Linear (or Ordered) Lists ADT

instances are of the form

$(e_0, e_1, e_2, \ldots, e_{n-1})$

where e_i denotes a list element

$n >= 0$ is finite

list size is n

Linear Lists

$L = (e_0, e_1, e_2, e_3, \ldots, e_{n-1})$

relationships

e_0 is the zero'th (or front) element

e_{n-1} is the last element

e_i immediately precedes e_{i+1}

Linear List Examples/Instances

Students in a course =

(Jack, Jill, Abe, Henry, Mary, …, Judy)

Exams =

(exam1, exam2, exam3)

Days of Week = (S, M, T, W, Th, F, Sa)

Months = (Jan, Feb, Mar, Apr, …, Nov, Dec)

Linear List Operations—size()

determine list size

$$L = (a,b,c,d,e)$$

size = 5

Linear List Operations—get(theIndex)

get element with given index

$L = (a,b,c,d,e)$

get(0) = *a*

get(2) = *c*

get(4) = *e*

get(-1) = error

get(9) = error

Linear List Operations—indexOf(theElement)

determine the index of an element

$L = (a,b,d,b,a)$

indexOf(d) = *2*

indexOf(a) = *0*

indexOf(z) = *-1*

Linear List Operations—remove(theIndex)

remove and return element with given index

$$L = (a,b,c,d,e,f,g)$$

remove(2) returns *c*

and *L* becomes *(a,b,d,e,f,g)*

index of *d,e,f,* and *g* decrease by *1*

Linear List Operations—remove(theIndex)

remove and return element with given index

$L = (a,b,c,d,e,f,g)$

remove(-1) => error

remove(20) => error

Linear List Operations—add(theIndex, theElement)

add an element so that the new element has a specified index

$$L = (a,b,c,d,e,f,g)$$

$add(0,h) \Rightarrow L = (h,a,b,c,d,e,f,g)$

index of $a,b,c,d,e,f,$ and g increase by 1

Linear List Operations—add(theIndex, theElement)

$L = (a,b,c,d,e,f,g)$

add(2,h) => $L = (a,b,h,c,d,e,f,g)$

index of *c,d,e,f,* and *g* increase by *1*

add(10,h) => error

add(-6,h) => error

Data Structure Specification

- Language independent
  - Abstract Data Type

Linear List Abstract Data Type

AbstractDataType *LinearList*

{

instances

ordered finite collections of zero or more elements

operations

isEmpty(): return true iff the list is empty, false otherwise

size(): return the list size (i.e., number of elements in the list)

get(index): return the *index*th element of the list

indexO f(x): return the index of the first occurrence of *x* in the list, return -1 if *x* is not in the list

remove(index): remove and return the *index*th element, elements with higher index have their index reduced by 1

add(theIndex, x): insert x as the *index*th element, elements with theIndex >= *index* have their index increased by 1

output(): output the list elements from left to right

}

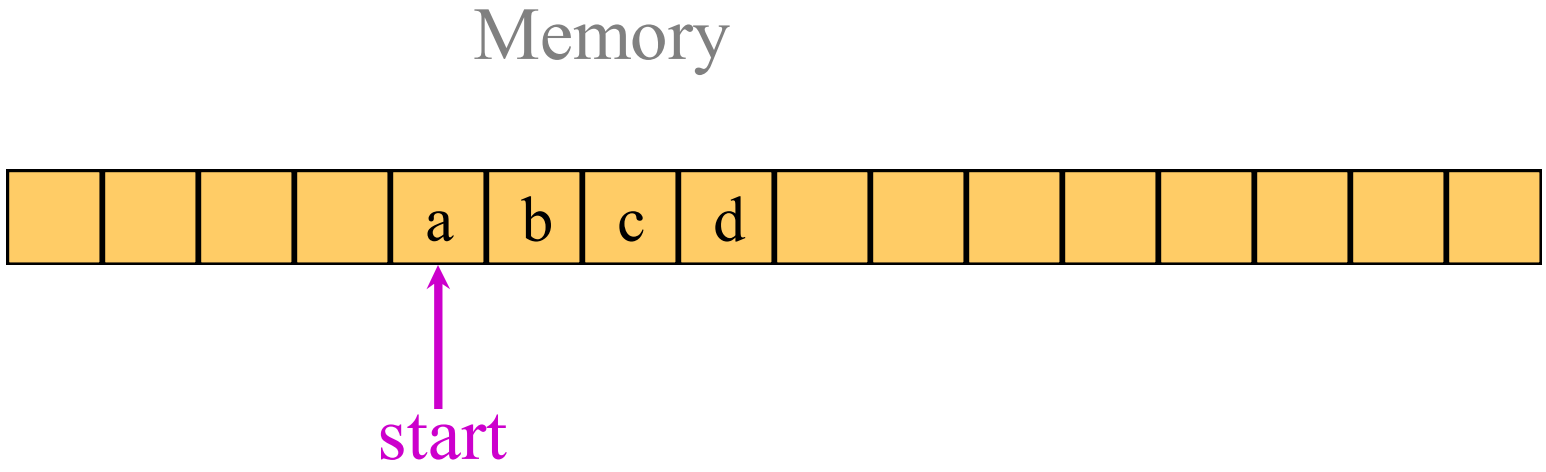
Implementation of Linear List

By Arrays

By Linked Lists

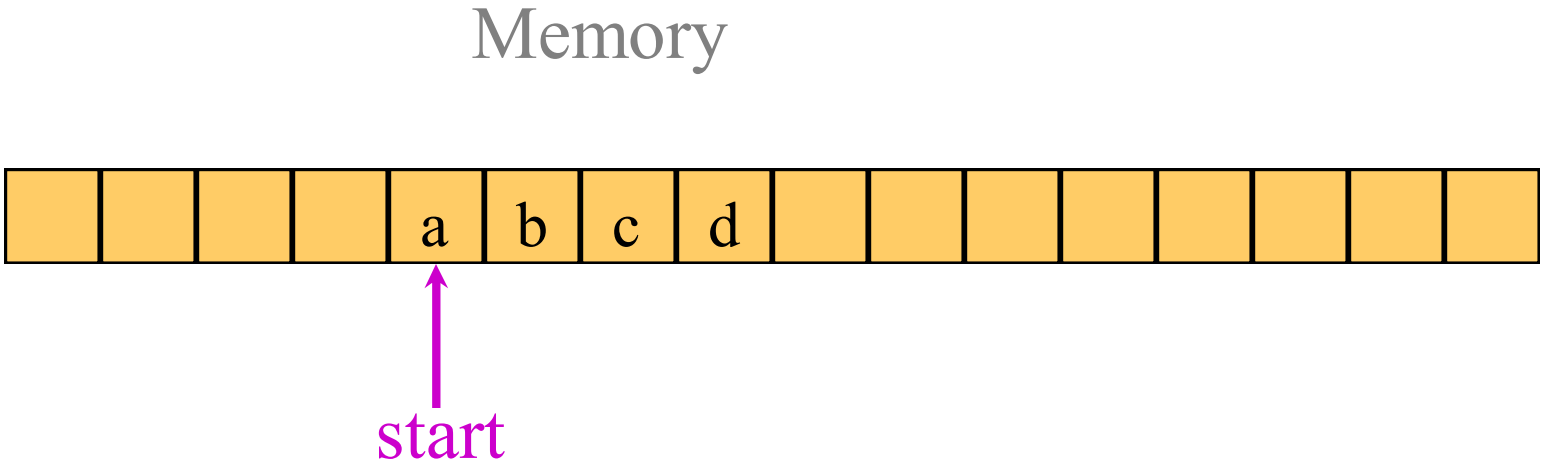
Array Implementation of List

1D Array Representation In C



- 1-dimensional array x = [a, b, c, d]
- map into **contiguous memory** locations

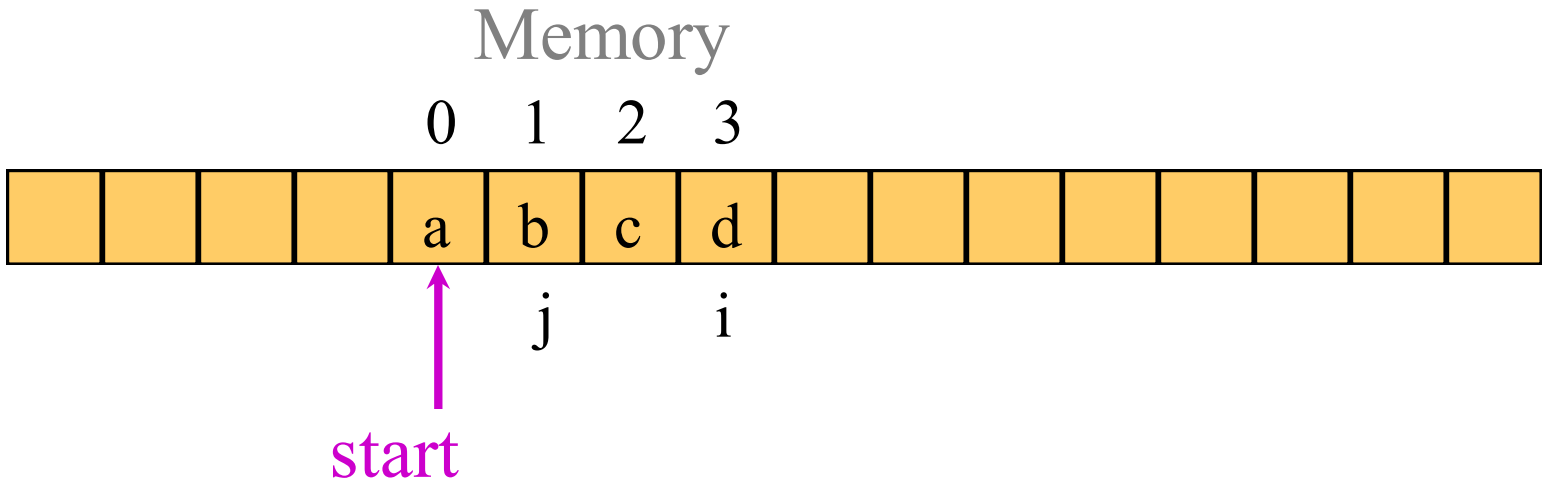
Space Overhead



space overhead = 4 bytes for start

(excludes space needed for the elements of x)

Space Overhead



- location(x[i]) = start + w.i
  - w = size of each element

- location(x[i]) = location(x[j]) + w.(i-j)

2D Arrays

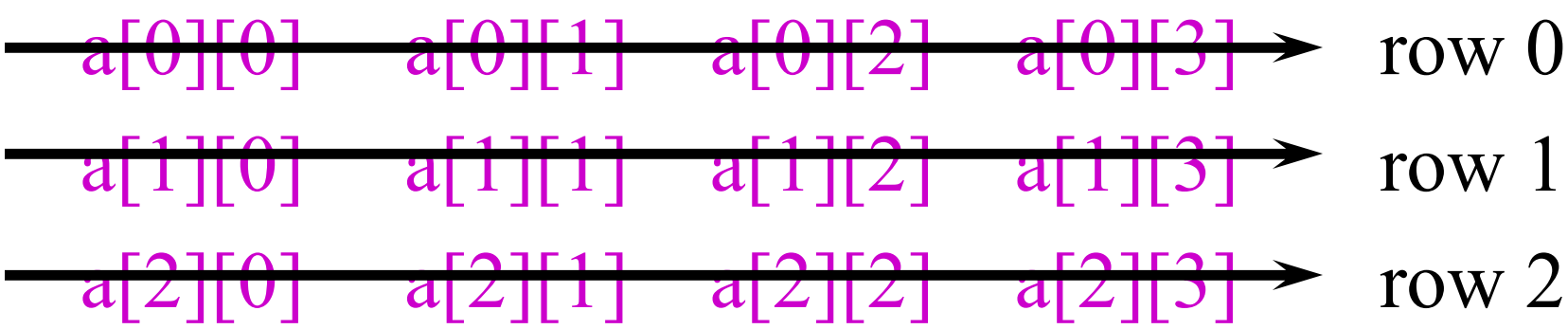
The elements of a 2-dimensional array a declared as:

```
int a[3][4] ;
```

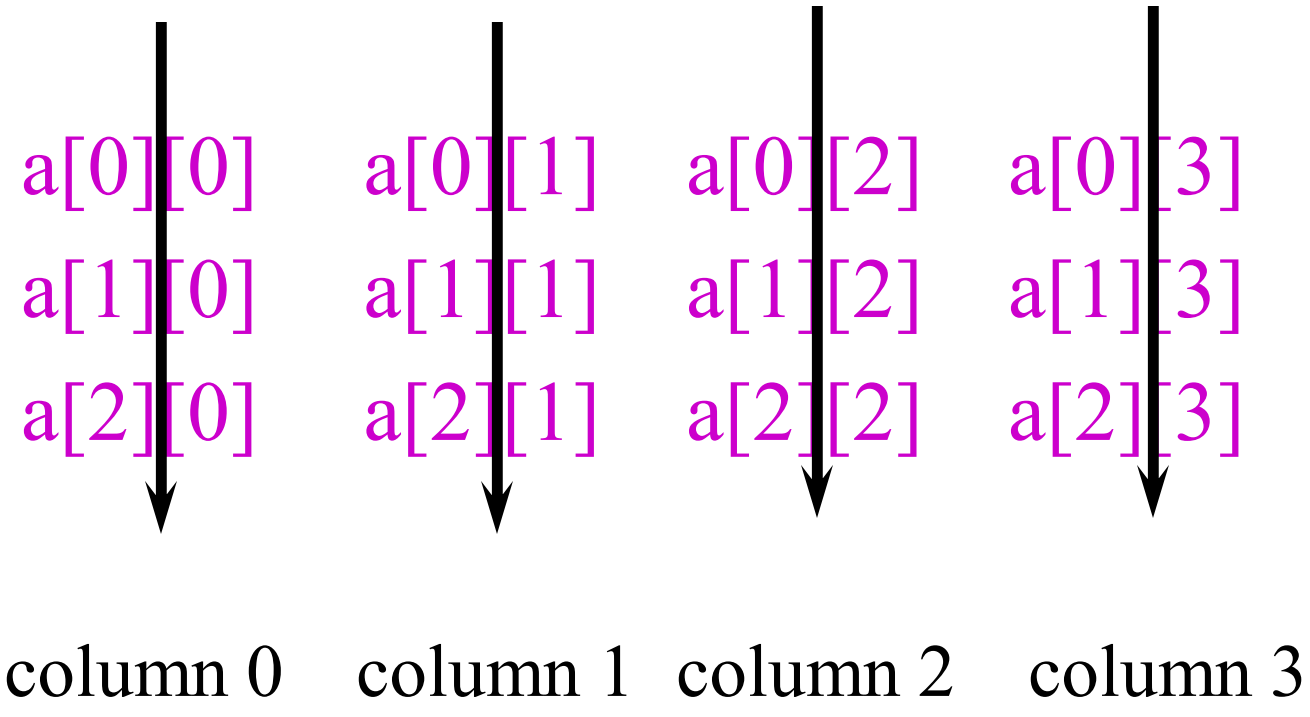
may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Rows Of A 2D Array



Columns Of A 2D Array



2D Array Representation

2-dimensional array x

a, b, c, d

e, f, g, h

i, j, k, l

view 2D array as a 1D array of rows

x = [row0, row1, row 2]

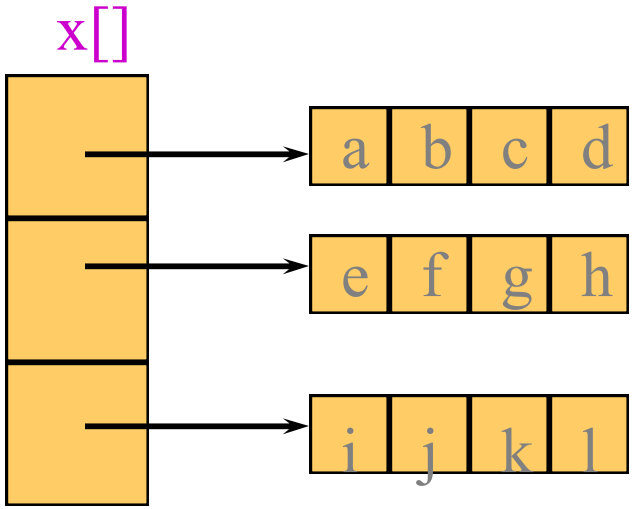
row 0 = [a,b, c, d]

row 1 = [e, f, g, h]

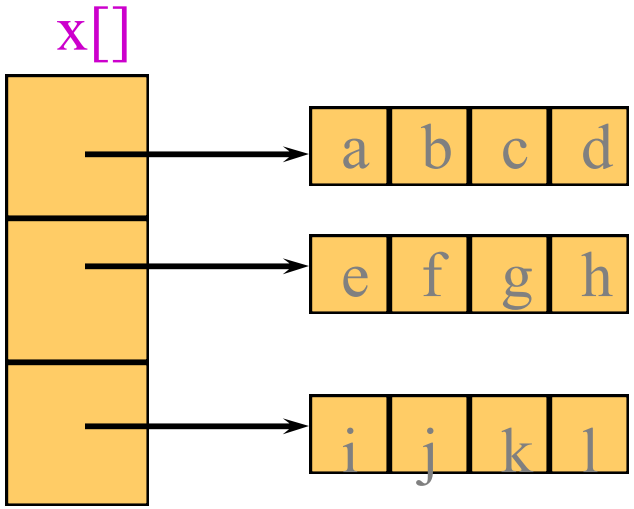
row 2 = [i, j, k, l]

and store as 4 1D arrays

2D Array Representation In C



Space Overhead



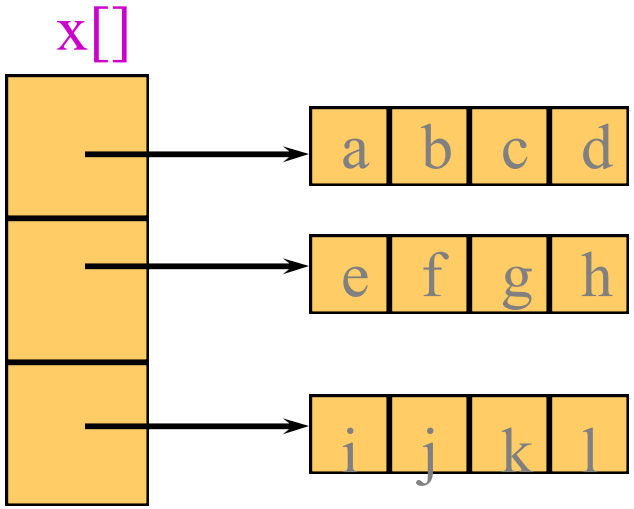
space overhead = space required by the array x[]

= 3 * 4 bytes

= 12 bytes

= number of rows x 4 bytes

Array Representation In C



- This representation is called the **array-of-arrays** representation.
- Requires contiguous memory of size 3, 4, 4, and 4 for the 4 1D arrays.
- 1 memory block of size number of rows and number of rows blocks of size number of columns

Row-Major Mapping

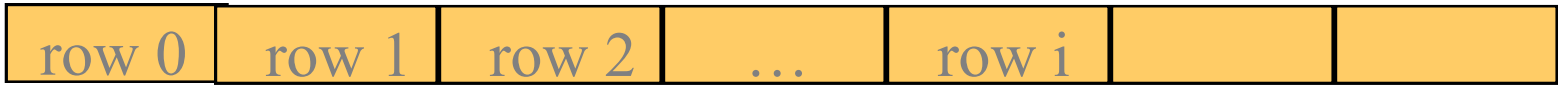
- Example 3 x 4 array:

a b c d

e f g h

i j k l

- Convert into 1D array by collecting elements by rows.
- Within a row elements are collected from left to right.
- Rows are collected from top to bottom.
- We get {a, b, c, d, e, f, g, h, i, j, k, l}



Locating Element x[i][j]

- assume x has r rows and c columns
- each row has c elements
- i rows to the left of row i
- so ic elements to the left of x[i][0]
- so x[i][j] is mapped to position

ic + j of the 1D array

Column-Major Mapping

a b c d

e f g h

i j k l

- Convert into 1D array by collecting elements by columns.
- Within a column elements are collected from top to bottom.
- Columns are collected from left to right.
- We get {a, e, i, b, f, j, c, g, k, d, h, l}

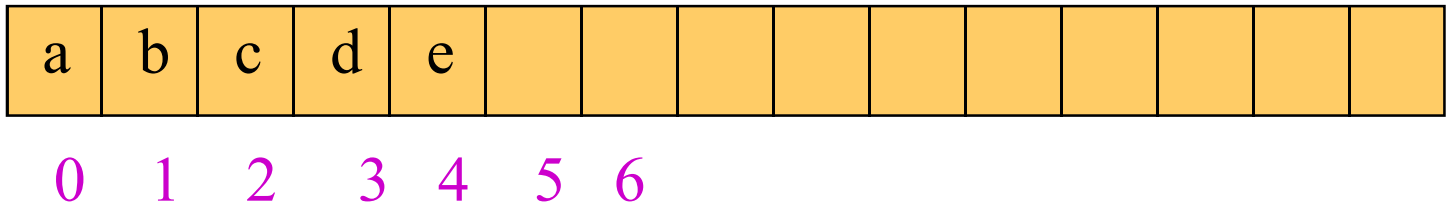
Locating Element x[i][j]

- assume x has r rows and c columns
- each column has r elements
- j columns to the left of column j
- so jr elements to the left of x[0][j]
- so x[i][j] is mapped to position

jr + i of the 1D array

Linear List Array Representation

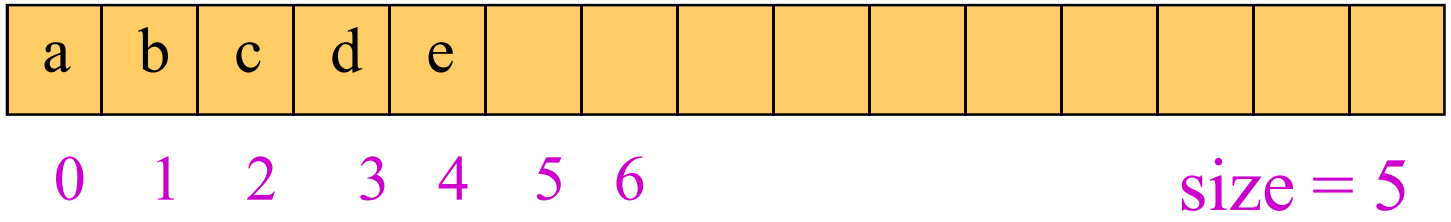
use a one-dimensional array element[]



L = (a, b, c, d, e)

Store element i of list in element[i].

Representation

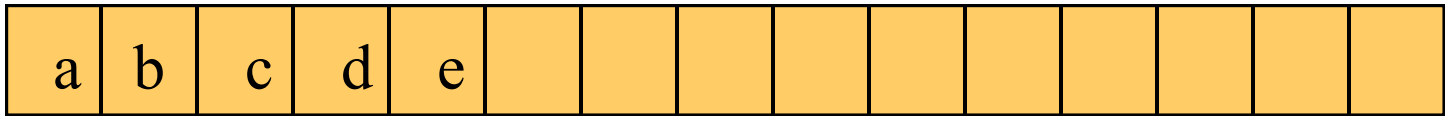


put element i of list in element[i]

use a variable size to record current number of elements

Add An Element

size = 5



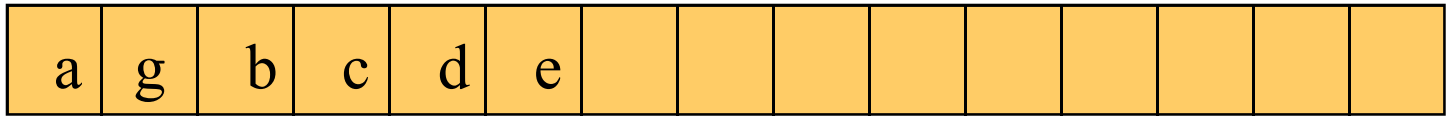
add(1,g)

add(k,val)

```
for(i=size-1,i>=k,i--)
    x[i+1]=x[i];
```

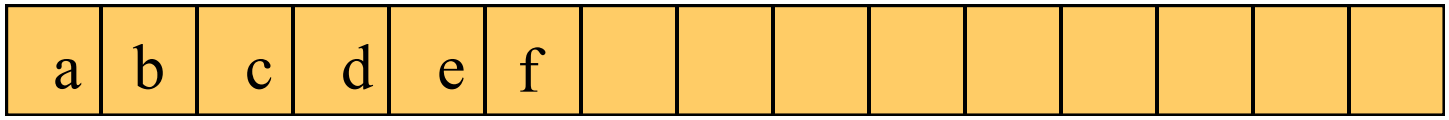
```
x[k]=val;  size++;
```

size = 6



Remove An Element

size = 6



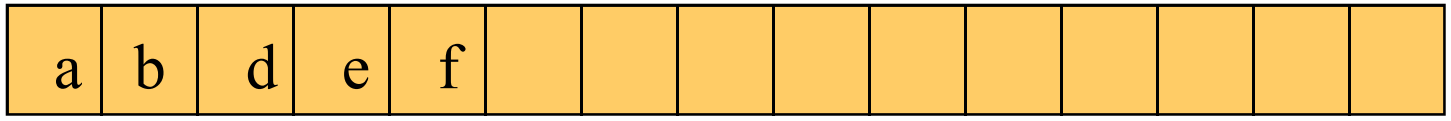
remove(2)

remove(k)

```
val=x[k];
for(i=k,i<size,i++)
    x[i]=x[i+1];

size--; return(val);
```

size = 5



Disadvantage

Need to estimate the maximum demand (r and c)

Solution: Dynamic Memory Allocation

Need **contiguous** memory of size rc.

Solution: Linked Lists