Mathematical Software Programming (02635)

Lecture 5 — October 2, 2025

Instructor: Martin S. Andersen

Fall 2025

This week

Topics

- Abstract data structures
- Dynamic memory

Learning objectives

- Describe and use **data structures** such as arrays, linked lists, stacks, and queues.
- Choose appropriate data types and **data structures** for a given problem.
- Design, implement, and document a program that solves a mathematical problem.

Abstract data types

An abstract data type is a high-level abstraction: the *behavior* is defined, not the implementation.

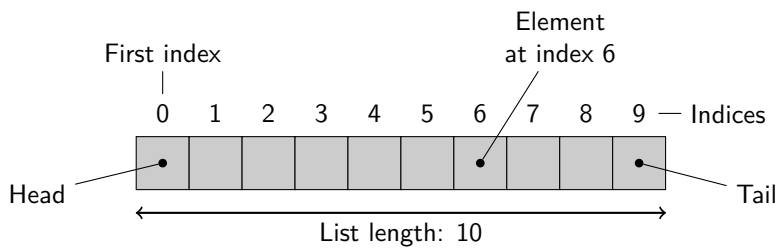
Examples

- sets and lists
- vectors, matrices, tensors
- polynomials
- complex numbers, rational numbers
- points, circles, triangles, rectangles, polygons, etc.

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 sublists
- two lists can be concatenated



Indices can be 0-based or 1-based.

Example

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

- concatenation of L_1 and L_2 (denoted $L_1 \frown L_2$)

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

- splitting L_3 at index 1 (0-based)

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

- accessing the 3rd element of L_2 (i.e., index 2)

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

- inserting the value 8 at index 2 of L_1

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

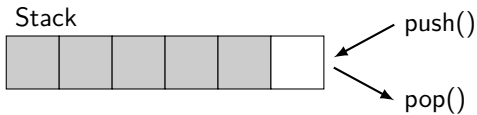
- deleting the head of L_2

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

Stacks and queues

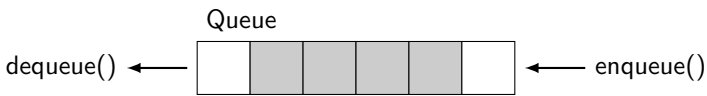
Stack

- a list where elements are inserted and deleted at one end only
- *first-in-last-out* (FILO)



Queue

- a list where elements are inserted at one end and deleted/extracted at the other
- *first-in-first-out* (FIFO)



Structures in C

A *structure* is a type declaration that *groups* a set of variables.

```
struct complex {
    double real;
    double imag;
};
```

```
struct complex a; // a is uninitialized
struct complex b = {1.0,0.0};
struct complex c = {.real=0.0, .imag=1.0};
```

- `a`, `b` and `c` are variables of type `struct complex`
- `real` and `imag` are so-called *members* of the structure
- period (`.`) is the *member access operator* (e.g., `c.real = 5;`)

Example: sum of elements in array

Data structure with array length and pointer to first element:

```
struct array {
    size_t length;
    double *x;
};
```

Sum function:

```
double sum(struct array a) {
    double sum = 0.0;
    for (size_t k=0; k<a.length; k++) sum += a.x[k];
    return sum;
}
```

```
double data[] = {1.0,2.0,3.0};
struct array a = {3, data};
printf("sum = %g\n", sum(a));
```

Nested structures

A struct member can be another structure, an array, etc.

Example

Structure representing a point in $\mathbb{R}^2$:

```
struct point {
    double x,y;
};
```

Structure representing a triangle:

```
struct triangle {
    struct point v1,v2,v3;
};
```

Arrays of structures

```
struct point point_array[10];      // uninitialized array of length 10
point_array[0].x = 1.0;            // init. member x of first element
point_array[0].y = 2.0;            // init. member y of first element
point_array[9].x = 4.0;            // init. member x of last element
point_array[9].y = 7.0;            // init. member y of last element
```

```
struct triangle {
    struct point vertices[3];
};
```

Pointer to a structure

- A pointer to a structure is useful as function input/output
- Use `p->member` to access a member of a structure via a pointer `p`

Example

```
double data[10] = {0.0};
struct array a = {10, data};
struct array * p = &a;
p->x[0] = 1.0;
```

Remark: `p->x` is equivalent to `(*p).x` whereas `*p.x` is equivalent to `*(p.x)`.

Structure as return value

A function can return a structure or a pointer to a structure.

Examples

```
struct point midpoint(struct point a, struct point b) {
    struct point c = {(a.x+b.x)/2.0,(a.y+b.y)/2.0};
    return c;
}
```

```
struct triangle * largest(struct triangle * a, struct triangle * b) {
    return (area(*a) >= area(*b)) ? a : b;
}
```

Never return a pointer to an *automatic* local variable.

Type definitions

Type definitions are used to create an alias for an existing type:

```
typedef <type> <alias>;
```

Example

```
typedef struct point { double x,y; } point_t;
typedef struct triangle { point_t vertices[3]; } triangle_t;
```

`point_t` and `triangle_t` are now aliases for `struct point` and `struct triangle`:

```
point_t a={.x=1.0,.y=-2.0}, b={.x=2.0,.y=8.0}, c={.x=5.0,.y=2.0};
triangle_t T={.vertices={a,b,c}};
```

Dynamic memory allocation

Prototypes (`stdlib.h`)

```
void *malloc(size_t size);
void *calloc(size_t nelements, size_t elementSize);
void *realloc(void *pointer, size_t size);
void free(void *pointer);
```

Allocating storage for an array of length N

```
double *p = malloc(N*sizeof(double));
if (p == NULL) {
    // Code to deal with memory allocation failure ...
}
```

The pointer `p` must be used to *free/deallocate* storage.

Extending dynamically allocated memory

```
size_t N = 100;
double *p = malloc(N*sizeof(double));
if (p == NULL) { /* Code to handle memory allocation failure */ }

...

// Request more memory (2*N)
N *= 2;
double *ptmp = realloc(p, N*sizeof(double));
if (ptmp == NULL) {
    // Code to handle reallocation failure ...
    //   p is still a valid pointer
}
else
    p = ptmp;
```

Freeing/deallocating memory

```c
double *p = malloc(100*sizeof(double));
if (p==NULL) { /* Handle memory allocation failure */ }
...

free(p);
```

Common errors

- Freeing memory twice
- Attempting to free automatically allocated memory (`p` was not returned by `malloc`)
- Using pointer to access memory *after* freeing memory
- Forgetting to free memory (may result in a *memory leak*)

Memory: stack vs heap

Stack (automatic allocation)

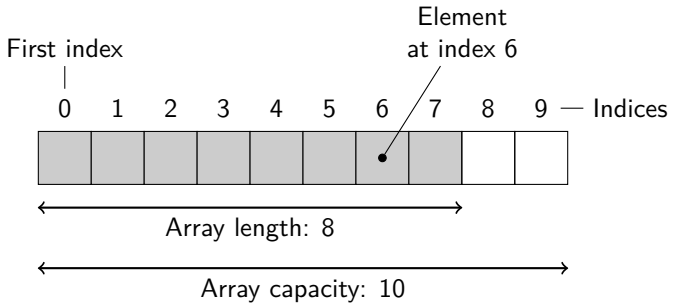
- Memory layout fixed at compile-time (no resizing of variables or arrays)
- Used for local variables (function parameters, temporary variables)
- Fast access (due to contiguous memory and "CPU cache friendliness")
- No manual memory management (automatically allocated/deallocated)
- Limited size (typicall 2-10 MB per thread)

Heap (dynamic allocation)

- Memory allocated at runtime (using `malloc`, etc.)
- Used for dynamic/global data (accessible across functions)
- Manual memory management (must explicitly allocate and free memory)
- Flexible size (only limited by system)
- Slower access (due to pointer indirection and possible fragmentation)
- Risk of memory leaks (if not properly deallocated)

Remark: Static and global variables are stored in the "data segment", not the stack or heap.

Dynamic array




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

Example: allocating/deallocating a dynamic array

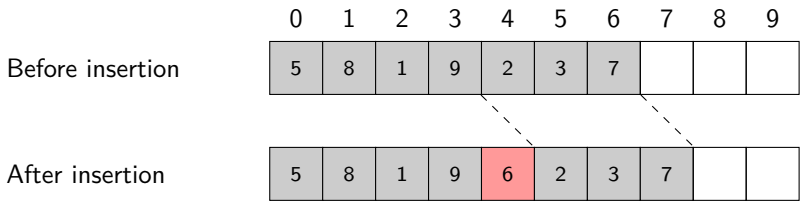
```c
array_t *array_alloc(const size_t capacity) {
  array_t *a = malloc(sizeof(*a));
  if (a == NULL) return NULL;
  a->capacity = (capacity > 0) ? capacity : 1; // Minimum capacity: 1
  a->len = 0;
  a->val = malloc((a->capacity) * sizeof(*(a->val)));
  if (a->val == NULL) { free(a); return NULL; }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

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

Important: `free(a->val)` must precede `free(a)`

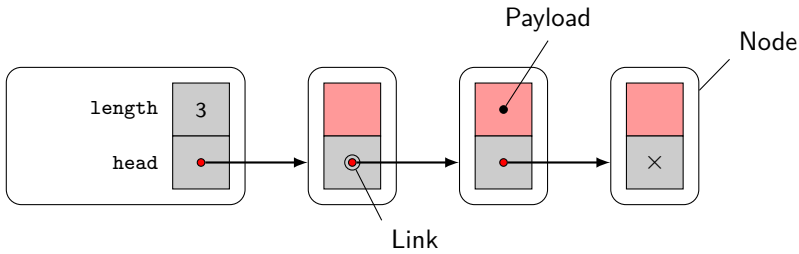
Inserting an element in a dynamic array

Insert “6” at index 4:



- Inserting at index 0 requires that all entries are relocated
- Appending (inserting at index `len`) may involve relocation if length is equal to capacity
- Increasing capacity may involve relocation
  - design choice: increase capacity by constant amount/factor/...

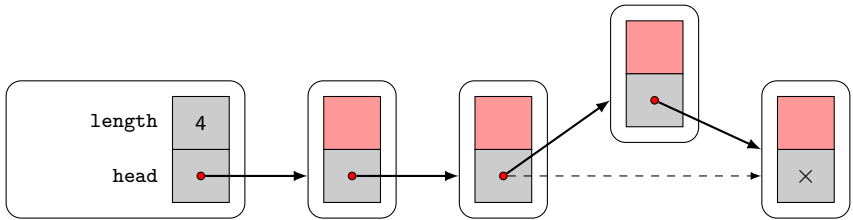
Singly linked list




```
struct snode {
    double x;
    struct snode * next;  };
struct sllist {
    size_t length;
    struct snode * head;  };
```

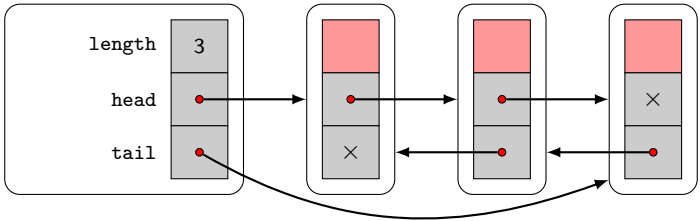
Inserting an element in a singly linked list

Insert node at index 2:



- Inserting a node at index 0 is fast
- Inserting a node at any other index:
  - find preceding node (up to n operations if list has length n)
  - insert new node (does not depend on length)

Doubly linked list




```
struct dnode {
    double x;
    struct dnode * next;
    struct dnode * prev;  };
struct dllist {
    size_t length;
    struct dnode * head;
    struct dnode * tail;  };
```

Implementing a list

Array-based implementation

- cost of finding/accessing an element does not depend on list length n
- worst-case cost of inserting/deleting an element is proportional to n

Linked list implementation

- worst-case cost of finding/accessing an element is proportional to n
- cost of inserting/deleting an element does not depend on n

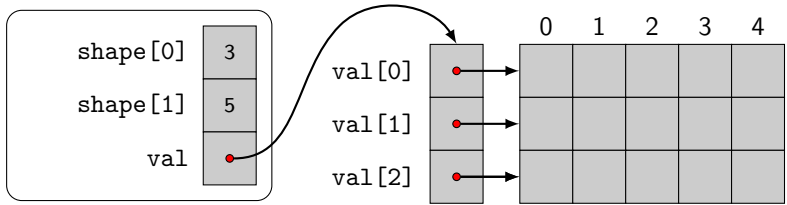
Two-dimensional arrays: linear indexing

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

```c
array2d_t *a = array2d_alloc((size_t []){3,5}, RowMajor);
a->val[i*5+j] = 2.0;    // set element at index (i,j) to 2.0
```

Two-dimensional arrays as an array or arrays

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



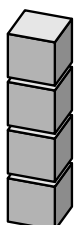

```
carray2d_t * a = carray2d_alloc((size_t []){3,5});
a->val[i][j] = 2.0;    // set element at index (i,j) to 2.0
```

Multidimensional arrays

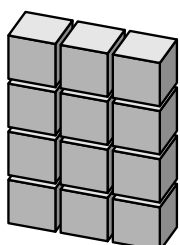
k-dimensional array

- array *shape* is a k-tuple: $(m_1, \ldots, m_k)$
- *strides* tuple $(s_1, \ldots, s_k)$ determines storage order (generalization of row or column major)
- indexing requires a *multi-index* which is a k-tuple of the form $(i_1, \ldots, i_k)$

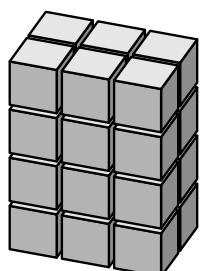
one-dimensional array



two-dimensional array



three-dimensional array



four-dimensional array

MSP Tools

Small, educational C library that is based on the data structures introduced in today's note.

```
CC=gcc
CPPFLAGS=-Imsptools/include
CFLAGS=-Wall -std=c17
LDLIBS=-lm -lmsptools
LDFLAGS=-Lmsptools/lib

.PHONY: all clean
all: msptools myprogram
msptools:
    git clone https://gitlab.gbar.dtu.dk/mskan/msptools.git
    make --directory=msptools
clean:
    -$(RM) myprogram
```

Exercises

- Abstract data types
- Structures and implementation
- Memory allocation

Remember that every `malloc()` call should have a matching `free()` call.