C++ Data Structures from Scratch, Vol. 2

Robert MacGregor

Building upon the first book in the series, *C++ Data Structures from Scratch, Vol. 2* 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:

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

Major topics:

- Heaps (STL *priority_queue*, *make_heap*, *push_heap*, *pop_heap*)
- Heap sort
- Selection sort
- Shell sort
- Merge sort
- Binary search
- B-trees
- Red-black trees (STL *map*)
- Skip lists
- Inheritance and polymorphism
- Smart pointers (STL *shared_ptr*)
- Singly-linked lists (STL *forward_list*)
- Binary representation and bitwise operations
- FNV hash
- Hash tables (STL *unordered_map*)

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: *IteratedConduits* by Mark J. Brady (*markjaybeefractal.com*)

C++ Data Structures from Scratch, Vol. 2

Robert MacGregor

Published by South Coast Books

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

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

ISBN-10: 0-9962115-3-5
ISBN-13: 978-0-9962115-3-6

1st Printing, March 2018

Dedicated to Chris L. Marchlewski
(1955-2017)

Table of Contents

Part 12: Hash Tables

Introduction and Getting Started

Chapter outline

- *A brief review of Volume 1*
- *Obtaining the accompanying source code*
- *Recommended study approach*
- *A brief overview of Volume 2*

Before we begin, let's briefly review the major topics that we covered in Volume 1 of *C++ Data Structures from Scratch*:

- Installing and configuring an IDE (integrated development environment)
- Compiling source code into an executable program
- Basic programming concepts (variables, arithmetic, and logic)
- Program organization (functions, namespaces, and header files)
- Indirection (pointers, arrays, references, and *const* correctness)
- Object-oriented programming (classes) and operator overloading
- Template metaprogramming and function objects
- Recursion
- Implementing the *bubble sort*, *insertion sort*, and *quick sort* algorithms
- Dynamic memory allocation (implementing the *Allocator* class)
- Implementing the *Traceable* class, to verify that our data structure implementations properly destroy their dynamically-allocated elements
- Implementing the data structure classes:
  - *Array* (fixed-size array)
  - *Vector* (dynamic array)
  - *List* (doubly-linked list)
  - *Ring* (single-block ring buffer / deque)
  - *MultiRing* (multi-block ring buffer / deque)
  - *BinaryTree* (unbalanced binary search tree)
  - *AVLTree* (balanced binary search tree)
- Implementing *iterator*s, *const_iterator*s, *reverse_iterator*s, and *const_reverse_iterator*s
- Iterator categories (tags) and time complexity

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

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

- *ds2*, which contains the (new) Volume 2 source code
- *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 (*ds2*) 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*
>
> - *Heap/1*
> - *Heap/common/memberFunctions_1.h*

indicates that Chapter 1.1 uses:

- All of the files in the folder *ds2/Heap/1*
- The file *ds2/Heap/common/memberFunctions_1.h*, but not the other files in *ds2/Heap/common*

The recommended study approach is unchanged from Volume 1:

- At the beginning of each chapter, compile the included source code and run the program.
- Read the chapter, following along with the included source code.
- Read the chapter again, recreating the source code from scratch.
- Compile the recreated source code and run the program, verifying the output.

Here's a brief overview of what we'll cover in Volume 2:

In Part 1, we'll implement the *Heap* class. A *heap*, also known as a *priority queue*, is a type of data structure that keeps the largest (or smallest) element at the top. We'll also create a set of generic stand-alone functions for transforming any type of random access container (*vector*, *deque*, etc.) into a heap. We'll then use those functions to implement the *heap sort* algorithm.

In Parts 2-4, we'll implement the *selection sort*, *Shell sort*, and *merge sort* algorithms.

In Part 5, we'll implement the *binary search* algorithm, an efficient method of searching sorted sequences, used extensively in Part 6. We'll also introduce the concept of *inheritance* and learn how it applies to the Standard Library's *iterator_tag*s.

In Part 6, we'll implement the *BTree* class. A *B-tree* is a type of balanced search tree, in which each node can contain more than one element and have more than two children.

In Part 7, we'll implement the *RedBlackTree* class. A *red-black tree* is an order-4 B-tree, implemented using a traditional binary search tree.

In Part 8, we'll implement the *SkipList* class. A *skip list* is a special type of linked list that provides the same operations as a balanced search tree along with comparable (logarithmic-time) performance, but

with a simpler implementation.

In Part 9, we'll introduce the concepts of *polymorphism*, *abstract classes*, and *virtual functions*. We'll also implement the *SharedPtr* (*shared pointer*) class, which automates the destruction of a dynamically-allocated object.

In Part 10, we'll use polymorphism to implement the *ForwardList* class. A *forward list* is a singly-linked list, which is more efficient (though less versatile) than its doubly-linked counterpart. We'll use forward lists in Part 12.

In Part 11, we'll discuss *binary numbers* and the hardware (*bit* / *byte*) representation of some basic data types (*char*, *string*, *int*, *double*). We'll also learn how to perform *bitwise operations* (manipulating objects at the bit level), completing the groundwork for Part 12.

In Part 12, we'll implement the *FNV hash function* and *HashTable* class. A *hash table* is an associative data structure that provides even faster (constant-time) access than balanced search trees.

C++ Data Structures from Scratch, Vol. 2

Part 1: Heaps (Priority Queues)

1.1: Introducing the *Heap* Class

Source files and folders

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

Chapter outline

- *Using an array, vector, or deque to represent a binary tree*
- *Member functions:*
  - *Default / copy constructors, destructor, and assignment operator*
  - *Accessor methods: empty, size, top*
  - *push*

A *heap*, also known as a *priority queue*, is a collection in which the largest element is always at the *top* of the heap (also called the *front* of the queue). A heap supports 3 main operations:

- *push*: Inserts a new element. If the new element is the largest in the heap, it's moved to the top.
- *top*: Returns the top (largest) element
- *pop*: Removes the top element, then moves the next largest element to the top.

Consider, for example, an empty *Heap<int> h*:

```
h.push(5)    // Insert 5 (5 becomes the top element)
h.top()      // Element 5

h.push(4)    // Insert 4 (5 remains at the top)
h.top()      // Element 5

h.push(7)    // Insert 7 (7 becomes the top element)
h.top()      // Element 7

h.push(6)    // Insert 6 (7 remains at the top)
h.top()      // Element 7

h.push(8)    // Insert 8 (8 becomes the top element)
h.top()      // Element 8
```

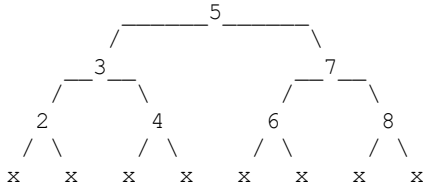
h now contains the elements {5, 4, 7, 6, 8}, but not necessarily in that order. A heap only guarantees that the largest element is at the top; no assumption can be made about the order of the other elements:

```
h.pop()      // Remove the top element, 8, then move 7 to the top
h.top()      // Element 7

h.pop()      // Remove the top element, 7, then move 6 to the top
h.top()      // Element 6

h.pop()      // Remove the top element, 6, then move 5 to the top
h.top()      // Element 5

h.pop()      // Remove the top element, 5, then move 4 to the top
h.top()      // Element 4

h.pop()      // Remove the top element, 4, after which h becomes empty
```

Internally, a heap is a binary tree. But unlike the trees we studied in Volume 1, a heap stores its elements in a single dynamic array as opposed to a set of linked nodes. When inserting a new element, we first place it at the back of the array, then move the largest element to the front via swapping.

Any container supporting *push_back*, *pop_back*, and random access will suffice, but for this implementation we'll use a *deque*. Although a *vector* would provide slightly faster element access (and therefore slightly faster sorting), *deque*'s *push* operation will incur much less overhead as the heap grows larger.

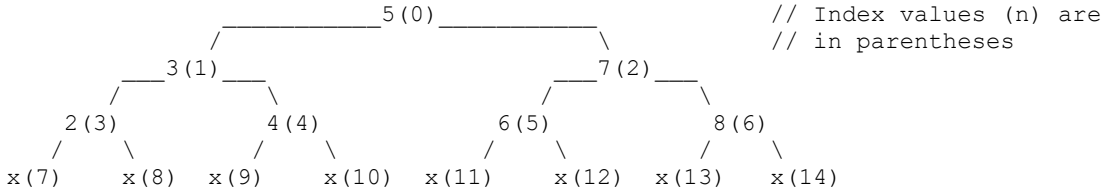
But how exactly can we use a *deque* (or any array-like structure, for that matter) to represent a binary tree? Consider, for example, the tree



We can store this layout in an array by assigning an index value to each node. The root, element 5, is placed at index 0. Proceeding left to right, element 3 is placed at index 1, element 7 at index 2, etc. The entire array thus becomes

Element	5	3	7	2	4	6	8
Node (n) (Array index)	0	1	2	3	4	5	6

and the tree diagram becomes




```
                                                    // Index values (n) are
                                                    // in parentheses
```

Given the index of a node *n*,

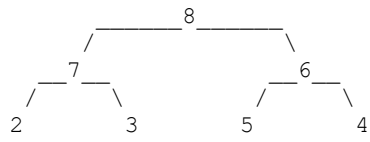
- *n*'s parent is located at index (*n*/2 – 1), rounded up to the nearest integer
- *n*'s left child is located at index (2*n* + 1)
- *n*'s right child is located at index (2*n* + 2), i.e. (the index of *n*'s left child + 1)

The following table demonstrates these formulas for the above tree:

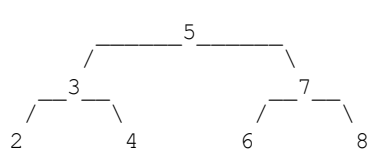
Element	Node (n)	Parent = n/2 - 1	Left = 2n + 1	Right = Left + 1
5	0	-1 (n/a)	1	2
3	1	0	3	4
7	2	0	5	6
2	3	1	7	8
4	4	1	9	10
6	5	2	11	12
8	6	2	13	14

The third row, for example, indicates that element 7 resides in node 2, its parent is node 0 (element 5), its left child is node 5 (element 6), and its right child is node 6 (element 8). Similarly, the sixth row indicates that element 6 resides in node 5, its parent is node 2 (element 7), its left child is node 11 (null), and its right child is node 12 (null).

In addition to the underlying storage mechanism, the other major difference between a heap and a traditional binary search tree lies in the relationships between elements. In a heap, the only requirement is that each element is greater than both of its children:




```
// This is a heap because:
// 8 is greater than both its children, 7 & 6
// 7 is greater than both its children, 2 & 3
// 6 is greater than both its children, 5 & 4
```




```
// This is not a heap because:
// 5 is not greater than its child 7
// 3 is not greater than its child 4
// 7 is not greater than its child 8
```

The actual relationship between siblings is undefined: in any sibling pair, the left child may be greater than the right or vice versa, as long as both siblings are less than their parent. In the above (first) diagram, for example, element 7 is greater than its right sibling (6), while element 2 is less than its right sibling (3). The defining property of a heap is that both elements in each sibling pair are less than their parent: 7 and 6 are less than 8, 2 and 3 are less than 7, and 5 and 4 are less than 6.

Heaps and traditional binary search trees are similar in that they both use a predicate to sort their elements. Recall from Volume 1 that a less-than predicate sorts a binary search tree in ascending order (least to greatest); conversely, a greater-than predicate sorts the tree in descending order (greatest to

least). By applying this concept to heaps, we can generate two symmetrical types:

- A *max heap*, which uses a less-than predicate, keeps the largest element at the top
- A *min heap*, which uses a greater-than predicate, keeps the smallest element at the top

The *Heap* class is defined in the file *Heap.h* (lines 9-30). The Standard Library header <*functional*> (line 5) provides the *less* / *greater* predicates. The template parameter *T* represents the element type, and the default *Predicate* type is *std::less*<*T*> (line 9).

container_type (line 13) is an alias of the container class (*deque*<*T*>) used to store the elements.

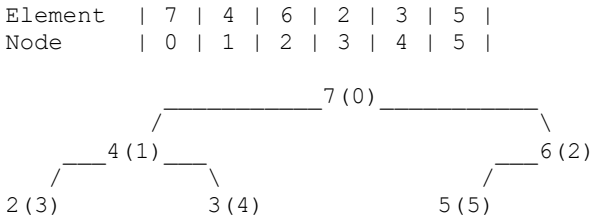
Node (line 26) is the type used to represent node index values. We're using a plain *int* here (as opposed to an *unsigned int*) because the root's (null) parent has an index of -1.

The member functions *empty* and *size* (*Heap.h*, lines 19-20) simply forward the calls to the deque (*memberFunctions_1.h*, lines 6-16). The member function *top* (*Heap.h*, line 21) returns the root element (at index 0), which is always at the front of the deque (*memberFunctions_1.h*, lines 18-22).

Because we won't be explicitly allocating memory, we can use the compiler-generated constructors, destructor, and assignment operator.

To push a new element onto the heap, we first place it at the back of the deque, which corresponds to the bottom of the tree (a leaf node). We then move the new element up the tree until it's at the correct location. In a max heap, we achieve this by comparing the new element to its parent. If the new element is less than its parent, we're done; otherwise, we swap them, climb up to the next level, and repeat the process. If we reach the root node, it also indicates that we're done because there are no more nodes left to compare.

Consider, for example, the max heap



and suppose that we're pushing a new element, 8:

```
// Place the new element (8) at the back of the deque

  Element  | 7 | 4 | 6 | 2 | 3 | 5 | 8 |
  Node     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
```

```
                  ____________7(0)____________      Parent
                 /                            \    /
            ___4(1)___                      ___6(2)___
           /          \                    /          \
      2(3)            3(4)              5(5)            8(6) <- Current node

  // Element 8 is not less than its parent (6), so swap them and climb up to
  // the next node

    Element  | 7 | 4 | 8 | 2 | 3 | 5 | 6 |
    Node     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

                               Parent
                              /
                  ____________7(0)____________      Current node
                 /                            \    /
            ___4(1)___                      ___8(2)___
           /          \                    /          \
      2(3)            3(4)              5(5)            6(6)

  // Element 8 is not less than its parent (7), so swap them and climb up to
  // the next node

    Element  | 8 | 4 | 7 | 2 | 3 | 5 | 6 |
    Node     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

                               Current node
                              /
                  ____________8(0)____________
                 /                            \
            ___4(1)___                      ___7(2)___
           /          \                    /          \
      2(3)            3(4)              5(5)            6(6)

  // We've reached the root of the tree.  There are no more nodes left to
  // compare, so we're done.
```

The *push* method is defined in lines 24-45 (*memberFunctions_1.h*). *n* (line 31) is the index of the current node (initialized to that of the new node), and the loop condition,

```
n > 0
```

tests whether or not the current node is the root (index 0). The statement (line 33)

```
Node nParent = static_cast<Node>(ceil(n / 2.0)) - 1;
```

gets us the index of *n*'s parent, using the aforementioned formula (Parent = *n*/2 – 1, rounded up to the nearest integer). If *n* is 3, for example, it becomes

```
Node nParent = static_cast<Node>(ceil(3 / 2.0)) - 1;
             = static_cast<Node>(ceil(1.5)) - 1;
             = static_cast<Node>(2.0) - 1;
             = 2 - 1;
             = 1;
```

The Standard Library function *ceil* (short for "ceiling") returns the smallest integer greater than or equal to the given value, as a floating-point number; this is why we're explicitly converting the result to an integer via *static_cast*. Similarly, the Standard Library function *floor* returns the largest integer less than or equal to the given value:

```
  n     ceil(n)    floor(n)
_____________________________

-2        -2         -2
-1.8      -1         -2
-0.5       0         -1
 0         0          0
 0.3       1          0
 1         1          1
 1.6       2          1
```

The <*cmath*> header (line 1) provides *ceil* and *floor*, and the <*utility*> header (line 2) provides the Standard Library *swap* function.

Once we have the index of *n*'s parent, we can compare the two elements and proceed accordingly. Consider, for example, an empty *Heap<int> h*, which uses the default (less-than) predicate:

```
h.push(5);

_deque.push_back(5);

  Element  | 5 |
  Node     | 0 |

Initializer:

  Node n = _deque.size() - 1;                          // n = 0

Terminate loop
```

```
h.push(3);

_deque.push_back(3);

  Element  | 5 | 3 |
  Node     | 0 | 1 |
```

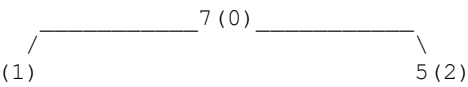
```
              ____________5(0)
             /
          3(1)
```

```
Initializer:

  Node n = _deque.size() - 1;                           // n = 1

Iteration 1:

  Node nParent = static_cast<Node>(ceil(n/2.0)) - 1;  // nParent = 0

  if (_predicate(_deque[n], _deque[nParent]))         // Is 3 less than 5?
  {
    break;                                            // Yes, terminate loop
  }
  else
  {
    swap(_deque[n], _deque[parent]);
    n = nParent;
  }
```

```
h.push(7);

_deque.push_back(7);

  Element  | 5 | 3 | 7 |
  Node     | 0 | 1 | 2 |
```

```
              ____________5(0)____________
             /                            \
          3(1)                           7(2)
```

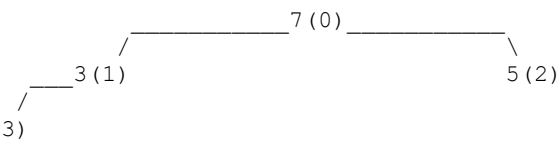
```
Initializer:

  Node n = _deque.size() - 1;                           // n = 2

Iteration 1:

  Node nParent = static_cast<Node>(ceil(n/2.0)) - 1;  // nParent = 0

  if (_predicate(_deque[n], _deque[nParent]))         // Is 7 less than 5?
  {
    break;
  }
  else
  {
    swap(_deque[n], _deque[parent]);                  // No, swap 7 and 5
    n = nParent;                                      // n = 0
  }

  Element  | 7 | 3 | 5 |
  Node     | 0 | 1 | 2 |
```



Terminate loop

```
h.push(2);

_deque.push_back(2);
```

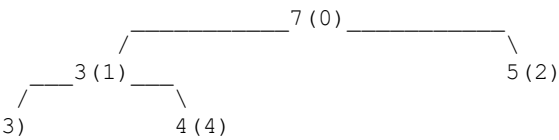
```
  Element  | 7 | 3 | 5 | 2 |
  Node     | 0 | 1 | 2 | 3 |
```




```
Initializer:

  Node n = _deque.size() - 1;                          // n = 3

Iteration 1:

  Node nParent = static_cast<Node>(ceil(n/2.0)) - 1;  // nParent = 1

  if (_predicate(_deque[n], _deque[nParent]))          // Is 2 less than 3?
  {
    break;                                             // Yes, terminate loop
  }
  else
  {
    swap(_deque[n], _deque[parent]);
    n = nParent;
  }
```

```
h.push(4);

_deque.push_back(4);
```

```
  Element  | 7 | 3 | 5 | 2 | 4 |
  Node     | 0 | 1 | 2 | 3 | 4 |
```



```
Initializer:

  Node n = _deque.size() - 1;                              // n = 4

Iteration 1:

  Node nParent = static_cast<Node>(ceil(n/2.0)) - 1;  // nParent = 1

  if (_predicate(_deque[n], _deque[nParent]))          // Is 4 less than 3?
  {
    break;
  }
  else
  {
    swap(_deque[n], _deque[parent]);                   // No, swap 4 and 3
    n = nParent;                                       // n = 1
  }

  Element  | 7 | 4 | 5 | 2 | 3 |
  Node     | 0 | 1 | 2 | 3 | 4 |

              ___________7(0)___________
             /                          \
        ___4(1)___                      5(2)
       /          \
  2(3)           3(4)

Iteration 2:

  Node nParent = static_cast<Node>(ceil(n/2.0)) - 1;  // nParent = 0

  if (_predicate(_deque[n], _deque[nParent]))          // Is 4 less than 7?
  {
    break;                                             // Yes, terminate loop
  }
  else
  {
    swap(_deque[n], _deque[parent]);
    n = nParent;
  }
```

```
h.push(6);

_deque.push_back(6);

  Element  | 7 | 4 | 5 | 2 | 3 | 6 |
  Node     | 0 | 1 | 2 | 3 | 4 | 5 |

              ___________7(0)___________
             /                          \
        ___4(1)___                      ___5(2)
       /          \                    /
  2(3)           3(4)              6(5)
```

```
Initializer:

  Node n = _deque.size() - 1;                              // n = 5

Iteration 1:

  Node nParent = static_cast<Node>(ceil(n/2.0)) - 1;  // nParent = 2
  if (_predicate(_deque[n], _deque[nParent]))         // Is 6 less than 5?
  {
    break;
  }
  else
  {
    swap(_deque[n], _deque[parent]);                  // No, swap 6 and 5
    n = nParent;                                      // n = 2
  }

  Element  | 7 | 4 | 6 | 2 | 3 | 5 |
  Node     | 0 | 1 | 2 | 3 | 4 | 5 |

              ___________7(0)___________
             /                          \
        ___4(1)___                      ___6(2)
       /          \                    /
  2(3)           3(4)             5(5)

Iteration 2:

  Node nParent = static_cast<Node>(ceil(n/2.0)) - 1;  // nParent = 0

  if (_predicate(_deque[n], _deque[nParent]))         // Is 6 less than 7?
  {
    break;                                            // Yes, terminate loop
  }
  else
  {
    swap(_deque[n], _deque[parent]);
    n = nParent;
  }
```

```
h.push(8);

_deque.push_back(8);

  Element  | 7 | 4 | 6 | 2 | 3 | 5 | 8 |
  Node     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

              ___________7(0)___________
             /                          \
        ___4(1)___                      ___6(2)___
       /          \                    /          \
  2(3)           3(4)             5(5)           8(6)
```

```
Initializer:

  Node n = _deque.size() - 1;                              // n = 6

Iteration 1:

  Node nParent = static_cast<Node>(ceil(n/2.0)) - 1;  // nParent = 2
  if (_predicate(_deque[n], _deque[nParent]))         // Is 8 less than 6?
  {
    break;
  }
  else
  {
    swap(_deque[n], _deque[parent]);                  // No, swap 8 and 6
    n = nParent;                                      // n = 2
  }

  Element  | 7 | 4 | 8 | 2 | 3 | 5 | 6 |
  Node     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

              ___________7(0)___________
             /                          \
        ___4(1)___                    ___8(2)___
       /          \                  /          \
  2(3)           3(4)            5(5)           6(6)

Iteration 2:

  Node nParent = static_cast<Node>(ceil(n/2.0)) - 1;  // nParent = 0

  if (_predicate(_deque[n], _deque[nParent]))         // Is 8 less than 7?
  {
    break;
  }
  else
  {
    swap(_deque[n], _deque[parent]);                  // No, swap 8 and 7
    n = nParent;                                      // n = 0
  }

  Element  | 8 | 4 | 7 | 2 | 3 | 5 | 6 |
  Node     | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

              ___________8(0)___________
             /                          \
        ___4(1)___                    ___7(2)___
       /          \                  /          \
  2(3)           3(4)            5(5)           6(6)

Terminate loop
```

To test the *push* method, we'll use the function (*main.cpp*, line 7)

```
void pushAndPrintTop(Heap<int>* h, int i);
```

which pushes the given value *i* onto the heap, then prints the current *size* and *top* element (lines 30-39). Our test program (lines 10-26) constructs a *Heap<int> h* and pushes the values {5, 3, 7, 2, 4, 6, 8}, generating the output

```
push 5
  size 1
  top 5

push 3
  size 2
  top 5

push 7
  size 3
  top 7

push 2
  size 4
  top 7

push 4
  size 5
  top 7

push 6
  size 6
  top 7

push 8
  size 7
  top 8
```