

---

## C Data Structures and Algorithms

A library of generic intrusive data structures and algorithms in ANSI C

- Background
- Examples
- Installation
- Running Tests
- Contributing

### Background

This repository is an ongoing project of implementing generic *intrusive* data structures and algorithms in ANSI C. I find I often use the same constructs that require a lot of boilerplate code, so I made this repository to both organize these constructs and allow easy integration of these constructs into existing projects.

My design philosophy with this repository is to offer a minimalistic API for each data structure, containing all tools needed to easily build more complex/niche constructs. Portability is top priority, so only ANSI C is used, and no header/source pair relies on another header/source pair. Furthermore, to promote being used in embedded systems, iterative algorithms are used exclusively over recursive algorithms, and each data structure is intrusive (for better memory locality).

Feel free to use these data structures and algorithms in your own way. The code is licensed under the ISC license (a simplified version of the BSD license that is functionally identical); thus, it may legitimately be reused in any project, whether Proprietary or Open Source.

### Examples

Each header is heavily documented and should hopefully answer all questions regarding functionality. An in-depth explanation on how each data structure operates is located at the top of the header file.

```
List
1 // Define your struct somewhere.
2 struct Object {
3     int some_value;
4     ...
5
6     // Don't forget to embed the ListNode!
7     ListNode node;
8 };
9
```

---

```

10 ...
11
12 // Create some Object variables.
13 struct Object obj1, obj2;
14 obj1.some_value = 1;
15 obj2.some_value = 2;
16
17 // Create your List.
18 List my_list;
19 list_init(&my_list);
20
21 // Populate your List. Notice how the API abstracts itself and only
    cares about the ListNode.
22 list_insert_back(&my_list, &obj1.node);
23 list_insert_back(&my_list, &obj2.node);
24
25 // Let's see what is stored in the List:
26 int i = 1;
27 ListNode *n;
28 list_for_each(n, &my_list) {
29     if (i == 1) {
30         assert(n == &obj1.node);
31     } else if (i == 2){
32         assert(n == &obj2.node);
33     }
34
35     ++i;
36 }
37
38 ...
39
40 // Let's get the ListNode at the front of the List.
41 ListNode *front_node_ptr = list_front(&my_list);
42
43 // Getting the "node" member from an Object variable is easy: ("obj1.
    node"),
44 // but how do you get the Object variable when you only have the "node"
    member?
45 // Solution: the macro "list_entry":
46 struct Object *obj_ptr = list_entry(front_node_ptr, struct Object, node
    );
47 assert(obj_ptr == &obj1);

```

## **RBTree**

```

1 // Define your struct somewhere.
2 struct Object {
3     int key;
4     ...
5
6     // Don't forget to embed the RBTreeNode!

```

---

```

7     RBTreeNode node;
8 };
9
10 ...
11
12 // You must define a compare function which compares a key with the key
    of a RBTreeNode.
13 int compare(const void *some_key, const RBTreeNode *some_node) {
14     return *(const int*)some_key - rbtree_entry(some_node, struct
        Object, node)->key;
15 }
16
17 // You can OPTIONALLY define a collide function which handles key
    collisions. When a key
18 // collision happens, no matter what, the old RBTreeNode will be
    replaced by the new
19 // RBTreeNode. If this function is defined, then it will be called
    after the old
20 // RBTreeNode is replaced in the RBTree. This function is great if you
    need to free up
21 // resources in the Object variable pertaining to the discarded
    RBTreeNode. This function
22 // is also great for enabling multi-key functionality in the RBTree.
23 void collide(const RBTreeNode *old_node, const RBTreeNode *new_node,
    void *auxiliary_data) {
24     // When a RBTree is created, we can give it auxiliary data to hold
        onto. This data is then
25     // passed to this function for usage. This data, for example, could
        be a memory pool struct
26     // that is needed to free up the resources used by the Object
        variable pertaining to the
27     // old_node parameter.
28
29     ...
30 }
31
32 ...
33
34 // Create some Object variables.
35 struct Object obj1, obj2;
36 obj1.key = 1;
37 obj2.key = 2;
38
39 // Create you RBTree. In this case we don't have any auxiliary data
    that needs to be used
40 // in the collide function, so we just pass NULL. As previously
    mentioned, the collide
41 // function itself is optional. If you don't need to do any special
    resource management,
42 // just pass "NULL" in place of "collide" in the initialize function.
43 RBTree my_rbtree;

```

---

---

```

44 rbtree_init(&my_rbtree, compare, collide, NULL);
45
46 // Populate your RBTree. Notice how the API abstracts itself and only
   cares about the
47 // RBTreeNode and its key. Because red black trees are always ordered
   in some way, order
48 // of insertion will never affect the inorderness of the tree.
49 rbtree_insert(&my_rbtree, &obj2.key, &obj2.node);
50 rbtree_insert(&my_rbtree, &obj1.key, &obj1.node);
51
52 // Because of the way our compare function is designed, the RBTree
   stores its RBTreeNodes
53 // inorder from smallest key to greatest key. Let's see what is stored
   in the RBTree:
54 int i = 1;
55 RBTreeNode *n;
56 rbtree_for_each(n, &my_rbtree) {
57     if (i == 1) {
58         assert(n == &obj1.node);
59     } else if (i == 2) {
60         assert(n == &obj2.node);
61     }
62
63     ++i;
64 }
65
66 ...
67
68 // Let's get the RBTreeNode with the greatest key.
69 RBTreeNode *greatest_node_ptr = rbtree_last(&my_rbtree);
70
71 // Getting the "node" member from an Object variable is easy: ("obj1.
   node"),
72 // but how do you get the Object variable when you only have the "node"
   member?
73 // Solution: the macro "rbtree_entry":
74 struct Object *obj_ptr = rbtree_entry(greatest_node_ptr, struct Object,
   node);
75 assert(obj_ptr == &obj2);

```

## HashTable

```

1 // Define your struct somewhere.
2 struct Object {
3     int key;
4     ...
5
6     // Don't forget to embed the HashTableNode!
7     HashTableNode node;
8 };
9

```

---

```

10 ...
11
12 // You must define a hash function which takes in a key and returns its
    hashCode. In this
13 // case we aren't going to do anything fancy since this is just an
    example.
14 size_t hash(const void *key) {
15     return *(const int*)key;
16 }
17
18 // You must define an equal function which determines if a key is equal
    to the key of a HashTableNode.
19 int equal(const void *some_key, const HashTableNode *some_node) {
20     return *(const int*)some_key == hashtable_entry(some_node, struct
        Object, node)->key;
21 }
22
23 // You can OPTIONALLY define a collide function which handles key
    collisions. When a key
24 // collision happens, no matter what, the old HashTableNode will be
    replaced by the new
25 // HashTableNode. If this function is defined, then it will be called
    after the old
26 // HashTableNode is replaced in the HashTable. This function is great
    if you need to free up
27 // resources in the Object variable pertaining to the discarded
    HashTableNode. This function
28 // is also great for enabling multi-key functionality in the HashTable.
29 void collide(const HashTableNode *old_node, const HashTableNode *
    new_node, void *auxiliary_data) {
30     // When a HashTable is created, we can give it auxiliary data to
        hold onto. This data is then
31     // passed to this function for usage. This data, for example, could
        be a memory pool struct
32     // that is needed to free up the resources used by the Object
        variable pertaining to the
33     // old_node parameter.
34
35     ...
36 }
37
38 ...
39
40 // Create some Object variables.
41 struct Object obj1, obj2;
42 obj1.key = 1;
43 obj2.key = 2;
44
45 // You must create a bucket array which is an array of pointers to
    HashTableNodes. This

```

---

---

```

46 // array is used by the HashTable for the duration of its lifetime. In
    this case we will
47 // have 50 buckets in the bucket array.
48 HashTableNode *bucket_array[50];
49
50 // Create you HashTable. In this case we don't have any auxiliary data
    that needs to be used
51 // in the collide function, so we just pass NULL. As previously
    mentioned, the collide
52 // function itself is optional. If you don't need to do any special
    resource management,
53 // just pass "NULL" in place of "collide" in the initialize function.
54 HashTable my_hashtable;
55 hashtable_init(&my_hashtable, bucket_array, 50, hash, equal, collide,
    NULL);
56
57 // Populate your HashTable. Notice how the API abstracts itself and
    only cares about the
58 // HashTableNode and its key.
59 hashtable_insert(&my_hashtable, &obj1.key, &obj1.node);
60 hashtable_insert(&my_hashtable, &obj2.key, &obj2.node);
61
62 // HashTable provides no guarantee on the ordering of HashTableNodes in
    the bucket array.
63 // Let's see what is stored in the HashTable:
64 int sum_of_keys = 0, bucket_index;
65 HashTableNode *n;
66 hashtable_for_each(n, bucket_index, &my_hashtable) {
67     sum_of_keys += hashtable_entry(n, struct Object, node)->key;
68 }
69 assert(sum_of_keys == 3);
70
71 ...
72
73 // Let's get the HashTableNode with the key that equals 1.
74 int key = 1;
75 HashTableNode *node_ptr = hashtable_lookup_key(&my_hashtable, &key);
76
77 // Getting the "node" member from an Object variable is easy: ("obj1.
    node"),
78 // but how do you get the Object variable when you only have the "node"
    member?
79 // Solution: the macro "hashtable_entry":
80 struct Object *obj_ptr = hashtable_entry(node_ptr, struct Object, node)
    ;
81 assert(obj_ptr == &obj1);

```

### Stack

```

1 // Define your struct somewhere.
2 struct Object {

```

---

```

3     int some_value;
4     ...
5
6     // Don't forget to embed the StackNode!
7     StackNode node;
8 };
9
10 ...
11
12 // Create some Object variables.
13 struct Object obj1, obj2;
14 obj1.some_value = 1;
15 obj2.some_value = 2;
16
17 // Create your Stack.
18 Stack my_stack;
19 stack_init(&my_stack);
20
21 // Populate your Stack. Notice how the API abstracts itself and only
    cares about the StackNode.
22 stack_push(&my_stack, &obj1.node);
23 stack_push(&my_stack, &obj2.node);
24
25 // Let's see what is stored in the Stack:
26 int i = 1;
27 StackNode *n;
28 stack_for_each(n, &my_stack) {
29     if (i == 1) {
30         assert(n == &obj2.node);
31     } else if (i == 2){
32         assert(n == &obj1.node);
33     }
34
35     ++i;
36 }
37
38 ...
39
40 // Let's get the StackNode at the top of the Stack.
41 StackNode *top_node_ptr = stack_peek(&my_stack);
42
43 // Getting the "node" member from an Object variable is easy: ("obj1.
    node"),
44 // but how do you get the Object variable when you only have the "node"
    member?
45 // Solution: the macro "stack_entry":
46 struct Object *obj_ptr = stack_entry(top_node_ptr, struct Object, node)
    ;
47 assert(obj_ptr == &obj2);

```

---

---

## Queue

```
1 // Define your struct somewhere.
2 struct Object {
3     int some_value;
4     ...
5
6     // Don't forget to embed the QueueNode!
7     QueueNode node;
8 };
9
10 ...
11
12 // Create some Object variables.
13 struct Object obj1, obj2;
14 obj1.some_value = 1;
15 obj2.some_value = 2;
16
17 // Create your Queue.
18 Queue my_queue;
19 queue_init(&my_queue);
20
21 // Populate your Queue. Notice how the API abstracts itself and only
22   cares about the QueueNode.
23 queue_push(&my_queue, &obj1.node);
24 queue_push(&my_queue, &obj2.node);
25
26 // Let's see what is stored in the Queue:
27 int i = 1;
28 QueueNode *n;
29 queue_for_each(n, &my_queue) {
30     if (i == 1) {
31         assert(n == &obj1.node);
32     } else if (i == 2){
33         assert(n == &obj2.node);
34     }
35     ++i;
36 }
37
38 ...
39
40 // Let's get the QueueNode at the front of the Queue.
41 QueueNode *front_node_ptr = queue_peek(&my_queue);
42
43 // Getting the "node" member from an Object variable is easy: ("obj1.
44   node"),
45 // but how do you get the Object variable when you only have the "node"
46   member?
47 // Solution: the macro "queue_entry":
48 struct Object *obj_ptr = queue_entry(front_node_ptr, struct Object,
49   node);
```



---

```
47 assert(obj_ptr == &obj1);
```

## Installation

This library is written in ANSI C, so the code should work with just about every compiler. Each header/source pair is independent of the others. This makes using an individual data structure easy. Just simply drag and drop the header/source pair into your project directly, and make sure to compile the source file along with your other files.

## Running Tests

You must have the GNU compiler available to run the tests. Make sure you have all the files downloaded pertaining to this library as well.

Simply run this command once in the “tests” directory to run all the tests:

```
1 make
```

Example:

```
1 cd tests/
2 make
3 gcc test_list.c ../src/list.c -o test_list -Wall -Wextra -Werror -
  pedantic-errors -std=c89
4 ./test_list C89
5
6 PASSED: List C89
7
8 rm -f test_list
9 gcc test_list.c ../src/list.c -o test_list -Wall -Wextra -Werror -std=
  gnu89
10 ./test_list GNU89
11
12 PASSED: List GNU89
13
14 rm -f test_list
15 g++ test_list.c ../src/list.c -o test_list -Wall -Wextra -Werror -
  pedantic-errors -std=c++11
16 ./test_list C++11
17
18 PASSED: List C++11
19
20 rm -f test_list
21 g++ test_list.c ../src/list.c -o test_list -Wall -Wextra -Werror -std=
  gnu++11
22 ./test_list GNU++11
```

---

```
23
24 PASSED: List GNU++11
25
26 rm -f test_list
27 gcc test_rbtrees.c ../src/rbtrees.c -o test_rbtrees -Wall -Wextra -Werror
   -pedantic-errors -std=c89
28 ./test_rbtrees C89
29
30 PASSED: RBTree C89
31
32 rm -f test_rbtrees
33 gcc test_rbtrees.c ../src/rbtrees.c -o test_rbtrees -Wall -Wextra -Werror
   -std=gnu89
34 ./test_rbtrees GNU89
35
36 PASSED: RBTree GNU89
37
38 etc... (this goes on for a while)
```

## Contributing

Contributions are welcome!

If you have a feature request, or have found a bug, feel free to open a new issue. If you wish to contribute code, please document thoroughly and make sure to write a test for each function/macro. I will help finalize the code and will help make sure that the conventions used are consistent.