Data Structures for Mathematical Objects

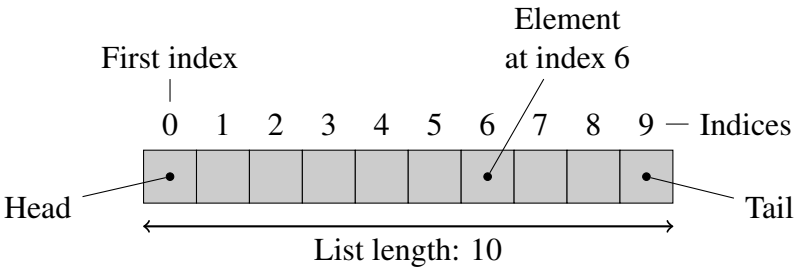
The purpose of this note is to provide an introduction to some abstract data types that are frequently used to represent mathematical objects such a vectors, matrices, tensors, polynomials, etc. We start be defining the notion of a *list* that can be used to represent and store a collection of *elements* or *objects*.

1 Lists

A *list* is an ordered set of elements with the following properties:

- an element can be accessed, inserted, or deleted at any position,
- a list can be split into sublist,
- two lists can be concatenated.

The length of the list is equal to the number of elements in the list, and each element is assigned an index. In a list of length n, the indices either range from 0 to $n-1$ (0-based indexing) or from 1 to n (1-based indexing). The element with the lowest index is the first element in the list (aka the *head* of the list) whereas the element with the largest index is the last element (aka the *tail* of the list). The below figure illustrates the terminology for 0-based indexing.



To illustrate the different list operations, we define three lists of integers:

$$L_1 = (5,3,9), \qquad L_2 = (9,4,7,0), \qquad L_3 = (10,-2,5,8).$$

The concatenation of L_1 and L_2, denoted $L_1 \frown L_2$, is the list

$$(5,3,9) \frown (9,4,7,0) = (5,3,9,9,4,7,0)$$

whereas $L_2 \frown L_1$ is the list

$$(9,4,7,0) \frown (5,3,9) = (9,4,7,0,5,3,9).$$

Assuming 0-based indexing, splitting L_3 at index 1 yields two lists, i.e.,

$$\text{split}(L_3, 1) \rightsquigarrow (10),\ (-2, 5, 8).$$

Accessing the 3rd element of L_2 (i.e., the element at index 2) yields

$$\text{access}(L_2, 2) \rightsquigarrow 7,$$

inserting the value 8 at index 2 of L_1 yields

$$\text{insert}(L_1, 2, 8) \rightsquigarrow L_1 = (5, 3, 8, 9),$$

and deleting the head of L_2 (i.e., its first element) yields

$$\text{delete}(L_2, 0) \rightsquigarrow L_2 = (4, 7, 0).$$

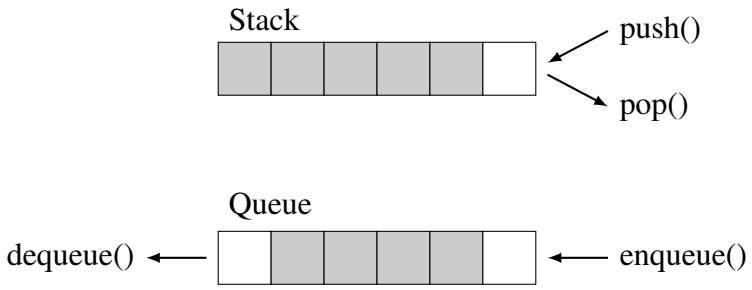
It is clear from the above example that lists are dynamic in the sense that their lengths may change. The abstract definition of a list can be implemented in software in various ways. We will consider two data types that can be used for this purpose, namely *linked lists* and *dynamic arrays*.

2 Stacks and queues

A *stack* can be defined as a list in which elements can be inserted and deleted at one end only (typically at the tail). This is also called a *first-in-last-out* (FILO) structure since the first element to be inserted cannot be deleted before the elements at a higher index are deleted. The terminology is often somewhat different: the insert and delete operations are typically called *push* and *pop*, and the number of elements is commonly referred to as the *size* of the stack.

A closely related abstract data type is a *queue* which can be defined as a list in which elements are inserted at one end and deleted/extracted at the other. Elements usually enter the queue at the *rear* (the tail) and come out at the front (the head). This is also called a *first-in-first-out* (FIFO) data structure. The insert operation is typically called *enqueue* whereas the delete operation is called *dequeue*.

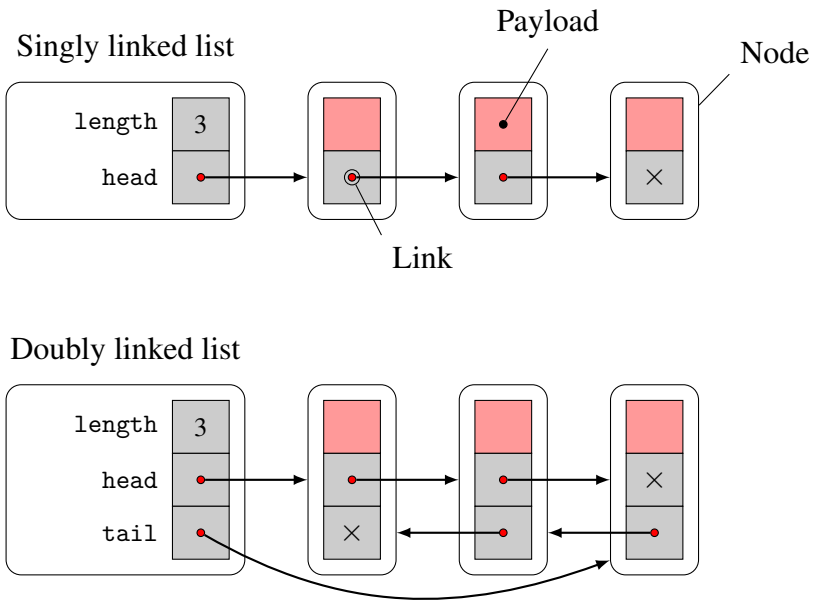
The following figure illustrates a stack and a queue as abstract data types.



3 Linked lists

One way to implement a list is to construct a so-called *linked list* in which each object/element is stored in a so-called *node* along with a *link* to another node (or possibly several links to other

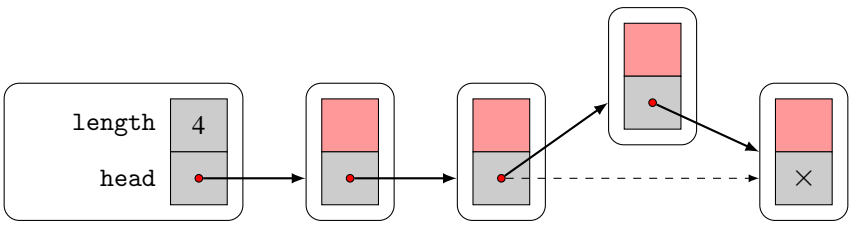
nodes). In a *singly linked list*, each node contains a *payload* (the object stored in the node) and a link to the next node. In addition to a link to the next node, nodes in a *doubly linked list* also store a link to the previous node. The following figure illustrates the difference between a singly linked list and a doubly linked list with three nodes.



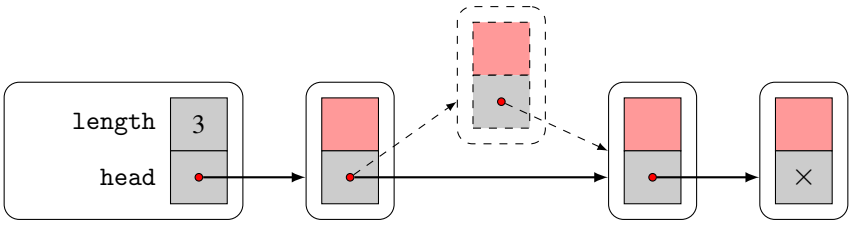
The tail of a singly linked list does not have a "next" node, so its link is empty, as indicated by the "×" in the figure. In a doubly linked list, the head does not have a "previous" link, and the tail does not have a "next" link.

The nodes of a linked list are generally not consecutive in memory. Thus, to access a given node in a singly linked list, we need to start at the head and repeatedly follow the link to the next node until the desired node is reached. As a consequence, the time required to access the tail of a linked list is proportional to the length of the list. A double linked list also contains a link to the tail elements which allows fast access at the head and tail. However, the time required to locate a node in the middle of a doubly linked list is still proportional to n.

To insert a new node at a specific position in a singly linked list, we first need a link to node that should precede the new node. Afterwards the insertion operation amounts to updating links, as illustrated in the following figure where a node with payload e is inserted at index 2 (i.e., between the second and third nodes).



A node can be deleted in a similar fashion. First we locate the node that precedes the node that we wish to delete, then we update its link, and finally, the desired node can be deleted. This is illustrated in the next figure.



Singly linked list in C

A singly linked list for nodes that contain a `double` as payload may be implemented in C using the following data structures:

```c
typedef struct snode {
    double x;               // Payload
    struct snode * next;    // Pointer to next snode (NULL if tail)
} snode_t;

typedef struct sllist {
    size_t length;          // Length (number of nodes)
    snode_t * head;         // Pointer to head (NULL if empty)
} sllist_t;
```

These data structures can be used to implement the generic list functionality described in the beginning of this note. For example, allocation and deallocation may be implemented based on the following prototypes:

```c
sllist_t * sllist_alloc(size_t len);
void sllist_dealloc(sllist_t * L);
snode_t * snode_alloc(double x);
void snode_dealloc(snode_t * N);
```

Furthermore, list operations (access, insertion, deletion, concatenation, and splitting) may be implemented based on the following prototypes:

```c
int sllist_insert(sllist_t *L, size_t i, double x);
int sllist_delete(sllist_t *L, size_t i);
snode_t *sllist_find(sllist_t *L, size_t i);
int snode_insert_after(snode_t *N, double x);
sllist_t *sllist_concat(sllist_t *L1, sllist_t *L2);
sllist_t *sllist_split(sllist_t *L, size_t i);
```

Doubly linked list in C

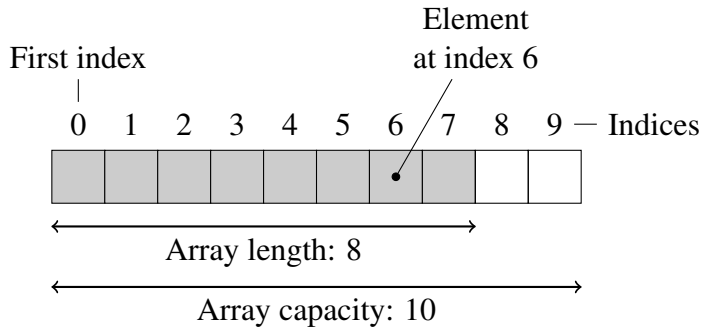
A doubly linked list for nodes with a `double` payload may be implemented in C using the following data types:

```c
typedef struct dnode {
    double x;               // Payload
```

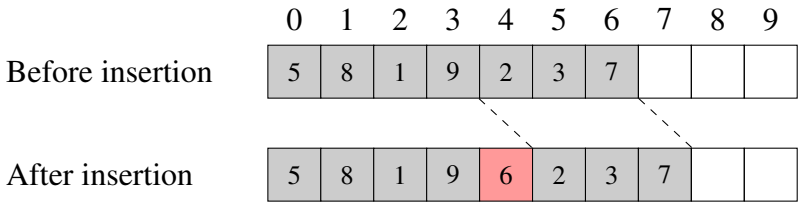
```c
    struct dnode * next;  // Pointer to next dnode (NULL if tail)
    struct dnode * prev;  // Pointer to previous dnode (NULL if head)
} dnode_t;

typedef struct dllist {
    size_t length;        // Length (number of nodes)
    dnode_t * head;       // Pointer to head (NULL if empty)
    dnode_t * tail;       // Pointer to tail (NULL if empty)
} dllist_t;
```

4 Dynamic arrays

A *dynamic array* is a contiguous collection of homogeneous elements with a given capacity that can be increased (or decreased) on demand. In other words, every element is of the same type, they are stored consecutively in memory, and the number of elements can be changed dynamically. The *capacity* of a dynamic array is the number of elements that can be stored without increasing the size of the array, and the *length* of the array refers to the number of elements currently in use. Indices can be 0-based or 1-based. An example of a dynamic array of length 8, capacity 10, and 0-based indexing is illustrated below.



Dynamic arrays can be used to implement a list. Access to an element of a dynamic array is fast: given the location of the first element, the location of the kth element is readily computed and accessed. Inserting an element in a dynamic array is generally more cumbersome. First, if the array is full (i.e., the length is equal to the capacity), then the capacity must first be increased before an element can be inserted. Second, elements at and after the position of insertion must be moved in order to make space for the new element. The following figure illustrates the array insertion operation: the value 6 is inserted at index 4 in a dynamic integer array of length 7.



Appending an element at the end of a dynamic array does not require any elements to be moved, and hence it is generally much more efficient than inserting an element in at the beginning of the array (e.g., at index 0). Note also that increasing the capacity may involve moving all elements to a new location in memory, and hence the capacity is usually increased by a constant

factor (e.g., by a factor of 2) rather than a fixed amount in order to avoid frequent resizing in applications where a large (but unknown) number of elements are inserted one by one.

Dynamic array in C

The following data structure can be used to implement a dynamic array of type `double` in C:

```c
typedef struct array {
    size_t len;         // Length of array
    size_t capacity;    // Capacity
    double * val;       // Pointer to array of length 'len'
} array_t;
```

Allocation and deallocation may be implemented as follows:

```c
array_t *array_alloc(const size_t capacity) {
  array_t *a = malloc(sizeof(*arr));
  if (a == NULL) return NULL;
  a->capacity = (capacity > 0 ? capacity : 1); // Minimum capacity: 1
  a->len = 0;
  a->val = malloc((a->capacity) * sizeof(*(a->val)));
  if (a->val == NULL) { free(a); return NULL; }
  return a;
}
```

```c
void array_dealloc(array_t *a) {
  if (a == NULL) return;
  free(a->val);
  free(a);
}
```

To resize the array (i.e., increase or decrease the capacity) we define the following function:

```c
int array_resize(array_t * a, size_t new_capacity) {
    if (!a) return 1;     // Received null pointer
    double * tmp = realloc(a->val, new_capacity * sizeof(double));
    if (!tmp) return 2;   // Reallocation failed
    a->val = tmp;
    a->capacity = new_capacity;
    a->len = (a->len <= new_capacity) ? a->len : new_capacity;
    return 0;             // Reallocation successful
}
```

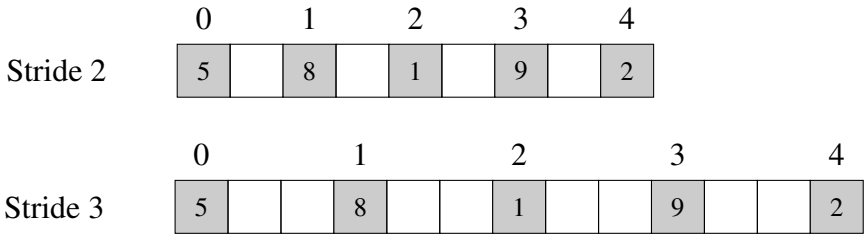
Finally, to insert an element at the end of the array, we define the following function:

```c
int array_push_back(array_t * a, double x) {
    if (!a) return 1;     // Received null pointer
```

```c
    if (a->capacity <= a->len) {
        int ret = array_resize(a,2*(a->capacity));
        if (ret != MSP_SUCCESS) return ret;
    }
    a->val[a->len++] = x;
    return 0;  // Success
}
```

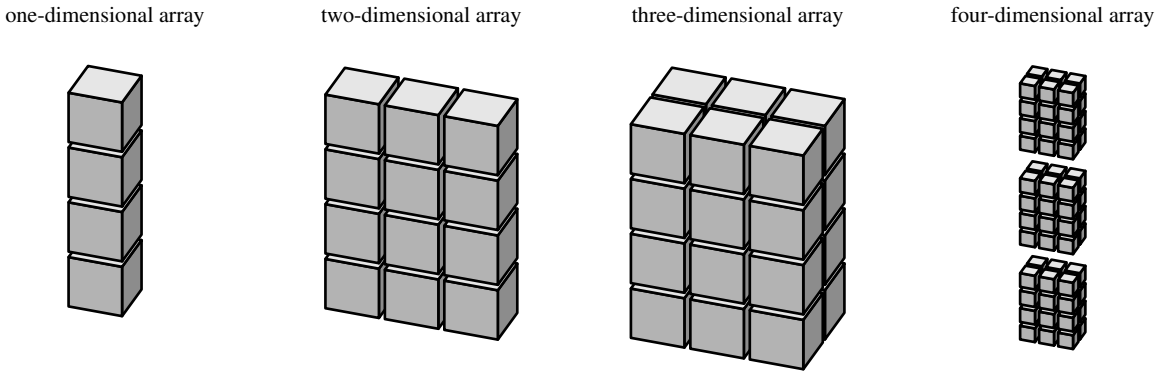
5 Strided arrays

Our definition of a dynamic array requires the elements to be consecutive in memory. Such an array is referred to as a *unit stride* array since the next and/or previous element is a single "unit" away in memory. It is sometimes useful to relax this assumption by allowing regularly spaced elements that are not necessarily adjacent. Such arrays are called non-unit stride arrays. The stride of an array may be defined as the distance between successive elements in memory, and it can be measured in units (i.e., the number of elements) or bytes. (We will use units in this note.) The following figure illustrates two non-unit stride arrays of length 5.



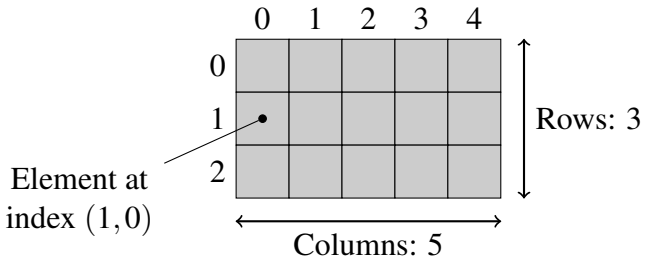
6 Multidimensional arrays

On a conceptual level, a multidimensional array is a contiguous collection of homogeneous elements that are indexed by a *multi-index* which is a tuple of indices. For example, a two-dimensional array is indexed by a tuple (i, j), and an k-dimensional array is indexed by a k-dimensional multi-index which is a k-tuple of the form $(i_1, \ldots, i_k)$. The *size* or *shape* of a k-dimensional array is also a k-tuple of the form $(m_1, \ldots, m_k)$, and it is sometimes written as $m_1 \times \cdots \times m_k$. The figure below illustrates multidimensional arrays with up to four dimensions.

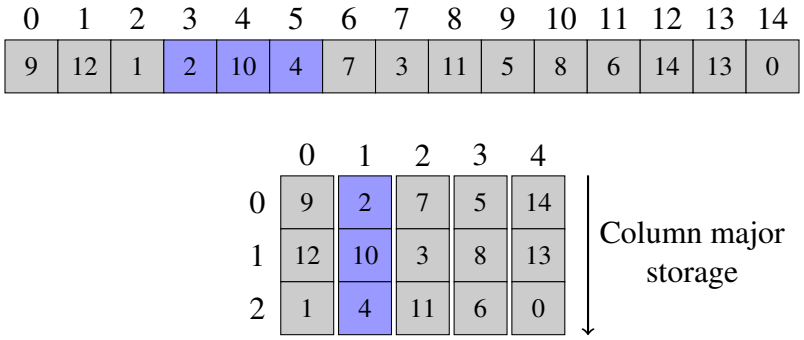


The one-dimensional array as size (4), the two-dimensional array has size $(4,3)$, the three-dimensional array has size $(4,3,2)$, and the four-dimensional array has size $(4,3,2,3)$.

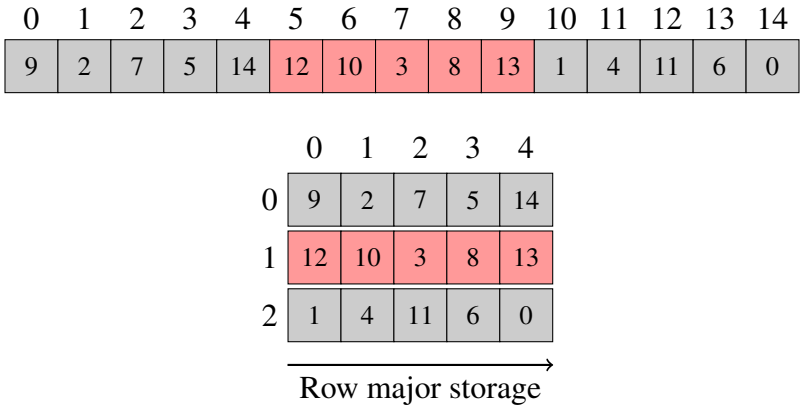
For a two-dimensional array of size (m,n), the indices m and n generally refer to the number of rows and columns, respectively. The following figure shows an example of a two-dimensional array of size $(3,5)$ with 0-based indexing.



Two-dimensional arrays can be mapped to memory in different ways. The so-called *column-major* storage order stores the elements of the two-dimensional array column by column in a one-dimensional array, as illustrated below.



In constrast, the so-called *row-major* order stores the elements row by row, as illustrated in the next figure.



Arrays with row-major storage order are also sometimes referred to as *C-style* arrays whereas the term *Fortran-style* array refers to the column-major storage order. Not surprisingly, this is because C stores automatically allocated multidimensional arrays using the row-major storage order whereas Fortran stores them using the column-major storage order. Note that the columns of a column-major two-dimensional array have unit stride, but the rows have stride m if the array has m rows. Similarly, the rows of a row-major two-dimensional array have unit stride, but the columns have stride n if the array has n columns.

The notion of column-major and row-major storage order can be generalized to higher-

dimensional arrays. For example, given a three-dimensional array of size (m,n,p), the row-major storage order corresponds the multi-index order

$$(0,0,0),(0,0,1),\ldots,(0,0,p-1),(0,1,0),(0,1,1),\ldots,(m-1,n-1,p-1)$$

whereas the column-major storage order corresponds to the multi-index order

$$(0,0,0),(1,0,0),\ldots,(m-1,0,0),(0,1,0),(1,1,0),\ldots,(m-1,n-1,p-1).$$

A total of mnp multi-indices are required to index all the elements, and each multi-index can be mapped to a one-dimensional *linear index* ranging from 0 through $mnp-1$. Using the row-major storage order, the multi-index (i,j,k) corresponds to the linear index

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix}^T \begin{pmatrix} pn \\ p \\ 1 \end{pmatrix} = k + jp + ipn,$$

i.e., it is the inner product of the multi-index (i,j,k) and the tuple $(pn,p,1)$. This tuple consists of the *strides* of the array, and these define how the elements of a multidimensional array are arranged in memory. Each stride is the number of unit steps in memory required to reach the next position along the corresponding axis. Similarly, using the column-major storage order, the multi-index (i,j,k) maps to the linear index

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix}^T \begin{pmatrix} 1 \\ m \\ mn \end{pmatrix} = i + jm + kmn,$$

where $(1,m,mn)$ corresponds to the strides of the array. More generally, the strides of a contiguous k-dimensional array of size $(m_1,\ldots,m_k)$ may be represented by a tuple $(s_1,\ldots,s_k)$, the elements of which may be defined recursively as either

$$s_k = 1, \quad s_i = s_{i+1}m_{i+1},\ i = 1,\ldots,k-1,$$

if the row-major storage order is used, or

$$s_1 = 1, \quad s_{i+1} = s_i m_i,\ i = 2,\ldots,k,$$

if the column-major storage order is used. We note that other strides can be used to define *non-contiguous* multidimensional arrays with regularly spaced elements.

6.1 Data structures for two-dimensional arrays in C

Two-dimensional arrays can be represented in many different ways. We will now consider two approaches: the first approach uses index calculations to map a multi-index to a linear index, and the second approach uses an array of pointers to store the location of each row.

Linear indexing

The following data structure represents a two-dimensional array:

```c
enum storage_order {RowMajor, ColMajor};
typedef struct array2d {
    size_t shape[2];
    enum storage_order order;  // RowMajor or ColMajor
    double * val;
} array2d_t;
```

The struct member `order`, which either takes the value `RowMajor` or `ColMajor`, determines how the storage that `val` points to should be interpreted. If `order` is `RowMajor`, then the entry at index (i, j) should be store at `val+i*shape[1]+j`, and if `order` is `ColMajor`, the entry should be stored at `val+i+j*shape[0]`.

The following function can be used to allocate the array data structure and storage for the values:

```c
array2d_t *array2d_alloc(
        const size_t shape[2],
        enum storage_order order) {
  array2d_t *a = malloc(sizeof(*a));
  if (a == NULL) return NULL;
  a->shape[0] = shape[0];
  a->shape[1] = shape[1];
  a->order = order;
  a->val = calloc(shape[0] * shape[1], sizeof(*(a->val)));
  if (a->val == NULL) { free(a); a = NULL; }
  return a;
}
```

The data structure can be deallocated as follows:

```c
void array2d_dealloc(array2d_t *a) {
  if (a) { free(a->val); free(a); }
}
```

Array of arrays

An alternative to the `array2d_t` type defined in the previous example is to construct a two-dimensional array as an array of arrays. This can be done using a so-called Iliffe-vector (pronounced *eye lif*) which is named after the computer scientist John Iliffe. For two-dimensional arrays, this is essentially an array of pointers to one-dimensional arrays.

```c
typedef struct carray2d {
    size_t shape[2];
    double ** val;
} carray2d_t;
```

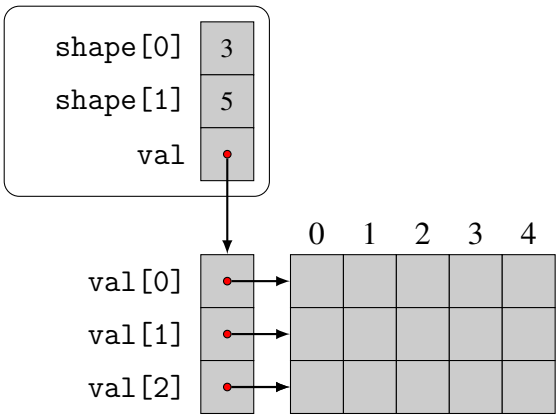
Notice that the member `val` is of the type `double **` which is a "pointer to a pointer to a double". The data structure and storage for the array elements may be allocated with the following function:

```c
carray2d_t *carray2d_alloc(const size_t shape[2]) {
  carray2d_t *a = malloc(sizeof(*a));
  if (a == NULL) { return NULL; }
  a->shape[0] = shape[0];
  a->shape[1] = shape[1];
  a->val = malloc(shape[0] * sizeof(*(a->val)));
  if (a->val == NULL) { free(a); return NULL; }
  a->val[0] = calloc(shape[0] * shape[1], sizeof(*(a->val[0])));
  if (a->val[0] == NULL) { free(a->val); free(a); return NULL; }
  for (size_t k = 1; k < shape[0]; k++)
    a->val[k] = a->val[0] + k*shape[1];
  return a;
}
```

The array may be deallocated using the following function:

```c
void carray2d_dealloc(carray2d_t *a) {
  if (a) {
    free(a->val[0]);
    free(a->val);
    free(a);
  }
}
```

Our allocation routine allocates storage for all rows at once, and then the pointer array pointed to by `val` is updated such that `val[k]` points to the first element of row k which starts at `val[0]+k*shape[1]`. As a consequence, the array is contiguous like automatically allocated two-dimensional arrays in C. If we were to allocate the rows one by one (i.e., calling `calloc` once for every row instead only one time), the rows would still be contiguous but not necessarily adjacent in memory. The next figure illustrates the `carray2d_t` data structure for an array with three rows and five columns.



Note that `val[i]` is a `double *` (i.e., a pointer to a `double`) whereas `val[i][j]` is a `double` (i.e., the element at index (i, j)). This data type avoids the need for index calculations, but unlike the `array2d_t` data structure from the previous example, it requires additional storage for the array of pointers (one for every row in the array).