C++ Data Structures from Scratch, Vol. 3.1

Robert MacGregor

Building upon the second book in the series, *C++ Data Structures from Scratch, Vol. 3.1* is a comprehensive guide to creating fully functional, STL-style implementations of more advanced data structures and algorithms, introducing new and powerful C++ language concepts along the way.

Key features:

- 190+ complete source code files, with detailed line-by-line analysis and diagrams
- 50+ sample programs directly illustrating key concepts from each chapter
- Free sample content and online support at the official website, *cppdatastructures.com*

Major topics:

- Tries (speed-optimized)
- Compressed tries (memory-optimized)
- Disjoint sets
- Tables
- Graphs
- Pathfinding algorithms
  - Dijkstra
  - Bellman-Ford
  - Floyd-Warshall
  - Cycle detection
- Connectivity algorithms
  - Kruskal (minimum spanning tree)
  - Tarjan-Hopcroft (articulation points, biconnected components)
  - Kosaraju, Tarjan (strongly connected components)
- Dependency analysis (topological sort)
- ASCII file input
- Exception handling

About the author:

- Robert MacGregor is the developer of a C++ API for financial market trading systems. He is also a CTA (Commodity Trading Advisor) in the National Futures Association, and a Chartered Market Technician in the CMT Association.

Cover illustration: *ConcentriCity* by Mark J. Brady (*markjaybeefractal.com*)

C++ Data Structures from Scratch, Vol. 3.1

Robert MacGregor

Published by South Coast Books

For errata, supplementary material, and contact / purchase information, visit www.cppdatastructures.com.

Cover illustration: *ConcentriCity* by Mark J. Brady (www.markjaybeefractal.com)

ISBN-10: 0-9962115-4-3
ISBN-13: 978-0-9962115-4-3

1st Printing, September 2021

Dedicated to Mom and Dad

Table of Contents

Introduction and Getting Started

Part 1: Tries

1.1: Introducing the *Trie* Class 1
1.2: Converting Characters to Index Values 9
1.3: Searching a Bucket 11
1.4: Completing the Node Implementation 15
1.5: Recursive In-Order Traversal 19
1.6: Lexicographical Comparison 21
1.7: Inserting Elements 27
1.8: String Pair Databases 47
1.9: Iterative In-Order Traversal 53
1.10: Implementing the Iterators 65
1.11: Implementing Search 73
1.12: Erasing Elements 85
1.13: Copy and Assignment 101
1.14: Mixed-Case Prefixes 113
1.15: Digits and Punctuation 121

Part 2: Compressed Tries

2.1: Introducing the *CompressedTrie* Class 127
2.2: Recursive In-Order Traversal 131
2.3: Inserting Elements 133
2.4: Iterative In-Order Traversal 149
2.5: Implementing the Iterators 155
2.6: Implementing Search 157
2.7: Erasing Elements 167
2.8: Copy and Assignment 179

Part 3: Disjoint Sets

3.1: Introducing the *DisjointSet* Class 189
3.2: Implementing the Iterators 201
3.3: Copy and Assignment 205

Part 4: Embedded Maps

4.1: Introducing the *EmbeddedMap* Class 213
4.2: Using a Non-Default *KeyRetrieve* Function 221

4.3: Introducing the *RestrictedEmbeddedMap* Class 223

Part 5: Tables

5.1: Introducing the *Table* Class 225
5.2: Erasing Rows and Columns 241
5.3: Copy and Assignment 247

Part 6: Graphs

6.1: Introducing the *Vertex* and *Edge* Classes 251
6.2: Introducing the *Graph* Class 265
6.3: Copy and Assignment 269
6.4: Loading a Graph from a File 275
6.5: Introducing the *DemoWeightedGraph* Class 279

Part 7: Graph Traversal

7.1: Depth-First Search 283
7.2: Breadth-First Search 293

Part 8: Pathfinding Algorithms

8.1: Dijkstra's Algorithm 303
8.2: Introducing the *Path* Class 321
8.3: Bellman-Ford Algorithm 327
8.4: Optimized Bellman-Ford Algorithm 355
8.5: Floyd-Warshall Algorithm 361
8.6: Cycle Detection 383

Part 9: Connectivity

9.1: Kruskal's Algorithm 393
9.2: Articulation Points 411
9.3: Biconnected Components 427
9.4: Transposition 439
9.5: Kosaraju's Algorithm 449
9.6: Tarjan's Algorithm 461

Part 10: Dependency Analysis

10.1: Topological Sort 471

Introduction and Getting Started

Chapter outline

- *A brief review of Volume 2*
- *Obtaining the accompanying source code*
- *Recommended study approach*
- *A brief overview of Volume 3.1*

Before we begin, let's briefly review the major topics that we covered in Volume 2 of *C++ Data Structures from Scratch*:

- Data structures
  - *Heap*
  - *BTree*
  - *RedBlackTree*
  - *SkipList*
  - *ForwardList* (singly-linked list)
  - *HashTable*

- Algorithms
  - Selection sort, Shell sort, and merge sort
  - Binary search
  - FNV hash

- Language concepts
  - Inheritance
  - Polymorphism, abstract classes, and virtual functions
  - Automatic resource management via shared pointers (*SharedPtr*)
  - Binary numbers, hardware (bit / byte) representation, and bitwise operations

If you haven't yet worked through Volumes 1 and 2, I highly recommended that you do so unless you're already familiar with the material. In addition to building upon the above concepts, we'll also reuse some of the source code from Volumes 1 and 2, which won't be reexplained.

To obtain the accompanying source code for this book (which includes the pertinent source code from Volumes 1 and 2), please visit the official website, *www.cppdatastructures.com*. The source code is divided into three main folders:

- *ds3*, which contains the (new) Volume 3 source code
- *ds2*, which contains (only) the reused source code from Volume 2
- *dss*, which contains (only) the reused source code from Volume 1

The *Source files and folders* section at the beginning of each chapter lists the relevant source code files

and / or folders for that chapter. The root folder (*ds3*) is omitted. If a folder is listed without specific filenames, it means that we'll be using all of the files in that folder. The listing for Chapter 1.1, for example,

Source files and folders

- *TrieNode/1*
- *TrieNode/common/memberFunctions_1.h*

indicates that Chapter 1.1 uses:

- All of the files in the folder *ds3/TrieNode/1*
- The file *ds3/TrieNode/common/memberFunctions_1.h*, but not the other files in *ds3/TrieNode/common*

Some chapters also include a listing of relevant database files and / or diagrams. The root folder (*ds3*) is omitted. The listing for Chapter 1.8, for example,

Database files

- *stringPairDatabase/lowercase_1.txt*

Diagrams

- *diagrams/figure_1.1.txt*

indicates that:

- *ds3/stringPairDatabase/lowercase_1.txt* will be read by the accompanying program
- *ds3/diagrams/figure_1.1.txt* will be referenced throughout the chapter

To view the diagrams, use a plain text editor such as Microsoft Notepad or Notepad++, with the *Word Wrap* function disabled. If *Word Wrap* is enabled, the diagrams may not display correctly.

The recommended study approach for each chapter is

- Open the required source files, database files, and diagrams.
- Read the chapter, following along with the files.
- Compile the source code and run the program.
- Read the chapter again, recreating the source code from scratch.
- Compile the recreated source code and run the program, verifying the output.

Before we begin, here's a brief overview of what we'll cover in Volume 3.1:

In Part 1, we'll implement the *Trie* class. A *trie* (pronounced “try”) is a type of data structure in which the key values are sorted by *prefix*. Searching for the prefix *pen*, for example, would return all the keys beginning with *pen*, such as {*penguin*, *penitentiary*, *pentagon*}.

In Part 2, we'll implement the *CompressedTrie* class, which provides the same functionality as *Trie*, but sacrifices raw speed for greater memory efficiency.

In Part 3, we'll implement the *DisjointSet* class. A *disjoint set* is a type of data structure that lets us create groups of elements, called *subsets*, which can be *merged* (combined). We can also efficiently determine whether two given elements belong to the same subset. Disjoint sets play an integral role in *Kruskal's Algorithm*, which we'll implement in Part 9.

In Part 4, we'll create the *EmbeddedMap* and *RestrictedEmbeddedMap* classes. Both of these classes are thin wrappers around a *std::map*, providing a more specialized interface. We'll use them to implement the *Table* and *Graph* classes in Parts 5 and 6.

In Part 5, we'll implement the *Table* class, a collection of elements sorted by rows and columns, similar to a spreadsheet. In addition to looking up individual elements, we can easily insert or erase an entire row (or column), as well as iterate through each element in a particular row (or column). We'll use the *Table* class to implement the *Floyd-Warshall Algorithm* in Part 8, and the *Hungarian Algorithm* in Volume 3.2.

In Part 6, we'll implement the *Graph* class, a type of data structure that models relationships between elements, called *vertices*. Unlike the nodes in a tree, the vertices in a graph aren't limited to parent-child relationships; any *vertex* (element) can be connected to any other vertex.

In Part 7, we'll implement two methods of visiting each vertex in a graph, *depth-first traversal* and *breadth-first traversal*.

In Part 8, we'll implement three algorithms for finding the shortest path between vertices: *Dijkstra's Algorithm*, the *Bellman-Ford Algorithm*, and the *Floyd-Warshall Algorithm*. We'll also implement an algorithm to detect *cycles* (looping paths) within a graph.

In Part 9, we'll implement some common algorithms that evaluate the *connectivity* of a graph (the robustness of the connections between vertices): *Kruskal's Algorithm*, the *Tarjan-Hopcroft Algorithm*, *Kosaraju's Algorithm*, and *Tarjan's Algorithm*.

In Part 10, we'll implement *topological sort*, a method of sorting vertices from least-dependent to most-dependent.

C++ Data Structures from Scratch, Vol. 3.1

Part 3: Disjoint Sets

3.1: Introducing the *DisjointSet* Class

Source files and folders

- *DisjointSet/1*
- *DisjointSet/common/DisjointSetNode.h*
- *DisjointSet/common/memberFunctions_1.h*
- *printDisjointSet*

Chapter outline

- *Overview and terminology*
- *Implementing push_back, findRoot, and unionByRank*
- *Printing the entire structure*

A *disjoint set* is a type of data structure in which the elements are divided into *non-overlapping subsets* (subsets that don't share any common elements). Each subset is represented as a tree, deriving its name from its root element. The disjoint set

```
a  b  c  w  x  y  z
```

for example, contains a total of 7 elements, {*a*, *b*, *c*, *w*, *x*, *y*, *z*}, which may be divided into several possible subsets, such as

```
a  b  c  w  x  y  z

  Subset a consists of element {a}
  Subset b consists of element {b}
  Subset c consists of element {c}
  Subset w consists of element {w}
  Subset x consists of element {x}
  Subset y consists of element {y}
  Subset z consists of element {z}
```

```
a  c  w__   z
|     |  |
b     x  y

  Subset a consists of elements {a,b}
  Subset c consists of element  {c}
  Subset w consists of elements {w,x,y}
  Subset z consists of element  {z}
```

```
b__   y_____
|  |  |  |  |
a  z  w  x  z

  Subset b consists of elements {b,a,z}
  Subset y consists of elements {y,w,x,z}
```

Disjoint sets are optimized to perform the following operations:

- Creating a new subset (inserting a new element)
- Determining which subset a given element belongs to (finding the root of a given element)
- Merging two subsets

A *DisjointSet<T>*, which contains elements of type *T*, is implemented as a *vector* of nodes. Each node contains

- an element of type *T*
- an index value, corresponding to the node's position within the *vector*
- a pointer to the parent node (null for root nodes)
- a rank value (only applicable to root nodes), used when merging two subsets

As usual, we'll refer to each node by its contained element. The *DisjointSet<char>*

```
index  0  1  2  3  4  5  6
node   a  b  c  w  x  y  z

a(1)__      w(2)_______
 |    |      |    |    |
 b    c      x    y    z
```

for example, contains a vector of 7 nodes:

- Node *a* contains the *char* value *a*, an index value of 0, a null parent pointer, and a rank of 1
- Node *b* contains the *char* value *b*, an index value of 1, and a parent pointer to node *a*
- Node *c* contains the *char* value *c*, an index value of 2, and a parent pointer to node *a*
- Node *w* contains the *char* value *w*, an index value of 3, a null parent pointer, and a rank of 2
- Node *x* contains the *char* value *x*, an index value of 4, and a parent pointer to node *w*
- Node *y* contains the *char* value *y*, an index value of 5, and a parent pointer to node *w*
- Node *z* contains the *char* value *z*, an index value of 6, and a parent pointer to node *w*

We'll begin by implementing the node class (*DisjointSetNode.h*). The template parameter *T* (line 8) is the element type. The member types (lines 11-14) are

- *value_type*, the element type *T*
- *Index*, an alias of *size_t*
- *Rank*, an alias of *size_t*

- *Node*, an alias of *DisjointSetNode*

and the data members (lines 20-23) are

- *element*, the contained element
- *index*, the node's relative position within the *vector*
- *parent*, a pointer to the node's parent
- *rank*, the node's rank

The constructor (line 16) initializes the *element* and *index* via the given arguments, the *parent* pointer to null, and the *rank* to 0 (lines 28-31).

The member function *isRoot* (line 18) returns *true* if the node is a root, by checking its *parent* pointer against null (line 39).

Now that we've implemented the node class, we're ready to begin implementing *DisjointSet*. The template parameter *T* (*DisjointSet.h*, line 12) is the element type. The member type *Node* (line 16) is an alias of *DisjointSetNode<T>*, and the private data members (lines 51, 59-60) are

- *_nodes*, a *vector<Node*>*
- *_alloc*, an *allocator<Node>*

The default constructor (line 25) initializes *_nodes* as empty (*memberFunctions_1.h*, lines 3-7).

The private member function *_createNode* (*DisjointSet.h*, line 55) creates a new node containing the given *element* and *index*, and returns a pointer to the new node (*memberFunctions_1.h*, lines 157-166).

_destroyNode (*DisjointSet.h*, line 56) destroys the given node *n* and deallocates the corresponding memory block (*memberFunctions_1.h*, lines 168-173).

_destroyAllNodes (*DisjointSet.h*, line 57) destroys each node in *_nodes* (*memberFunctions_1.h*, lines 175-180).

The destructor (*DisjointSet.h*, line 26) simply calls *_destroyAllNodes* (*memberFunctions_1.h*, lines 9-13).

The public member function *empty* (*DisjointSet.h*, line 28) returns *true* if the set is empty (*memberFunctions_1.h*, lines 15-19).

size (*DisjointSet.h*, line 29) returns the total number of elements (*memberFunctions_1.h*, lines 21-25).

capacity (*DisjointSet.h*, line 30) returns the maximum number of elements that the set can contain, at which point inserting another element will automatically trigger a reallocation (*memberFunctions_1.h*, lines 27-31).

front / *back* (*DisjointSet.h*, lines 31-32, 37-38) return references to the first and last elements (*memberFunctions_1.h*, lines 33-43, 64-74).

operator[] (*DisjointSet.h*, lines 33, 39) returns a reference to the element at the given *index* (*memberFunctions_1.h*, lines 45-50, 76-81).

beginBlock (*DisjointSet.h*, lines 34, 45) returns a pointer to the first node pointer (*memberFunctions_1.h*, lines 52-56, 109-113).

endBlock (*DisjointSet.h*, lines 35, 46) returns a pointer to the one-past-the-last node pointer (*memberFunctions_1.h*, lines 58-62, 115-119).

reserve (*DisjointSet.h*, line 40) increases the capacity to meet or exceed the desired *newCapacity*, triggering a reallocation. If the *newCapacity* is less than or equal to the current capacity, the function does nothing (*memberFunctions_1.h*, lines 83-87).

push_back (*DisjointSet.h*, line 41) inserts the *newElement* at the back of the set. The newly inserted element is the root of a new subset (*memberFunctions_1.h*, lines 89-93).

clear (*DisjointSet.h*, line 42) removes all elements from the set (*memberFunctions_1.h*, lines 95-100).

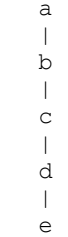
The private member function *_findRoot* (*DisjointSet.h*, line 53) finds the root of the given node *n*, sets *n*'s parent link to the root, and returns a pointer to the root:

- If *n* isn't the root (*memberFunctions_1.h*, line 130), we recursively call *_findRoot* for *n*'s parent and assign the return value (the root of *n*) to *n*'s parent link (line 132). We then return *n*'s parent (which is now the root of *n*) (line 133).

- If *n* is the root, we simply return *n* (line 136).

Given the set

```
a  b  c  d  e
```



for example, the function call

```
_findRoot(d);
```

performs the following operations:

```
_findRoot(d)
{
  d->parent = _findRoot(c)
              {
                c->parent = _findRoot(b)
                            {
                              b->parent = _findRoot(a)
                                          {
                                            return a;
                                          }
                                        = a;

                              return b->parent (a);
                            }
                          = a;

                return c->parent (a);
              }
            = a;

  return d->parent (a);
}
```

The layout of the tree thus becomes

```
a_____
|  |  |
b  c  d
      |
      e
```

in which every node along the ancestral path from *d* to *b* has been *promoted* (been made an immediate child of the root). Node *c*, which was formerly node *a*'s grandchild, is now a direct child of node *a*. Similarly, node *d*, which was formerly node *a*'s great-grandchild, is now a direct child of node *a*.

The public member function *findRoot* (*DisjointSet.h*, line 47) simply forwards the call to *_findRoot* (*memberFunctions_1.h*, lines 121-125). This will allow us to update *findRoot* later on, without having to duplicate the code in *_findRoot*.

The member function *flatten* (*DisjointSet.h*, line 44) promotes each node in the set:

- For trees consisting of 1 node, the height (0) is unchanged.
- For trees consisting of 2 nodes, the height (1) is also unchanged.
- For trees consisting of 3 or more nodes, the height (which is greater than or equal to 1) is reduced to 1.

To implement this function, we simply call *_findRoot* for each node in the set (*memberFunctions_1.h*, lines 105-106). Given the set

```
b    d    i    a    j    h    e

a    h
|    |
b    i
|    |
c    j
|
d
|
e
```

for example, *flatten* performs the following operations:

```
_findRoot(b);          // no promotion (b->parent is the root)
```

```
_findRoot(d);

  a_____      h
  |  |  |     |
  b  c  d     i
        |     |
        e     j
```

```
_findRoot(i);          // no promotion (i->parent is the root)
```

```
_findRoot(a);          // no promotion (a is the root)
```

```
_findRoot(j);

  a_____      h__
  |  |  |     |  |
  b  c  d     i  j
        |
        e
```

```
_findRoot(h);          // no promotion (h is the root)
```

```
_findRoot(e);

  a________      h__
  |  |  |  |     |  |
  b  c  d  e     i  j
```

The private member function *_unionByRank* (*DisjointSet.h*, line 54) takes two nodes *a* and *b*, and merges their respective subsets according to rank. If *a* and *b* are in the same subset, no merging takes place.

We begin by finding the roots of *a* and *b*, designated as *x* and *y* (*memberFunctions_1.h*, lines 142-143). If *x* and *y* are the same node (meaning that *a* and *b* have the same root, thus belonging to the same subset), we return without doing anything (lines 145-146).

If *a* and *b* belong to different subsets, the lower-ranking root becomes a child of the higher-ranking root:

- If *x* ranks lower than *y*, then *x* becomes a child of *y* (lines 148-149)
- If *x* ranks higher than or equal to *y*, then *y* becomes a child of *x* (lines 150-151)
- If *x* ranks equal to *y*, *x*'s rank increases by 1 (lines 153-154)

Given the set

```
a(0)    b(0)    c(0)    w(0)    x(0)    y(0)    z(0)
```

for example, let's walk through a few calls to *_unionByRank*:

```
_unionByRank(a, b)
{
  a's root (a) ranks equal to b's root (b)
    attach b to a;
    ++a->rank;
}

  a(1)    c(0)    w(0)    x(0)    y(0)    z(0)
   |
   b
```

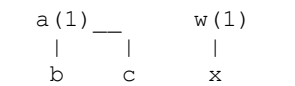
```
_unionByRank(b, c)
{
  b's root (a) ranks higher than c's root (c)
    attach c to a;
}

  a(1)__      w(0)    x(0)    y(0)    z(0)
   |    |
   b    c
```

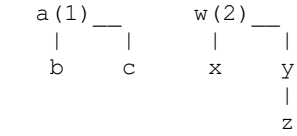
```
_unionByRank(w, x)
{
  w's root (w) ranks equal to x's root (x)
    attach x to w;
    ++w->rank;
}

  a(1)__      w(1)    y(0)    z(0)
   |    |      |
   b    c      x
```

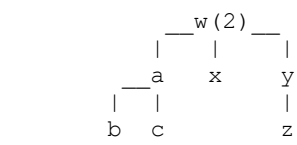
```
_unionByRank(y, z)
{
  y's root (y) ranks equal to z's root (z)
    attach z to y;
    ++y->rank;
}
```



```
_unionByRank(x, z)
{
  x's root (w) ranks equal to z's root (y)
    attach y to w;
    ++w->rank;
}
```



```
_unionByRank(b, w)
{
  b's root (a) ranks lower than w's root (w)
    attach a to w;
}
```



The public member function *unionByRank* (*DisjointSet.h*, line 43) simply forwards the call to *_unionByRank* (*DisjointSet.h*, lines 63-67). This will allow us to update *unionByRank* later on, without having to duplicate the code in *_unionByRank*.

The standalone function (*printDisjointSet.h*, lines 8-9),

```
template <class DisjointSet>
void printDisjointSet(DisjointSet& d);
```

prints the following information for each node in the given set *d* (lines 24-29):

- The index value
- The contained element
- The root element

- The rank (for root nodes only)

Note that the parameter *d* (line 12) is a non-*const* reference, which we need in order to call the *findRoot* method (which modifies the structure of the set).

Our test program (*main.cpp*) begins by constructing a set of *Traceable<char>* and reserving a capacity of 7 elements (lines 12-16). We then insert the elements $\{a, b, c, w, x, y, z\}$, each belonging to their own subset, and print the entire structure (lines 18-25).

In lines 27-45, we replicate the earlier examples of *_unionByRank*. This time, however, we call *printDisjointSet* after each union, which flattens the entire structure by calling *findRoot* on each node:

```
index            0       1       2       3       4       5       6
element(rank)    a(0)    b(0)    c(0)    w(0)    x(0)    y(0)    z(0)

(index element root rank):
0 a a 0
1 b b 0
2 c c 0
3 w w 0
4 x x 0
5 y y 0
6 z z 0
```

```
unionByRank(a, b);            // unionByRank(*(p + 0), *(p + 1));

  a(1)    c(0)    w(0)    x(0)    y(0)    z(0)
   |
   b

printDisjointSet();           // no promotion

(index element root rank):
0 a a 1
1 b a
2 c c 0
3 w w 0
4 x x 0
5 y y 0
6 z z 0
```

```
unionByRank(b, c);            // unionByRank(*(p + 1), *(p + 2));

  a(1)__      w(0)    x(0)    y(0)    z(0)
   |    |
   b    c

printDisjointSet();           // no promotion

// Continued on next page
```

```
(index element root rank):
0 a a 1
1 b a
2 c a
3 w w 0
4 x x 0
5 y y 0
6 z z 0
```

```
unionByRank(w, x);            // unionByRank(*(p + 3), *(p + 4));

  a(1)__      w(1)    y(0)    z(0)
   |    |      |
   b    c      x

printDisjointSet();           // no promotion

(index element root rank):
0 a a 1
1 b a
2 c a
3 w w 1
4 x w
5 y y 0
6 z z 0
```

```
unionByRank(y, z);            // unionByRank(*(p + 5), *(p + 6));

  a(1)__      w(1)    y(1)
   |    |      |       |
   b    c      x       z

printDisjointSet();           // no promotion

(index element root rank):
0 a a 1
1 b a
2 c a
3 w w 1
4 x w
5 y y 1
6 z y
```

```
unionByRank(x, z);            // unionByRank(*(p + 4), *(p + 6));

  a(1)__      w(2)__
   |    |      |    |
   b    c      x    y
                     |
                     z
```

```
printDisjointSet();           // promotes z

  a(1)__      w(2)_______
   |    |      |    |    |
   b    c      x    y    z

(index element root rank):
0 a a 1
1 b a
2 c a
3 w w 2
4 x w
5 y w
6 z w
```

```
unionByRank(b, w);            // unionByRank(*(p + 1), *(p + 3));

           ___w(2)_______
          |    |    |    |
      ____a    x    y    z
     |    |
     b    c

printDisjointSet();           // promotes b and c

   _____________w(2)_______
  |    |    |    |    |    |
  c    b    a    x    y    z

(index element root rank):
0 a w
1 b w
2 c w
3 w w 2
4 x w
5 y w
6 z w
```

```
~ a
~ b
~ c
~ w
~ x
~ y
~ z

All Traceables destroyed
```