# Useful data structures and their implementations in C

Julia PHAM BA NIEN

April 21, 2024

# Contents

I have decided to show implementation in C, I don't particularly like the Knuth's pseudocode style, and choosing a low level language shows the small memory-related implementations details.
I will use `T` as a type place holder, because data structures often are generics, and I will use a `type<T>` syntax to specify what the underlaying T would be. (even though it isn't truly valid C), also, I may not use struct names when specifying `struct` with its parameters, and will use `structure_name vals ...` to indicate that it's a slight modification of `structure` with fields being `vals ...` (other than the structure specific ones), also I'll use some non standard functions like `swap`, you know what it means, so do as if I defined it.
Data structures are *incredibly* related to algorithms, I'll assume some familiarity with the $O$ notation and analysis of algorithms.
Distinction between data structures are from me, their borders are quite blurry, but I think it still makes sense for the most part.
I won't show any proofs for the time complexities, or anything really.

# 1 What's a data structure

A data structure in a way to structure data.
Though, this answer lacks a lot, because the question isn't the right one. (*checkmate person writing this exact guide*)
I think it's important to see data structure by the view of algorithms, and thus exploring *why* they are.
Data structures are ways to structure data to have *efficients* ways to do the operations you want to perform.
Thus, you should choose your data structures depending on *what* you want to perform, and how often you perform it.
That's why there are so many of them, because there are different use cases.

# 2 Meta structures

Meta structures are what the language offers to represent structures.

## 2.1 Primitives

Primitives are structures which represent *values* and wich looking at its parts doesn't make sense.
For example, `int` is a primitive, though `strings` aren't (because they are meant to see parts of it, its characters).
Even such small types have structure, which have different trade off in speed, memory usage and precision.

## 2.2 Product types

It's the normal way to put multiple values of different types together.

As a structure, we only care when putting only a finite amount of types together, so that in C terms, it's a struct. (In some languages, it's possible to put an infinite amount of value together, those are the case for functions which return type depends of its input in some languages).

## 2.3 Sum types

Sum types are types which are the union of multiple other types.
And they are instanciated by having a variable.

### 2.3.1 Enums

Enums are sum types which have some fixed known to be values.
In C, they are litteraly enums.

### 2.3.2 Tagged unions

Tagged unions are the "true" sum types (other that they are the sum of finite amount of types), where you can have more complexe stuffs than predetermined list of values.
The idea is that you will have a "tag", which will tell you what the structure of the rest is to be read as.
A very simple example is making an int_or_float_t.

```c
enum int_or_float_e {
  INT,
  FLOAT,
};

struct int_or_float_s {
  enum int_or_float_e tag;
  union {float f; int i;} value;
};
```

And you know if you need to access .value.i or .value.f depending on .tag.
It's not as good as with other languages as it's not checked, but it is useful.
A particular one that is quite useful is option, due to its simplicity you can make it without unions nor enums.

```c
struct option_s {
  bool has_value;
  T value;
}
```

Where .value can be left uninitialised or undefined of .has_value is false.

# 3 Container structures

A container structure is one which its goal is to put multiple values of the same type into one structure, so that you can access them, pass them around, etc.

## 3.1 Linked lists

The idea is that each item as a pointer to the next one, unless you're the last value.
Adding a new element is $O(n)$, accessing the i-th element is $O(1+i)$, and also it's not very compact memory wise, and will cache miss a lot.
In general it's not such a great data structure, it is mostly useful when you want an expandable container where its elements are very expansive to copy.
Implementation:

```
typedef struct linked_list_s {
  T value;
  struct linked_list_s *next;
} *linked_list_t;

linked_list_t init_linked_list() {
  return NULL;
}

// 0-based indexing
T get_ith(linked_list_t l, int i) {
  if (l == NULL)
    exit(1); // access out of bound, deal with it as you want
  if (i == 0)
    return l->value;
  return get_ith(l->next, i-1);
}

/* In imperative form:
T get_ith(linked_list_t l, int i) {
  while (l && i) {
    l = l->value;
    --i;
  }
  if (!l)
    exit(1);
  return l->value;
}
*/

linked_list_t append(linked_list l, T val) {
  if (!l) {
    linked_list_t new = (linked_list_t) malloc(sizeof(linked_list_s
    ));
    new->value = val;
    new->next = NULL;
    return new;
  } else {
    l->next = append(l->next, val);
    return l;
  }
}

void free_list(linked_list_t l) {
  if (l) {
    free_list(l->next);
```

```
46      free(l);
47    }
48  }
```

Note that for appending, it is better to have a reference to the last element, so that it becomes an $O(1)$ operation, but then it's not truly a linked list anymore. (also, yes I know that there are ways to make generic linked list, some aren't the best (`void *value; struct linked_list *next;`) and quite good one do some "advanced" C tricks (`https://felipec.github.io/good-taste/parts/1.html`), I don't cover it because it doesn't matter and it's not translatable directly into most other languages.)

## 3.2 Array

An array is a continuous memory blob.
There are some very good part about it, indexing is an $O(1)$ operation, so it's the backbone of a lot of other structures.
Though, it has a huge problem if taken as is, you don't know if your indexing is valid unless you know its size.
There are two main way to have some size indication.

### 3.2.1 Sentinel valued array

The idea is that at the end of your array, you put a value that is supposed to be impossible to get, so once you get there, you know it's the end.
For example, common such values are `'\0'` for strings or $-1$ for positive integer arrays.
Having this value is very useful for the implementation of some algorithms, such as doing a recursive decent parser.
Though, if you want to check your indexing, you have to go through all values to see if you are indexing after the sentinel value, making it an $O(1 + \min(n, i))$ operation.

### 3.2.2 Fat pointers

The idea of a *fat pointer* is that it is a pointer, to which you couple with the size of the array underneath.
Basically, it is

```
1  struct fat_s {
2    T *p;
3    size_t size;
4  };
```

It doesn't require having an impossible value, has $O(1)$ out of bound check and $O(1)$ knowing the length.
A very good feature of it is that it can do $O(1)$ sublististing, and that without any (heap) memory allocation (and not allocating is a good way to not have memory problems).

```c
// 0 based indexing
struct fat_s sublist(struct fat_s of, size_t start, size_t end) {
  if (end > of.size - 1)
    exit(1); // out of bound sublisting
  struct fat_s new;
  new.size = end - start + 1;
  new.p = of.p + start;
  return new;
}
```

Be careful when freeing though, as you have indirect references to the same memory.

### 3.2.3 Vectors

Now, all fun and good, but adding a new element to an array seems *horrible*!
You have to allocate a new bigger array, and copy all your previous elements to your new one, thus being an $O(n)$ operation.
Thanksfully some smart people made a quite good in general data structure to deal with it, the "vector".
The idea is that you want to reallocate as little as possible.
We won't be using the standard idea of the Landau's $O$ notation, we'll consider the "amortized" time, aka. we'll look at how it behaves when adding $n$ element (from empty), not a single one.
Let's analyse some ideas on how to grow the size of the array.
Adding one to the size: the operation is then $\sum_{k=1}^{n} O(k) = O(n^2)$ (because $\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = O(n^2)$).
Now, what about magically increasing the array to the right size ? It is then $\sum_{k=1}^{n} O(1) = O(n)$.
Though, we don't know the size, but at least we know that the operation is $\Omega(n)$, so that's what we'll aim for.
What about increasing by multiplying by a constant greater (or equal) than 2 ? We'll call this constant $c$.
It is then $\sum_{k=1}^{n} O(1) + \sum_{k=0}^{\lfloor \log_c(n) \rfloor} O(c^k) = O(n) + \sum_{k=0}^{\lfloor \log_c(n) \rfloor} = O(n)$
Because $\sum_{k=0}^{\lfloor \log_c(n) \rfloor} c^k = \frac{1-c^{1+\lfloor \log_c(n) \rfloor}}{1-c} = O(c^{1+\lfloor \log_c(n) \rfloor}) = O(c^{\lfloor \log_c(n) \rfloor})) = O(n)$.
Thus it is optimal in term of time comlexity (and exercice: show that it is optimal in term of space complexity), so good enough, we'll use that scheme of allocation.
We'll keep in mind the actual size of the array and the length of it (different here, because you may over allocate).
Implementation-wise it goes as so (here $c = 2$, though litterature suggest that $c = 3$ is better):

```c
struct vector_s {
  T *data;
  size_t len;
  size_t ize;
}

void grow(struct vector_s *t) {
```

```
  size_t new_size = t->size == 0 ? 1 : 2*t->size;
  T *new_data = malloc(new_size * sizeof(T));
  if (t->data) {
    memcpy(new_data, t->data, t->len * sizeof(T));
    free(t->data);
  }
  t->data = new_data;
  t->size = new_size;
}

void add(struct vector_s *t, T el) {
  if (t->len == t->size)
    grow(t);
  t->data[(t->len)++] = el;
}

T get(struct vector_s *t, int n) {
  return t->data[n];
}

T pop(struct vector_s *t) {
  return t->data[--(t->len)];
}

T top(struct vector_s *t) {
  return t->data[(t->len) - 1];
}

struct vector_s init_vector() {
  struct vector_s v;
  v->size = 0;
  v->len = 0;
  v->data = NULL;
  return v;
}

int free_vector(struct vector_s *l) {
  if (t->data)
    free(t->data);
  t->size = 0;
  t->len = 0;
}
```

It is a very common data structure in most of programming languages.

## 3.3 Stacks

A stack is a data structure last in, first out (LIFO), the most common way to implement it is to just use a vector underneath, it trivially accomodate all the stack operations so I won't explain further.

## 3.4 Queues

A queue is a first in, first out (FIFO) structure, a quite common way to implement it is to use *2* vectors underneath.

A basic implementation goes as follow

```
typedef struct queue_s {
  size_t read_position;
  struct vector_s *to_read;
  struct vector_s *to_fill;
} *queue_t;

queue_t init_queue() {
  queue_t q = malloc(sizeof(struct queue_s));
  q->read_position = 0;
  q->to_read = malloc(sizeof(struct vector_s));
  q->to_fill = malloc(sizeof(struct vector_s));

  *(q->to_read) = init_vector();
  *(q->to_fill) = init_vector();

  return q;
}

void push_queue(queue_t q, T element) {
  add(q->to_fill, T);
}

T pop_queue(queue_t q) {
  if (q->to_read->len == 0 && q->to_fill->len == 0)
    exit(1); // no element to pop

  if (q->read_position < q->to_read->len)
    return get(q->to_read, q->read_position++);

  // swap the reader and writter
  q->read_position = 0;
  swap(&q->to_fill, &q->to_read);

  return pop_queue(q);
}

void free_queue(queue_t q) {
  free_vector(q->to_read);
  free_vector(q->to_fill);
  free(q);
}
```

All operations being $O(1)$ amortized.

## 3.5 Hashmaps

There are a lot of different implementation of hashmaps, though, they all come down from the same idea, the *hash*.
It's a function, taking `T` and returning a `size_t`.
The idea is that you first find in which congruency class induced by `hash` you element is, and then do your operations in the class.
Because memory isn't infinite, you store the number of class you have, $n$, and you'll `% n` the class number given by `n`.
So a basic implementation of an hashmap would be:

```
typedef fat<list {
  struct {
    K key;
    V val;
  }
}> hash<K, V>;

void set(hash h, K key, V val) {
  size_t class_num = hash(key) % h.size;
  list {
    K key;
    V val;
  } class = hash.p[class_num];
  while (class->next != NULL) {
    if (class->val.key == key) {
      class->val.val = val;
      return;
    }
  }
  struct {
    K key;
    V val;
    list *next;
  } *new = malloc(sizeof(struct {
    K key;
    V val;
  }));
  new->key = key;
  new->val = val;
  new->next = NULL;
  class->next = new;
}

void get(hash h, K key) {
  size_t class_num = hash(key) % h.size;
  list<
    struct {
     K key;
     V val;
    }
  > class = hash.p[class_num];

  while (class) {
    if (class->key == key)
      return class->value;
    class = class->next;
  }
  exit(1); // asking for non existent key
}
```

### 3.5.1 Hashing

As we've seen, a crucial part of an hashmap is hashing, but how to devise a good hashing function ?
First of all, it depends on the patterns your input will have.

You don't really want your hashing to be predictable, you want it to seem random (while being deterministic) as it lowers the risk of intentional collision (people will have an harder time retro engeneering conflicting keys).
And we want it to seem random under modulo of some number.
By the way it's a good idea to have this "modulo number" be prime, and for similar reasons of unpredictibility, abusing prime numbers for hasing (but a different one).
For example, some hashing ideas:

```
size_t hash_string(char *s) {
  size_t p = 1000000009;
  size_t res = 0;
  size_t r = 1;
  while (*s) {
    res += r * (*s);
    s++;
    r *= p;
  }
  return res;
}

size_t hash_size_t(size_t n) {
  size_t p = 1000000009;
  return pow(n, p); // you have to defined your pow, making it O(
    log(n)) plz
}
```

### 3.5.2 Transposition tables

Sometimes, you want to speedup computation of some redundant tasks, but it'll be too big to memoize it.
For that, you can use a transposition table, it is used a lot to make game bot using some sort of minimax search.
The idea, is that you'll store the last element of the same hash, and erase the previous ones.
This is less costly memory-wise, and the (time) proximity makes it good enough as a *transposition table* for lots of games, thus its name.
The types are as so:

```
typedef fat<option<struct {K key; V val;}>> transposition_table<K,
    V>;

option<V> get(transposition_table_t , K);
void set(transposition_table_t , K, V);
```

### 3.5.3 Perfect hashmaps

Now, there are cases where you hash function is *very* good, so good that you won't ever have a collision.
In that case, you don't need the lists sheniganens, neither do you need to store the keys.

So the type become:

```
typedef fat<V> perfect_hashmap<K, V>;

V get(perfect_hashmap<K, V> h, K key) {
  return h.p[hash(key) % h.size];
}

void set(perfect_hashmap<K, V> h, K key, V val) {
  h.p[hash(key) % h.size] = val;
}
```

### 3.5.4 Sets

Sets are often defined as hashmap.
Basically:

```
typedef hashmap<K, void> set<K>;
```

You can make some optimisations because of void having only one value, and change the get to return a bool.
And if your set is a perfect hashmap, you can use an fat<bool>.

## 3.6 Bitsets

Hashmaps are quite slow, so for some use cases, when the keys are *small* size_t, you can use an integer as a set.
The main use case is being able to do very efficient intersection and merging of sets.
The idea is to abuse the representation of an integer, we'll say that size_t are 64-bits.
We'll look at the bits of a size_t, and say that 0 means that the "element" isn't included and 1 if it is.
The element being a number from 0 to 63.
For example, 11 = 0b0000000000000000000000000000000000000000000000000000000000001011 represent the set $\{0, 1, 3\}$.
Note that you can only store sizeof(size_t) * 8 elements, if you need more you can make some structs.
It is useful to tranlate your definition domain, for example, doing c - 'a' when dealing with latin letters strings.
The basic operations are as follow:

```
// add a new element e to s
s |= ((size_t) 1) << e;

// check if e is in s
0 != s & ((size_t) 1) << e;

// remove an element e to s
s &= ~(((size_t) 1) << e);

// remove an element e to s if it's here, add it otherwise
```

```
s ^= ((size_t) 1) << e;

// union of s and s2
s | s2;

// intersection of s and s2
s & s2;
```

### 3.7 Run length encodings

There are times where the same element is repeated multiple times, it is sometimes useful to know how time the number is repeated.
So instead of storing $[v_0, \ldots, v_n]$, you store $[(v_0, \text{how many time } v_0, \ldots]$
It's a bit clearer by seeing the algorithm:

```
typedef fat<struct {T val; size_t repeated;} rle<T>;

rle<T> init(fat<T> v) {
  if (v.size == 0) {
    rle = rle<T>.init(0);
    return rlel;
  }
  T last = v.p[0];
  size_t num_diff = 1;
  for (size_t i = 1; i < v.size; ++i) {
    num_diff += last != v.p[i];
    last = v.p[i];
  }

  rle<T> r = rle<T>.init(num_diff);
  last = v.p[0];
  size_t num = 1;
  int j = 0;

  for (size_t i = 1; i < v.size; ++i) {
    if (last == v.p[i]) {
      num_diff++;
    } else {
      rle.p[j].val = last;
      rle.p[j].repeated = num;
      num = 1;
      j++;
    }
    last = v.p[i];
  }
}
```

It is also sometimes useful to sort the vector first.

## 4 Graphs

I'll assume quite a lot of familiarity with graphs, if you don't know a term, wikipedia them.

We'll denote the nodes from 0 to $n-1$ (where $n$ is the number of nodes), this choices leads to much simplier and efficient implementations.

## 4.1 General graphs

A graph to which you don't know many structural feature, thus to which you don't optimize its structure much.
I'll assume that between two node, there can be only one edge, it is the case for most useful graphs, and the implementations are much better.
There are a lot of way to represent them, but three of them are the most usefuls (never got a case where incidence matrix was useful).

### 4.1.1 Adjacency lists

You represent your graph with a length $n$ fat pointer $p$ of fat pointers of int (or vector of int if you want to be able to change your graph).
Basically, the node $i$ will direct into all nodes of $p.p[i]$.
If your graph is weighted, you can have a tuples of nodes and weights instead of just nodes.
This is especially good for sparses graphs and for common graphs traversals (dfs and bfs).
For directed graphs, you either specify only out neighbors, or you use two different fat pointers to for in and out.
In "C", an example to type it would be:

```
typedef fat<struct {
  fat<struct {
    size_t node;
    double weight;
  }> in, out;
}> adjacency_list_t;
```

### 4.1.2 Adjacency matrices

We'll consider that non weigthed graphs are weighted graphs with all edge weights put to the value 1.
Basically, we construct an $n \times n$ matrix, and at index $i, j$, you put 0 if there are no edge from $i$ to $j$ and you put the weight of this edge otherwise.
It is possible that in some cases, 0 is a valid weight, so use $-1$, but sometimes it is a valid weight, in that case, use an option<T>.
So in "C", its type would be:

```
typedef fat<option<T>> adjency_matrix_t;
```

It is useable only when $n$ is relatively small (as it is $O(n^2)$ memory wise).
The best trick to know about it is that if $M$ is the adjacency matrix of $G$, a non weighted graph.
Then $M^k_{i,j}$ is the number of path of length $k$ going from $i$ to $j$.

And the operation $M^k$ is an $O(n^2 \log k)$ operation, so it can be very efficient for really big $k$.

### 4.1.3 Actual graph definition

No, not incidence matrix, I never had to use it.
We know that a graph is a pair $(V, E)$ where $V$ are the vertices and $E$ the edges. Because of our numering scheme, we don't have to store $V$ and only need $|V|$. And for the edges, we can store them as tuple (from, to, weight), where from and to are vertices.
Thus a graph is defined as so

```
struct graph_s {
  size_t n;
  fat<struct {
    size_t from, to;
    double weigth;
  }> edges;
};
```

It can be convenient to only keep .edges.
This is to use when you want to iterate through edges by some order, or when $|E|$ is negligeable compared to $|V|$.

## 4.2 Rooted trees

If the tree isn't rooted in an obvious way, not specialising it and using an adjacency list is quite common.

### 4.2.1 General rooted trees

One of the most general way to implement them is as so:

```
struct rooted_tree_s {
  struct rooted_tree_s *parent; // itself or NULL if it is the root
    , depends on what you want
  fat<struct rooted_tree_s> childs;
};
```

It's a bit cumbersome to construct, but it allows for decently easy tree related traversals.
For binary rooted tree with a left, right distinction, it's as so:

```
struct binary_rooted_tree_s {
  struct binary_rooted_tree_s *parent; // itself or NULL if it is
    the root, depends on what you want
  struct binary_rooted_tree_s *left, *right;
};
```

Sometimes you don't specify the parent.

### 4.2.2 Parent oriented trees

Not an actual name.
We'll use the convention that the parent of the root is itself.
What you may think about is doing

```
typedef fat<struct parent_oriented_tree_s {
  struct parent_oriented_tree_s *parent;
}> parent_oriented_tree_t;
```

But we'll once again use the the vertix numerotation is $0..n-1$, we don't need to point to a struct, we can give out the vertice number directly.
And that for all vertices from $0..n-1$.
So a fat pointer does the job perfecly

```
typedef fat<size_t> parent_oriented_tree_t;
```

Where the $i$th element is the parent of $i$.
It's very good for accessing the parent, doesn't use much space, and is quite efficient for what it's designed for, but horrible for a lot of common operations, so definitely not a general purpose data structure.

### 4.2.3 Complete binary trees

Due to its highly regular structure, there are a useful trick to represent complete binary tree which doesn't require to store for each nodes its parents, left and right side.
Instead we can store all of them in a fat pointer, and do some indexing tricks.
We'll put the root at the 0-th index, and we'll put the left node of the $i$-th index at index $2i+1$ and the right at index $2i+2$.
To see why it'd work (why it uses the indexes from 0 to $n-1$, without overlap), it's easier to reason in base 1.
So our rule become root at index 1 and left and right at $2i$ and $2i+1$ respectively.
The no overlap is obvious because composing some $\times 2$ and some $\times 2+1$ and applying it to 1 is akin to using Hörner polynomial decomposition, it creates a unique representation of an integer strictly greater than 0.
And we see that the left part have smaller index than the right one, thus the indexes are 1 to $n$.
Now to translate back to our 0 based index, we put everything one case left.
For practical reasons, we uses the indices for the methods.
So an implementation of such would be:

```
typedef fat<T> perfect<T>;

T get(perfect<T> p, size_t i) {
  return p.p[i];
}

size_t parent(size_t i) {
  return (i-1) / 2; // note that "-1/2" is 0 in C
}

```

```
size_t left(size_t i) {
  return 2*i + 1;
}

size_t right(size_t i) {
  return 2*i+2;
}
```

Note that it can be used for non perfect ordered binary trees as well, by using an option<T> instead of T, though the memory usage is polynomial with the tree height so it's not advisable for very badly balanced trees.
It can also be used for ternary, even $n$-ary perfect trees like so:

```
typedef fat<T> perfect<T, n>; // n is the number of possible
    children

T get(perfect<T, n> p, size_t i) {
  return p.p[i];
}

size_t parent(size_t i) {
  return (i-1) / n; // same n as before
}

// the num-th child of i, with 0 based indexing
size_t nth(size_t i, size_t num) {
  return n*i + num + 1;
}
```

### 4.2.4 Heaps

A heap is a complete binary tree where each element has a value less than its children's.
So, of course, we'll use perfect<T>, but with a vector<T> instead of a fat<T> to be able to add and remove elements.
The main goal is being able to find the minimum, adding an element and removing an element efficiently.
Finding the min is very easy:

```
void min(heap<T> h) {
  return h.p[0];
}
```

Now, removing and adding elements is a bit harder, there are different ways to implement those but I'll introduce one which in my opinion, is easy to understand and to extend, the "swift up", "swift down" approach.
The "swift up" and when you have an almost heap, it is an heap for indexes from 0 to $i-1$ included, and you want to make it an heap from 0 to $i$ by bubbleing up the element.
They'll both return the new index of the previously $i$-th element.

```
// n <= h.size
size_t swift_up(heap<T> h, size_t i, size_t n) {
  while (1) {
```

```
    size_t p = parent(i);
    size_t l = left(p);
    size_t r = right(p);

    size_t smallest = p;

    if (l < n && get(h, l) < get(h, p))
      smallest = l;

    if (r < n && get(h, r) < get(h, p))
      smallest = r;

    if (p == smallest)
      return i;

    swap(&h.p[p], h.p[smallest]);

    i = p;
  }
}
```

And swift down is when you have an almost heap, an heap from index $i+1$ to $n-1$ and you want to make it an heap from $i$ to $n-1$.

```
// n <= h.size
size_t swift_down(heap<T> h, size_t i, size_t n) {
  while (1) {
    size_t l = left(i);
    size_t r = right(i);

    size_t smallest = i;

    if (l < n && get(h, l) < get(h, p))
      smallest = l;

    if (r < n && get(h, r) < get(h, p))
      smallest = r;

    if (p == smallest)
      return i;

    swap(&h.p[p], h.p[smallest]);

    i = smallest;
  }
}
```

It is very similar and that mean you can easily copy-paste your swift_up and modify some value very quickly.
To add an element, you can put it last, and swift it up.

```
void add(heap<T> h, T el) {
  size_t n = h.size;
  add_vector(&h, el);
  swift_up(h, n, h.size);
}
```

To delete an element, you put the last element of the vector at its place, pop the vector, then heap the whole again.

```
void pop(heap<T> h, size_t i) {
  h.p[i] = h.p[h.size-1];
  pop(h);
  swift_down(h, swift_up(h, i, h.size), h.size);
}
```

Note that it can be used to create an easy $O(n \log n)$ in place sorting algorithm, heap sort:

```
void heapsort(fat<T> p) {
  heap<T> h = p;
  size_t n = p.size;

  // make it an actual heap
  for (int i = 0; i < n; ++i)
    swift_up(h, i, h.size);

  // sort it in reverse
  for (int i = n-1; i; --i) {
    swap(&h.p[i], &h.p[0]);

    swift_down(h, 0, i);
  }

  // reverse it
  for (int i = 0; i < n/2; ++i)
    swap(&h.p[i], &h.p[n-i-1]);
}
```

There are actually a l of different kind of heaps, the other "important" one is the Fibonnacci's heap (used for better time complexity for Prim's and Dijkstra's algorithms), but its implementation is quite complicated.

# 5 Priority Queues

Priority queues are data structures which represent a queue where the first element out is the smallest one.
An easy way to implement it is by using an heap, different kind of heap makes different cases where the queue is good at.

# 6 Disjoint set unions

Imagine, you want to making a civilisation-like game.
People are in guilds (of just themselves if they don't have friends), only one.
And they can decide to merge guilds.
And quite often, players will ask if they are in the same guild as someone else because otherwise they'll start a war.

This is the kind of problems this datastructure will solve, the two main operations are same(dsu, person1, person2) and merge(dsu, person1, person2) (with merge returning if it indeed had to do something to merge).
I'll present multiple partial solutions as it's better for learning in my opinion.
Let $n$ be the number of persons.
One of the easiest way to modilise it is with a bipartite $n + n$ graph, where on the left side there are the persons and on the right the guilds, and a connection if that person is in that guild, we'll say that by default, persons are matched with guild of the same "level".
Making for $O(1)$ same and $O(n)$ merging.

```
typedef fat<size_t> dsu;

dsu init_dsu(size_t n) {
  dsu d = init_fat<size_t>(n);
  while (--n)
    d.p[n] = n;
  return d;
}

bool same(dsu d, size_t a, size_t b) {
  return d.p[a] == d.p[b];
}

bool merge(dsu d, size_t a, size_t b) {
  if same(d, a, b)
    return false;
  size_t guild_of_b = d.p[b];
  for (int i = 0; i < d.size; ++i)
    if (d.p[i] == guild_of_b)
      d.p[i] = d.p[a];
  return true;
}
```

You may think that iterating through everything is wasteful and you'd want to iterate only the one which are on the same guild as b.
It is indeed better in average, but it has the same mean and worstcase complexity, so we won't go that route.
Now, there are an obvious bijection between the guilds and the persons, so let's consider that a connect mean "has the same guild as this person", we want a parent oriented tree/graph !
We'll create a function find which find the root of the tree, the "representent" of the guild.

```
size_t find(dsu d, size_t i) {
  while (i != d.p[i])
    i = d.p[i];
  return i;
}

bool same(dsu d, size_t a, size_t b) {
  return find(d, a) == find(d, b);
}

```

```
bool merge(dsu d, size_t a, size_t b) {
  size_t fa = find(d, a);
  size_t fb = find(d, b);
  if (fa == fb)
    return false;
  d.p[fb] = fa;
  return true;
}
```

And if we look at the time complexity, it's now $O(n)$ for all operations... that's not better, but we have some tricks up our sleeves.
We can remark that the operation find is $O(n)$ because we allow our tree to we very deep, unnecessary deep.
In a perfect world, you'd want it to be a complete tree or at least complete until the leaf level, that make it $O(\log n)$ and all operations will become $O(\log n)$ as a consequence.
To have this kind of results, there are two common heuristics, the "weight" and "depth" ones.
The weight one is storing how many nodes are attached to it, and depth one is storing the height of the tree of nodes attached to it.
They give out the same time complexity ($O(\log n)$ for find and thus for all operations), but I prefer the "weight" one because it is very slightly faster and will play nicely for other operations, though the "depth" one is more often used.

```
struct dsu {
  size_t n;
  size_t *succ; // array of size n
  size_t *w; // array of size n
}

struct dsu init_dsu(size_t n) {
  struct dsu d = {.n = n, .succ = malloc(n*sizeof(size_t)), .w =
    malloc(n*sizeof(size_t))};
  for (int i = 0; i < n; ++i) {
    dsu.w[i] = 1;
    dsu.succ[i] = i;
  }
}

size_t find(struct dsu d, size_t i) {
  while (i != d.succ[i])
    i = d.succ[i];
  return i;
}

bool same(struct dsu d, size_t a, size_t b) {
  return find(d, a) == find(d, b);
}

bool merge(struct dsu d, size_t a, size_t b) {
  size_t fa = find(d, a);
  size_t fb = find(d, b);

  if (fa == fb)
    return false;
```

```
31
32    if (d.w[fb] > d.w[fa])
33      swap(&fa, &fb);
34
35    d.succ[fb] = fa;
36    d.w[fa] += d.w[fb];
37    return true;
38  }
```

Now, suppose you call find(d, 3) a lot of time without merging in between, it feels a bit wasteful to do an $O(\log n)$ operation each time when you could memoize it.
And that is the idea behind the "path compression" optimisation, when you are finding the representent of a node, you put it as its successor directly, making it much more efficient in amortized time.

```
1  size_t _find(struct dsu d, size_t i) {
2    while (i != d.succ[i])
3      i = d.succ[i];
4    return i;
5  }
6
7  size_t find(struct dsu d, size_t i) {
8    size_t repr = _find(d, i);
9    while (i != repr) {
10     size_t next = d.succ[i];
11     d.succ[i] = repr;
12     i = next;
13   }
14   return repr;
15 }
```

Note on how we don't care about the weights of nodes, also it have a quite nice non tail-call recursive way to write it, but it's not tail call recursive so I prefer this way as it's constant in memory usage.
All operations are now $O(\alpha(n))$ amortized, where $\alpha$ is the inverse function of the Ackermann's, so it grows *very* slowly, and for all reasonable inputs it's less than 4.
Note though that there are not only just one easy to backtrack operation done when doing a find, so to use it for backtracking-heavy algorithms (like negamax), you may decide to not implement the path compression.
Now, let's add a functionality to our game, a player can decide to quit its guild and join its own.
An idea is that if nobody is connected to them, they can just take themselfs as parent, otherwise, if they are their own guild representent, they need to take its ownership to someone else, and use the path compression to put everyone else to the other representent to be alone.
Though this require to have an accurate represententation of weights for non root nodes.

```
1  void flatten(struct dsu d) {
2    for (int i = 0; i < d.size; ++i)
3      find(d, i);
```

```
}

size_t find(struct dsu d, size_t i) {
  size_t repr = _find(d, i);
  size_t prev_weight = d.w[i];
  while (i != repr) {
    size_t next = d.succ[i];
    d.succ[i] = repr;
    i = next;
    if (i != repr) {
      size_t tmp = d.w[i];
      d.w[i] -= prev_weight;
      prev_weight = tmp;
    }
  }
  return repr;
}

// O(n)
bool disconnect(struct dsu d, size_t i) {
  if (d.w[i] == 1) {
    if (d.succ[i] == i)
      return false;
    --d.w[d.succ[i]];
    d.succ[i] = i;
    return;
  }

  flatten(d);

  if (d.w[i] == 1) {
    if (d.succ[i] == i)
      return false;
    --d.w[d.succ[i]];
    d.succ[i] = i;
    return;
  }

  // i is the representent of a guild with >= 2 members
  for (int k = 0; k < d.n; ++k) {
    if (i == k)
      continue;
    if (d.succ[k] == i) {
      d.succ[i] = k;
      d.succ[k] = k;
      d.w[k] = d.w[i]--;
      break;
    }
  }
  flatten(d);
  --d.w[d.succ[i]];
  d.succ[i] = i;
}
```

# 7 Range queries

Imagine you want a binary operators $a \top b$ and an array $v$, and you want to be able to quickly know $\overset{j}{\underset{k=i}{\top}} v_k$ for some $i \leq j$ not given (those will be the queries). You could store all the queries ahead of time, having a time complexity of $O(n^2)$ or you can use some of the followings data structures

## 7.1 Prefix sums

If $\top$ is commutative, divisible and that the values wouldn't change, then we can use a prefix "sum" (the term is as so because you usually use it for addition):

```
typedef fat<T> prefix<T>;

prefix<T> init(fat<T> f) {
  prefix<T> = copy(f);
  if (p.size <= 1)
    return p;

  T s = p.p[0];

  for (int i = 1; i < p.size; ++i)
    p.p[i] = s = op(s, p.p[i]);

  return p;
}

T query(prefix<T> p, size_t i, size_t j) {
  if (i > j)
    exit(1); // plz handle errors correctly, can't return 0 because
     op isn't unitful
  if (!i)
    return p.p[j];
  return div(p.p[j], p.p[i-1]);
}
```

With $O(1)$ query, $O(n)$ initialisation, and $O(n)$ updating (just initialise it again).

## 7.2 Segment trees

We'll now suppose that $\top$ is associative and unitful (if it isn't, we can make it so by using an option<T> and consider the none value as being the unit), so we'll consider $\top$ to be unitful.
What you'd do is having a complete binary tree with last layer be your vector (and unit element to fill it up), and the value of a non-leaf node $n$ is node_left($n$) $\top$ node_right($n$).
I'll consider that the unit element of $\top$ is called unit.
In code it becomes:

```
typedef struct {
  size_t depth;
  T *p; // of size 2^(depth+1) - 1
```

```
} segment_tree<T>;

segment_tree<T> init(fat<T> f) {
  size_t n = f.size;
  size_t d = 0;
  size_t size = 1;
  size_t pow2 = 2;

  while (size < n) {
    d++;
    size += pow2;
    if (size < n)
      pow2 *= 2;
  }

  segment_tree<T> st;
  st.depth = depth;
  st.p = malloc(size * sizeof(T));
  for (int i = size - pow; i < size - pow + n; ++i)
    st.p[i] = f.p[i - size + pow];
  for (int i = size-pow+n; i < size; ++i)
    st.p[i] = unit;
  for (int i = size-pow-1; i >= 0; --i)
    st.p[i] = op(st.p[left(i)], st.p[right(i)]);
}

T _query(T *p, size_t i, size_t diameter, size_t l, size_t r) {
  if (r - l + 1 == diam)
    return p[i];
  if (diameter == 1)
    return unit;

  size_t mid = diameter/2;
  if (r < mid)
    return _query(p, left(i), mid, l, r);
  if (l >= mid)
    return _query(p, right(i), mid, l-mid, r-mid);
  return op(_query(p, left(i), mid, l, mid),  _query(p, right(i),
    mid, 0, r-mid))
}

T query(segment_tree<T> st, size_t l, size_t r) {
  size_t diam = 1ULL << st.depth;
  if (l < 0 || r >= diam)
    exit(1);
  return _query(st.p, 0, 1ULL << st.depth, size_t l, size_t r);
}

void update(segment_tree<T> st, size_t i, T val) {
  size_t diam = 1ULL << st.depth;
  size_t size = (1ULL << (st.depth+1))-1;
  i = size - diam + i;
  st.p[i] = val;
  while (i != 0) {
    i = parent(i);
    st.p[i] = op(st.p[left(i)], st.p[right(i)]);
  }
```

```
}
```

With $O(\log n)$ query, $O(n)$ initialisation, $O(\log n)$ updating.

## 7.3 Fenwick/BIT trees

We will now suppose that $\top$ forms a group.
There are actually two common types of Fenwick's (or Binary Interval Tree) trees. (yes, I repeat myself with the "tree tree"), the 1-based index and the 0-based index.
The 1-based index is more common, but I find that letting the index 0 unused is quite sad, so we'll speak about the 0-based index version.
The idea is that you'll use a fat<T> of size $n$, and the index $i$ is sum of values at indices from $i\&(i+1)$ (changing all leading 1 in the based 2 representation into 0s) to $i$ included.
And thus, when updating a value of index $i$, you have to modify all indices $j$ with the property: $j\&(j+1) \leq i \leq j$.
To find, you can start with $i$ and flip the last 0 into an 1 for each iteration, which is the operation $j|(j+1)$ and you stop once $j$ will out of bound.
With those information actually writing the data structure becomes doable:

```
typedef fat<T> fenwick<T>;

T op_to(fenwick<T> fw, size_t to) {
  T res = unit;
  while (to >= 0) {
    res = op(fw.p[to], res);
    to = (to & (to + 1)) - 1;
  }
  return res;
}

T query(fenwick<T> fw, size_t from, size_t to) {
  return div(op_to(fw, to), op_to(fw, from - 1));
}

void op_update(fenwick<T> fw, size_t i, T val) {
  while (i < fw.size) {
    fw.p[i] = op(fw.p[i], val);
    i = i | (i+1);
  }
}

void update(fenwick<T> fw, size_t i, T val) {
  T cur = query(fw, i-1, i);
  op_update(fw, i, div(val, cur));
}

fenwick<T> init(fat<T> f) {
  fenwick<T> fw = fat<T>.init(f.size);
  for (size_t i = 0; i < f.size; ++i)
    fw[i] = unit;
  for (size_t i = 0; i < f.size; ++i)
    op_update(fw, i, f[i]);
```

```
  return fw;
}
```

Though this initialisation is $O(n \log n)$, not the best if $\top$ is also abelian, as you can do it in $O(n)$ as so:

```
fenwick<T> init(fat<T> f) {
  fenwick<T> fw = fat<T>.init(f.size);
  for (size_t i = 0; i < f.size; ++i)
    fw[i] = unit;
  for (size_t i = 0; i < f.size; ++i) {
    fw[i] = op(fw[i], f[i]);
    size_t next = i | (i+1);
    if (next < f.size)
      fw[next] = op(fw[next], fw[i]);
  }
  return fw;
}
```

Giving similar time complexity of a segment tree but with better constants.

# 8 Memory management

## 8.1 Safe guards

I'll call by "safe guards" data structures which doesn't *do* anything news, but prevent doing *bad things*, such as freeing memory still in use, or having race conditions.
The second kind are built-in the CPU, but it still can be useful to know how it'd work.

### 8.1.1 Rcs

Sometimes, it's very hard to track the lifetimes of your objects, especially when you're starting to pass around views and/or allocating in a multi-threaded environment.
Wouldn't it be so simple to just call free and if you don't have any other view of the memory, it'll free and and otherwise do "nothing" ?
Well, that's what reference counting (Rc) is for.
The idea is that you count how many references to that memory you have, and when you free, you remove a reference, and if that reference count goes to 0, you actually free your memory.

```
typedef struct {T v; size_t count} *Rc<T>;

Rc<T> init_rc(T val) {
  Rc r = malloc(sizeof(*(Rc)0)); // some compilers may really not
    like it
  r->v =  val;
  r->count = 1;
  return r;
}

```

```
// to call each time you're creating a new reference
Rc<T> copy(Rc<T> r) {
  r->count++;
  return r;
}

// to call each time a reference is destroyed
Rc<T> free_rc(Rc<T> r) {
  r->count--;
  if (r->count == 0) {
    free(r->val);
    free(r);
  }
}
```

Note that Rc is by itself a pointer instead of a struct to be able to modify the count accross references.
Be careful to not do circular referencing of Rcs, you won't be able to free them out.
But it's impossible to do so in languages without mutation, so Rc-based garbage collection is a viable strat for those.

### 8.1.2 Mutexes

There are a lot of data structures for thread safety, though the most common one is the mutex.
The idea is that along your `T`, you put a variable to lock the reading/writing when you need to access it yourself.
You could use a `bool` for the locking variable, but it itself is prone to race condition, so you have to look what your language's thread safe locking type is.
In C it is `pthread_mutex_t`.
In code you'd do:

```
typedef struct {T val; pthread_mutex_t mutex;} mutex<T>;

mutex<T> init(T val) {
  mutex<T> m;
  m.val = m;
  pthread_mutex_create(&m.mutex, NULL);
  return val;
}

void free_mutex(mutex<T> m) {
  pthread_mutex_destroy(&m.mutex);
}

void lock(mutex<T> m) {pthread_mutex_lock(m.mutex);}
void unlock(mutex<T> m) {pthread_mutex_unlock(m.mutex);}
```

## 8.2 Allocators

Have you ever wondered where your memory comes frmo and what were the choices ?

Well, here we go.
Because a lot of our allocators will take others as an argument, I'll define a small allocator interface

```
typedef struct allocator_s {
  void (*free_memory)(struct allocator_s *self, void *addr, size_t
    size); // yeah, no check if that works
  void *(*alloc_memory)(struct allocator_s *self, size_t size);
} allocator;
```

And we'll demand that the first two fields of any allocators we'll use have those two fields in this order at first before any other potential field.

### 8.2.1 Page allocators

The user side and kernel side is a bit different as users have the already made allocator from the kernel.
We'll call this allocator a page_allocator.
It's not really a data structure per se, it's merely a general purpose allocator (kernel code) backed by a fix buffer allocator (the computer memory) (those term will be explained later).
Though I decided to explain it first as it is how you can create most of your own allocators.
Your OS gives your two several (virtual) memory spaces (so don't forget to thank it), there are two which will interest us, the stack and the heap.
To get stack space, you can use strpbrk, it is useful and you can use to allocate, but we'll look at the heap implementations (as they have less constraints in term of scoping).
To get a valid pointer to some heap memory (without malloc), the OS way (in the C stdlib) is the mmap method and the munmap to free.
The type signature:

```
void *mmap(void addr[.length], size_t length, int prot, int flags,
    int fd, off_t offset);
int munmap(void addr[.length], size_t length);
```

You can use mmap to map file memory, the fd argument, but here we'll use NULL to indicate that we're using a new heap.
For speed reasons, it'll usually give you out a memory address multiple of the OS page size, and it'll give you multiple of your page size for acceptable read/write location.
In most systems, it is 4096, so we'll make it extra clear in our implementation.

```
#define PAGE_SIZE 4096

typedef allocator page_allocator; // doesn't need any new field

size_t to_page_size(size_t n) {
  return PAGE_SIZE*((n/PAGE_SIZE) + (n % PAGE_SIZE != 0));
}

void free_memory_page(allocator *self, void *addr, size_t size) {
```

```
  size = to_page_size(size);
    return NULL;
  munmap(addr, size);
}

void *alloc_memory_page(allocator *self, void *addr, size_t size) {
  size = to_page_size(size).

  // it's not defined what mmap should do for those values
  // so I'd just return nonsense
  if (size <= 0)
  return mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_ANONYMOUS |
    MAP_PRIVATE, -1, 0);
}

page_allocator init_page_allocator(void) {
  page_allocator page;
  page.free_memory = &free_memory_page;
  page.alloc_memory = &alloc_memory_page;
  return page;
}
```

Read the man pages for the flags and prot of mmap.

### 8.2.2 Fix buffer allocators

A huge problem of page allocators is that they do a system call for each allocations, and also they waste a lot of memory (for example, if you need 3 bytes, it'll give you 4096 instead).
What we can do is to allocate a bigger memory space, and then manage in that space what we'll allocate in there.
So here is the FBA, you have a memory address, you cannot free by the way (sad time) and you keep track of what space you used or you haven't used with an offset, what's left of the offset is used, and what's right isn't.

```
typedef struct {
  // layout requirements
  void (*free_memory)(struct allocator_s *self, void *addr, size_t
    size);
  void *(*alloc_memory)(struct allocator_s *self, size_t size);

  char *buffer;
  size_t size;
  size_t head;
} FBA;

void free_memory_FBA(allocator *self, void *addr, size_t size) {
  return; // yeah, no op
}

void *alloc_memory_FBA(allocator *self, size_t size) {
  char *current = self->buffer + self->head;
  self->buffer.head += size;
  // you can check if self->head > self->size, in that case it'll
    buffer overflow
  return current;
```

```
}

FBA init_FBA(allocator *altor, size_t size) {
  FBA fba;
  fba.free_memory = &free_memory_FBA;
  fba.alloc_memory = &alloc_memory_FBA;
  fba.buffer = altor.alloc_memory(size);
  fba.size = size;
  fba.head = 0;
  return fba;
}

void free_fba(allocator *altor, FBA *fba) {
  altor.free_memory(fba->buffer);
}
```

### 8.2.3 Arena allocators

ARENA = List<FBA> One problem with the previous way is that if we don't have enough memory to allocate our new object, we're just screwed, and that's bad, a way to circumway that is to allocate a new FBA if we don't have enough space anymore.
And let's put those in a linked list (putting those in a vector wouldn't work because you can't copy memory addresses, only their values, that's one of the cases where using a vector isn't possible while using a linked list is).
In code:

```
typedef struct arena_node_s {
  char *buffer;
  size_t size;
  size_t head;
  arena_node_s *next;
} arena_node;

typedef {
  // layout requirements
  void (*free_memory)(struct allocator_s *self, void *addr, size_t
    size);
  void *(*alloc_memory)(struct allocator_s *self, size_t size);

  allocator *altor;
  arena_node *node;
} arena;

arena_node *init_arena_node(allocator *altor, size_t size) {
  arena_node *n = (arena_node *) altor->alloc_memory(sizeof(
    arena_node));
  n->buffer = (char *) altor->alloc_memory(size);
  n->size = size;
  n->head = 0;
  n->next = NULL;
  return n;
}

void free_memory_arena(void *addr, size_t size) {
```

```
27   return;
28 }
29
30 void *alloc_memory_node_arena(allocator *altor, arena_node *node,
       size_t size) {
31   if (node->head + size <= node->size) {
32     void *rv = node->head+node->buffer;
33     node->head += size;
34     return rv;
35   }
36   if (!node->next)
37     node->next = init_arena_node(max(size, node->size));
38   return alloc_memory_node_arena(altor, node->next, size);
39 }
40
41 void *alloc_memory_arena(allocator *altor, size_t size) {
42   arena *a = (arena *)(void *)altor;
43   return alloc_memory_node_arena(a->altor, &a->node, size);
44 }
45
46 arena *init_arena(allocator *altor, size_t size) {
47   arena a;
48
49   a->free_memory = &free_memory_arena;
50   a->alloc_memory = &alloc_memory_arena;
51
52   a.allocator = altor;
53   a.node = init_arena_node(allocator *altor, size_t size);
54
55   return a;
56 }
57
58 void arena_node_free(allocator *altor, arena_node *node) {
59   if (node->next)
60     arena_node_free(altor, node->next);
61   altor->free_memory(node->buffer);
62   altor->free_memory(node);
63 }
64
65 void arena_free(arena *a) {
66   arena_node_free(a->altor, a->node);
67 }
```

A good thing about it is that you can allocate all the memory you want, and then deallocate everything at once, without having to care the slightliest about lifetimes.
Though like before, we cannot really free stuffs.
It's also quite common to create a function reset which won't deallocate your arena, but will invalidate all its memory, so that you can use it back again without having to allocate again.

### 8.2.4 General purpose allocators

A lot of allocators can be used as "general purpose".
The most common ones are the "slabs" and "buddy" allocators.

They are a bit too complexe for me to explain them clearly and make a good implementation of them (implementations are multiple hundreds of lines long), so I'll let you search online to learn how to do so.

# 9 After words

Thanks for reading, I hope you understand stuffs, and now feel much more competant for writing your own, personal data structures when you'll have a need to.
My goal in this guide was to show a bit the thought process on how to come up with those, though for brievety reasons, I didn't go in depth into how to discover all of them.
If you liked it, try reading others of my guides/articles/blogs/papers on my site: `https://juliapbn.pages.dev`