc2(v,el)             // cap. increment = 4
    If v.cap = 0 then
        v.cap=1;
        v.els = new TElement[4]
    Else
        If v.n = v.cap then
            newEls = new TElement[v.cap + 4]
            for i=0, v.n-1 do                    // copy els
                newEls [i]=v.els[i]
            endfor
            delete [] v.els
            v.els= newEls
            v.cap = v.cap + 4
        endif
    endif
    v.els[v.n] = el
    v.n=v.n+1
end_addLastWithRealloc2
```

```
Subalg. removeLastWithShrink1 (v)            // half capacity
    v.n=v.n-1
    If v.n*2 = v.cap then
        newEls = new TElement [ v.cap div 2 ]
        for i=0, v.n-1 do                    // copy els
            newEls [i]=v.els[i]
        endfor
        delete [] v.els
        v.els= newEls
        v.cap = v.cap div 2
    endif
end_removeLastWithShrink1
```