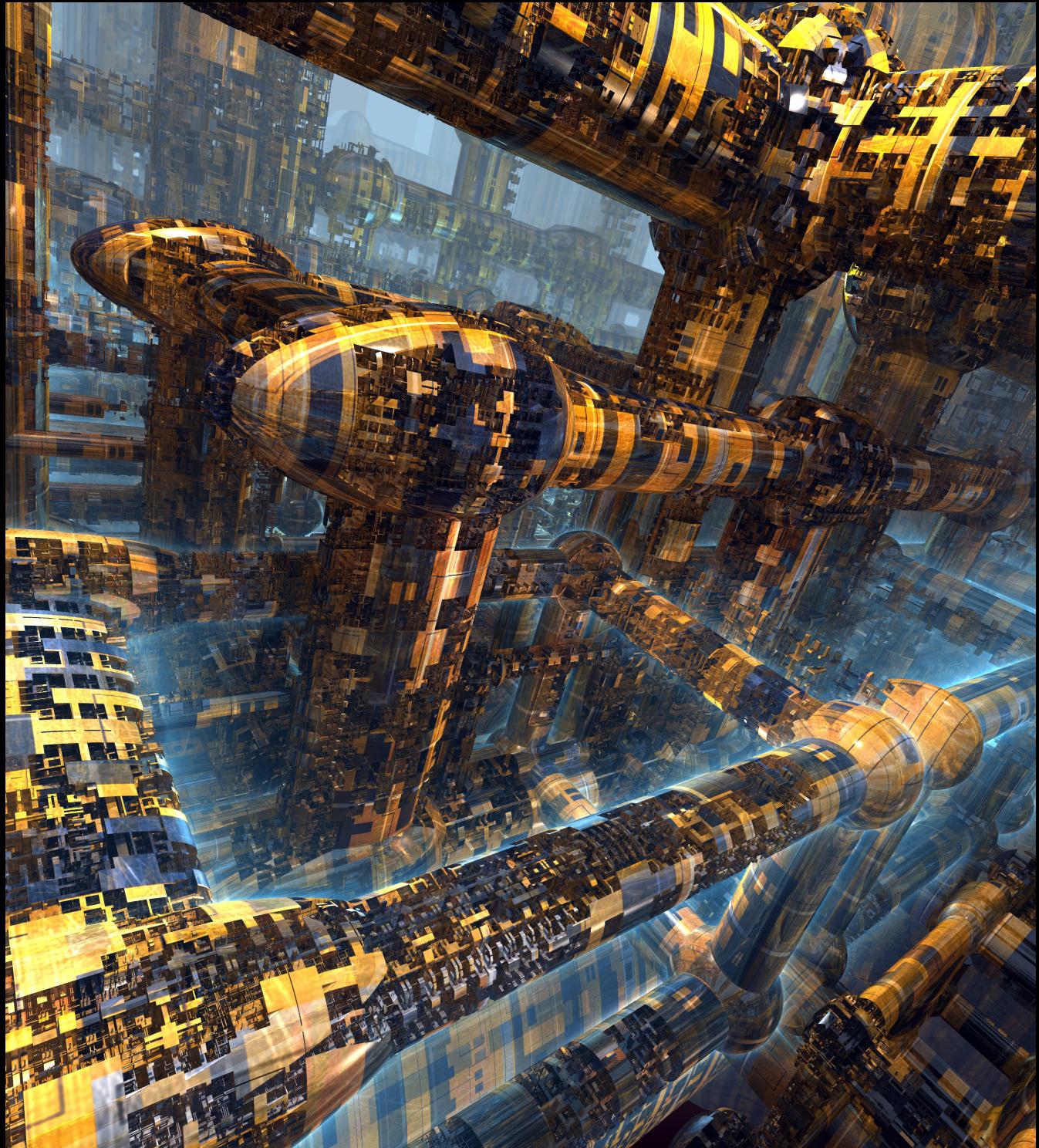


C++ Data Structures from Scratch, Vol. 2



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.

C++ Data Structures from Scratch, Vol. 2

Robert MacGregor

Copyright 2018 by Robert MacGregor. All rights reserved.

No part of this book may be reproduced or transmitted by any means without the prior written consent of the author.

Although every precaution has been taken to verify the accuracy of the information contained herein, the author and publisher assume no responsibility for errors or omissions. Furthermore, no liability is assumed for any damages resulting from the use of the information or programs contained herein.

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

Introduction and Getting Started

Part 1: Heaps (Priority Queues)

1.1: Introducing the <i>Heap</i> Class	1
1.2: Completing the <i>Heap</i> Class	13
1.3: Rearranging an Existing Sequence into a Heap	27
1.4: The <i>pushHeap</i> Function	33
1.5: The <i>popHeap</i> Function	37
1.6: Heap Sort	41

Part 2: Selection Sort

2.1: Finding the Smallest and Largest Element	45
2.2: Completing the Implementation	51

Part 3: Shell Sort

3.1: Subsequence Sorting	55
3.2: Choosing a Series of Gap Sizes	63
3.3: Completing the Implementation	67

Part 4: Merge Sort

4.1: Splitting a Sequence in Half	69
4.2: Merging Sorted Halves	75
4.3: Completing the Implementation	83

Part 5: Binary Search

5.1: Inheritance and Iterator Tags	85
5.2: Finding the Upper and Lower Bound	89
5.3: Completing the Implementation	97

Part 6: B-Trees

6.1: Introducing the <i>KmPair</i> Class	99
6.2: Implementing Binary Search for <i>KmPairs</i>	105
6.3: Introducing the <i>BTreeNode</i> Class	109

6.4: Recursive In-Order Traversal	115
6.5: Introducing the <i>BTree</i> Class	119
6.6: Iterative In-Order Traversal	151
6.7: Implementing the Iterators	159
6.8: Erasing Elements	163
6.9: Implementing Copy and Assignment	199

Part 7: Red-Black Trees

7.1: Introducing the <i>RedBlackTree</i> Class	205
7.2: Inserting Elements	213
7.3: Erasing Elements	243

Part 8: Skip Lists

8.1: Introducing the <i>SkipList</i> Class	273
8.2: Increasing the Capacity	299
8.3: Implementing the Iterators	307
8.4: Erasing Elements	311
8.5: Implementing Copy and Assignment	323

Part 9: Polymorphism and Smart Pointers

9.1: Abstract Classes and Virtual Functions	335
9.2: Introducing the <i>SharedPtr</i> Class	345

Part 10: Forward Lists

10.1: Introducing the <i>ForwardList</i> Class	351
10.2: Erasing the Front Element	357
10.3: Implementing the Iterators	359
10.4: Inserting and Erasing Elements in the Middle	361
10.5: Implementing Copy and Assignment	365
10.6: Introducing the <i>ForwardListSize</i> Class	369

Part 11: Bit Representation and Bitwise Operations

11.1: Binary Numbers, Characters, and Strings	371
11.2: Integers	381
11.3: Floating-Point (Real) Numbers	389
11.4: Bitwise Operations	397

Part 12: Hash Tables

12.1: The FNV Hash Function	415
12.2: Introducing the <i>HashTable</i> Class	423
12.3: Implementing the Local Iterators	433
12.4: Erasing Elements	437
12.5: Implementing Copy and Assignment	441
12.6: Adjusting the Max Load Factor	447
Index	451

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 *iterators*, *const_iterators*, *reverse_iterators*, and *const_reverse_iterators*
- Iterator categories (tags) and time complexity

If you haven't yet worked through Volume 1, I highly recommend 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/I*
- *Heap/common/memberFunctions_I.h*

indicates that Chapter 1.1 uses:

- All of the files in the folder *ds2/Heap/I*
- The file *ds2/Heap/common/memberFunctions_I.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_tags*.

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/I*
- *Heap/common/memberFunctions_I.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

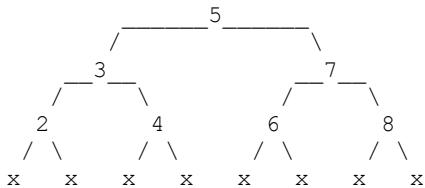
h.pop()      // Remove the top element, 4, after which h becomes empty

```

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.

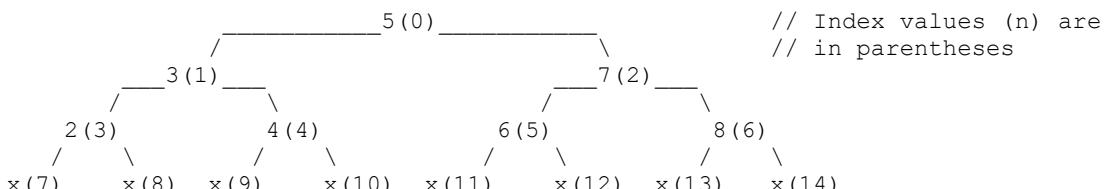
But how exactly can we use a *deque* (or any array-like structure, for that matter) to represent a binary tree? Consider, for example, the tree



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



Given the index of a node n ,

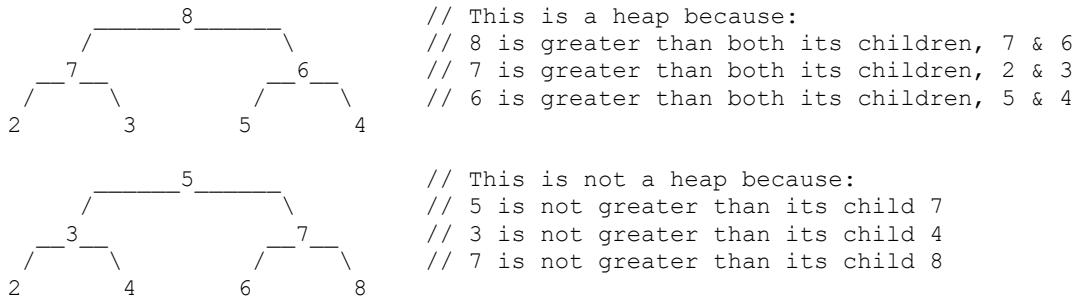
- n 's parent is located at index $(n/2 - 1)$, rounded up to the nearest integer
- n 's left child is located at index $(2n + 1)$
- n 's right child is located at index $(2n + 2)$, i.e. (the index of n 's left child + 1)

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:



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.

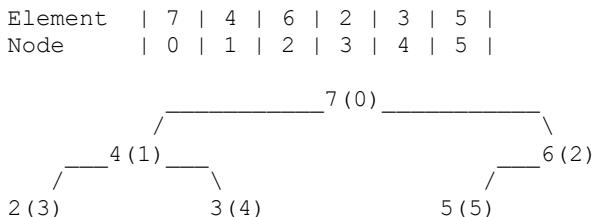
Node (line 26) is the type used to represent node index values. We're using a plain *int* here (as opposed to an *unsigned int*) because the root's (null) parent has an index of -1.

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) 8(6) <- Current node
 / \   / \
2(3) 3(4) 5(5) 6(6)

// 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 |

    7(0) _____ Parent
   /   \   /
4(1) 8(2) 6(6) Current node
 / \   / \
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 |

    8(0) _____ Current node
   /   \   /
4(1) 7(2) 6(6)
 / \   / \
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	<i>ceil</i> (n)	<i>floor</i> (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 |

```



Initializer:

```
Node n = _dequeue.size() - 1; // n = 1
```

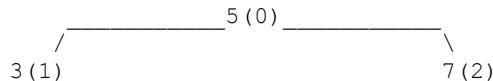
Iteration 1:

```
Node nParent = static_cast<Node>(ceil(n/2.0)) - 1; // nParent = 0
if (_predicate(_dequeue[n], _dequeue[nParent])) // Is 3 less than 5?
{
    break; // Yes, terminate loop
}
else
{
    swap(_dequeue[n], _dequeue[parent]);
    n = nParent;
}
```

```
h.push(7);
```

```
_dequeue.push_back(7);
```

Element	5	3	7	
Node	0	1	2	



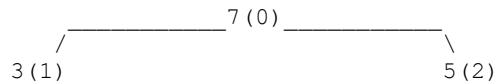
Initializer:

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

Iteration 1:

```
Node nParent = static_cast<Node>(ceil(n/2.0)) - 1; // nParent = 0
if (_predicate(_dequeue[n], _dequeue[nParent])) // Is 7 less than 5?
{
    break;
}
else
{
    swap(_dequeue[n], _dequeue[parent]); // No, swap 7 and 5
    n = nParent; // n = 0
}
```

Element	7	3	5	
Node	0	1	2	

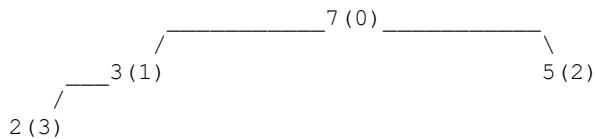


Terminate loop

```
h.push(2);
```

```
_dequeue.push_back(2);
```

Element	7	3	5	2	
Node	0	1	2	3	



Initializer:

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

Iteration 1:

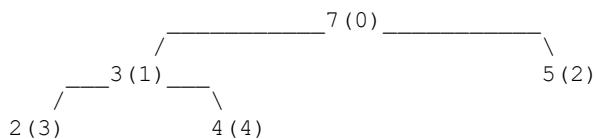
```

Node nParent = static_cast<Node>(ceil(n/2.0)) - 1; // nParent = 1
if (_predicate(_dequeue[n], _dequeue[nParent]))           // Is 2 less than 3?
{
    break;                                                 // Yes, terminate loop
}
else
{
    swap(_dequeue[n], _dequeue[parent]);
    n = nParent;
}
```

```
h.push(4);
```

```
_dequeue.push_back(4);
```

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



Initializer:

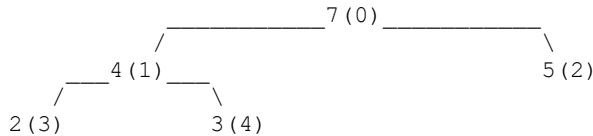
```
Node n = _dequeue.size() - 1; // n = 4
```

Iteration 1:

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

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

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



Iteration 2:

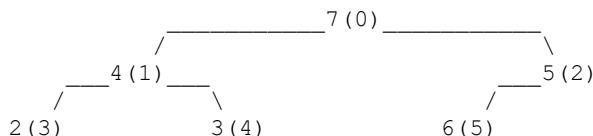
```
Node nParent = static_cast<Node>(ceil(n/2.0)) - 1; // nParent = 0

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

`h.push(6);`

`_dequeue.push_back(6);`

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



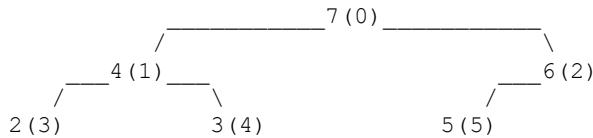
Initializer:

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

Iteration 1:

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

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



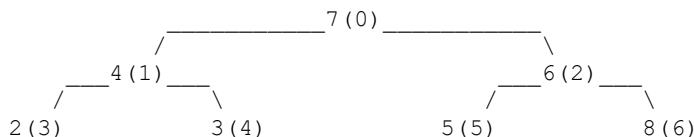
Iteration 2:

```
Node nParent = static_cast<Node>(ceil(n/2.0)) - 1; // nParent = 0
if (_predicate(_dequeue[n], _dequeue[nParent])) // Is 6 less than 7?
{
    break; // Yes, terminate loop
}
else
{
    swap(_dequeue[n], _dequeue[parent]);
    n = nParent;
}
```

```
h.push(8);
```

```
_dequeue.push_back(8);
```

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



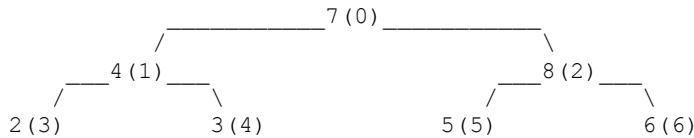
Initializer:

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

Iteration 1:

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

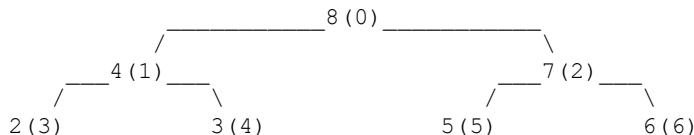
Element	7	4	8	2	3	5	6
Node	0	1	2	3	4	5	6



Iteration 2:

```
Node nParent = static_cast<Node>(ceil(n/2.0)) - 1; // nParent = 0
if (_predicate(_dequeue[n], _dequeue[nParent])) // Is 8 less than 7?
{
    break;
}
else
{
    swap(_dequeue[n], _dequeue[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



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 $\text{Heap} < \text{int} >$ 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
```

1.2: Completing the *Heap* Class

Source files and folders

- *Heap/2*
- *Heap/common/memberFunctions_2.h*

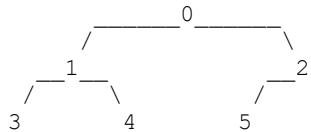
Chapter outline

- *Implementing Heap's pop method*

Recall from the previous chapter that when inserting a new node, we always place it at the back of the deque. Because the nodes are indexed left-to-right beginning at the root, every node will fall into one of three categories:

- The node is a leaf, i.e. it has no children
- The node has a left child only
- The node has two children

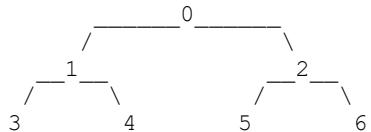
It isn't possible, in other words, for any node to have a right child only; before a node can have a right child, it must already have a left child. To convince ourselves that this is true, we can sketch a tree diagram of just the index values:



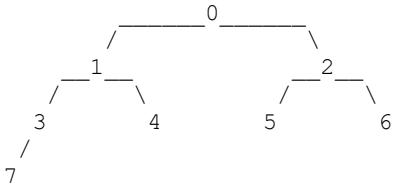
The index values indicate the order in which these nodes were inserted (0, 1, 2, 3, 4, 5). The diagram shows that for any node to have a right child, it must already have a left child:

- For node 0 to have its right child (2), it must have already had its left child (1)
- For node 1 to have its right child (4), it must have already had its left child (3)

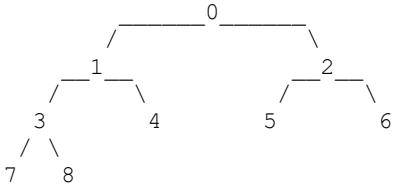
The next node, 6, will be the right child of node 2, which already had its left child (5):



The next node, 7, must be the left child of node 3:



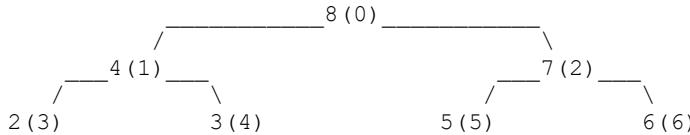
The next node, 8, will be the right child of node 3, which already had its left child (7):



The fact that every node is guaranteed to have either no children, a left child only, or two children is an important part of the *pop* algorithm.

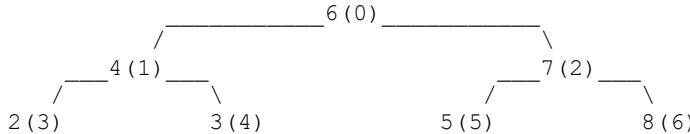
To pop the top element, we swap it with the bottom element, at the back of the deque. The bottom element, which is guaranteed to be a leaf, can then be erased by simply calling *pop_back* on the deque. We then move the top element down the tree until it's at the correct position. Once the process is complete, the new largest element will be at the top. Consider, for example, the heap

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



```
// To pop the top element (8), swap it with the bottom element (6), which is
// guaranteed to be a leaf
```

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



```
// Erase the bottom element (8) by calling pop_back on the deque
```

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

```

    Current node
    /
   6(0) _____ \
   /           \
4(1)         7(2)
   / \       /
2(3) 3(4) 5(5)

// Move the top element (6) down the tree until it's at the correct position

// Element 6 has two children (4 and 7), so we compare it to its larger
// child (7)

// 6 is less than 7, so swap them and proceed to the child (node 2)

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

    7(0) _____ \
    /           \
4(1)         6(2) <- Current node
   / \       /
2(3) 3(4) 5(5)

// Element 6 has a left child only (5), so we compare it to the left child

// 6 is greater than 5, so it's at the correct position

// The new largest element (7) is now at the top

```

Note that if an element has two children, we must compare it to its largest child. This is necessary to determine whether the element is larger than both of its children. In the above example, if we had compared element 6 to its smaller child (4) instead of its larger one (7), we would've mistakenly concluded that element 6 was at the correct position.

Before discussing the implementation, let's walk through another example:

```

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)

// To pop the top element (7), swap it with the bottom element (5), which is
// guaranteed to be a leaf

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

```

```

      5 (0)
     /   \
  4 (1)   6 (2)
 / \       /
2 (3) 3 (4) 7 (5)

// Erase the bottom element (7) by calling pop_back on the deque

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

      Current node
      /
      5 (0)
     /   \
  4 (1)   6 (2)
 / \       /
2 (3) 3 (4)

// Move the top element (5) down the tree until it's at the correct position

// Element 5 has two children (4 and 6), so we compare it to its larger
// child (6)

// 5 is less than 6, so swap them and proceed to the child (node 2)

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

      6 (0)
     /   \
  4 (1)   5 (2) <- Current node
 / \       \
2 (3) 3 (4)

// Element 5 is a leaf, so the process is complete

// The new largest element (6) is now at the top

```

But how do we determine whether a given node has two children, a left child only, or no children? Recall the proof from the beginning of this chapter: if a node n has a right child, then n must have two children. A node n therefore has two children if the index value of n 's right child lies within the deque's bounds.

To get the index values of n 's left and right children, we'll write the private member functions (*Heap.h*, lines 29-30)

```

Node _leftChild(Node n) const;      // Returns 2n + 1
Node _rightChild(Node n) const;     // Returns _leftChild(n) + 1

```

which simply use the formulas from the previous chapter (*memberFunctions_2.h*, lines 29-41). We can then use the *_rightChild* method to implement the member function (*Heap.h*, line 34)

```
bool _hasTwoChildren(Node n) const;
```

which will tell us whether n has two children by checking the index of n 's right child against the deque's upper bound. If the right child's index value is in bounds, then the right child exists and n has two children (*memberFunctions_2.h*, lines 69-73). The *static_cast* (line 72) isn't required, but without it the compiler would warn us about comparing a signed integer (*_rightChild(n)*) with an unsigned one (*_dequeue.size()*).

Because a node n cannot have a right child only, it follows that if n doesn't have a left child, then n must be a leaf. The member function (*Heap.h*, line 33)

```
bool _isLeaf(Node n) const;
```

will tell us whether n is a leaf by checking the index of n 's left child against the deque's upper bound. If the left child's index value is out of bounds, then the left child doesn't exist and n is a leaf (*memberFunctions_2.h*, lines 63-67).

As illustrated above, to move a node n down the tree:

- If n has two children, we must compare n to its larger child (either the left or right)
- If n has a left child only, we simply compare n to its left child

The member function (*Heap.h*, line 31)

```
Node _getChildForComparison(Node n) const;
```

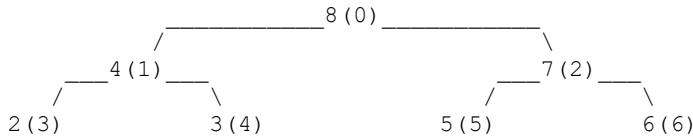
returns the index of n 's larger child (if n has 2 children), or the index of n 's left child (if n doesn't have 2 children) (*memberFunctions_2.h*, lines 43-61). Note that the phrase "largest child" only applies to a max heap; for a min heap, the function would return the index of n 's smallest child since we would be using the inverse predicate.

We can now implement the *pop* method (*Heap.h*, line 24) using *_isLeaf* and *_getChildForComparison* (*memberFunctions_2.h*, lines 5-27). The loop (lines 13-26) descends the tree, beginning at the root. n is the index of the current node, and we descend the tree until we've reached a leaf node.

In each iteration, *nChild* is either the index of n 's larger child (if n has 2 children) or n 's left child (if n has a left child only). We then compare the corresponding elements. If the current element is less than the child element (line 17), we swap them, proceed to the child node, and repeat the process (lines 19-20). If, however, the current element is larger than the child element (line 22), the current element is at the correct position so we can terminate the loop (line 24).

Once again using the heap

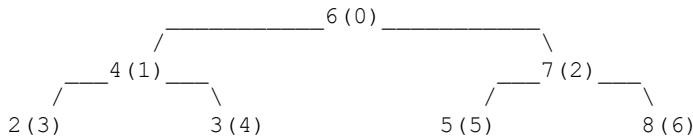
Element		8		4		7		2		3		5		6	
Node		0		1		2		3		4		5		6	



let's walk through the code. To pop the top element, 8:

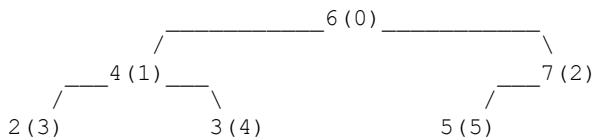
```
swap(_dequeue.front(), _dequeue.back());
```

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



```
_dequeue.pop_back();
```

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



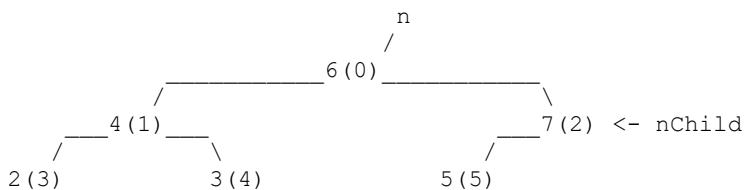
Initializer:

```
Node n = 0;
```

Iteration 1:

```
Node nChild = _getChildForComparison(n); // nChild = 2
```

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

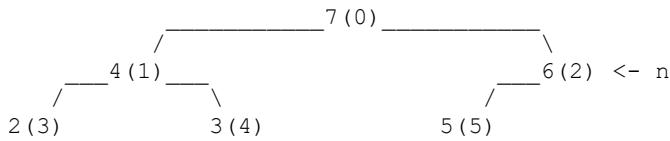


```

if (_predicate(_dequeue[n], _dequeue[nChild]))           // Is 6 less than 7?
{
    swap(_dequeue[n], _dequeue[nChild]);
    n = nChild;
}
else
{
    break;
}

```

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



Iteration 2:

```

Node nChild = _getChildForComparison(n);           // nChild = 5

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


```

```

graph TD
    7((7(0))) --- 4((4(1)))
    7 --- 6((6(2)))
    4 --- 2((2(3)))
    4 --- 3((3(4)))
    6 --- 5((5(5)))
    6 --- nChild((n))
    style 7 fill:none,stroke:none
    style 4 fill:none,stroke:none
    style 6 fill:none,stroke:none
    style 2 fill:none,stroke:none
    style 3 fill:none,stroke:none
    style 5 fill:none,stroke:none
    6 --> "nChild"

```

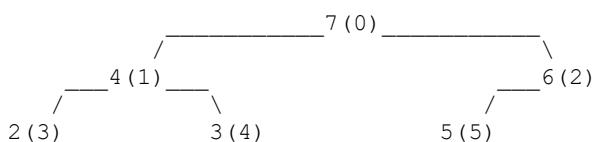
```

if (_predicate(_dequeue[n], _dequeue[nChild]))           // Is 6 less than 5?
{
    swap(_dequeue[n], _dequeue[nChild]);
    n = nChild;
}
else
{
    break;                                              // No, terminate loop
}

```

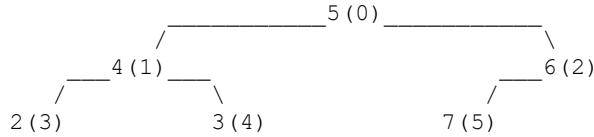
To pop the top element, 7:

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



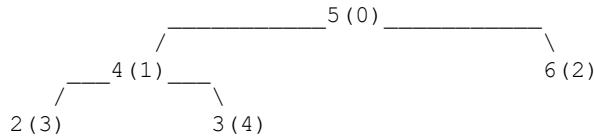
```
swap(_deque.front(), _deque.back());
```

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



```
_deque.pop_back();
```

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



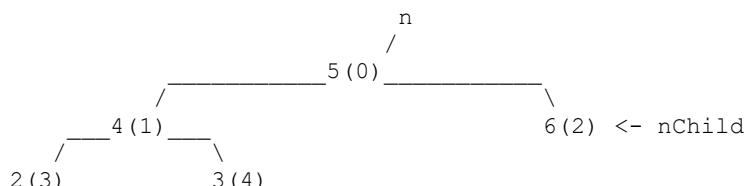
Initializer:

```
Node n = 0;
```

Iteration 1:

```
Node nChild = _getChildForComparison(n); // nChild = 2
```

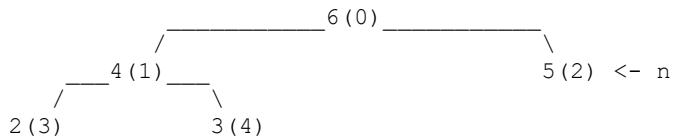
Element		5		4		6		2		3	
Node		0		1		2		3		4	



```

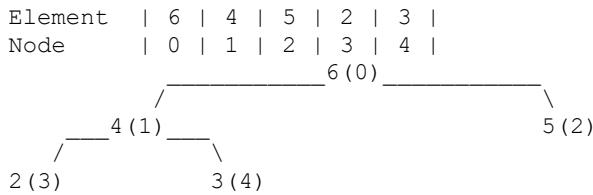
if (_predicate(_deque[n], _deque[nChild])) // Is 5 less than 6?
{
    swap(_deque[n], _deque[nChild]); // Yes, swap 5 and 6
    n = nChild; // n = 2
}
else
{
    break;
}
  
```

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

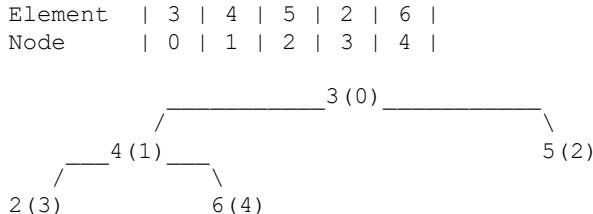


n is a leaf, so the loop terminates

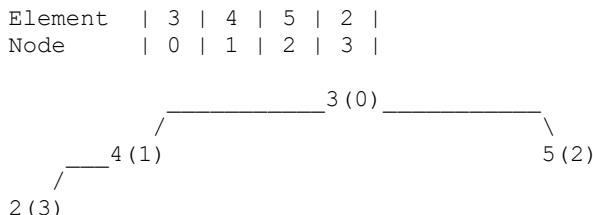
To pop the top element, 6:



`swap(_dequeue.front(), _dequeue.back());`



`_dequeue.pop_back();`



Initializer:

`Node n = 0;`

Iteration 1:

`Node nChild = _getChildForComparison(n);` // nChild = 2



```

n
/
3(0) _____ \
   /           \
4(1)           5(2) <- nChild
   / \
2(3)           3(1)
if (_predicate(_dequeue[n], _dequeue[nChild]))           // Is 3 less than 5?
{
    swap(_dequeue[n], _dequeue[nChild]);                  // Yes, swap 3 and 5
    n = nChild;                                         // n = 2
}
else
{
    break;
}

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

      5(0) _____ \
      /           \
4(1)           3(2) <- n
      / \
2(3)           3(1)

n is a leaf, so the loop terminates

```

To pop the top element, 5:

```

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

      5(0) _____ \
      /           \
4(1)           3(2)
      / \
2(3)           3(1)

swap(_dequeue.front(), _dequeue.back());

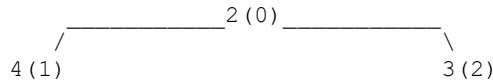
Element | 2 | 4 | 3 | 5 |
Node     | 0 | 1 | 2 | 3 |

      2(0) _____ \
      /           \
4(1)           3(2)
      / \
5(3)           3(1)

_dequeue.pop_back();

```

Element		2		4		3	
Node		0		1		2	



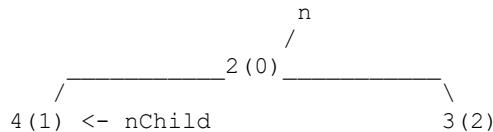
Initializer:

```
Node n = 0;
```

Iteration 1:

```
Node nChild = _getChildForComparison(n); // nChild = 1
```

Element		2		4		3	
Node		0		1		2	

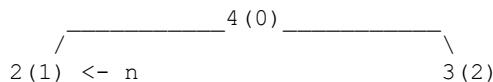


```

if (_predicate(_dequeue[n], _dequeue[nChild])) // Is 2 less than 4?
{
    swap(_dequeue[n], _dequeue[nChild]); // Yes, swap 2 and 4
    n = nChild; // n = 1
}
else
{
    break;
}

```

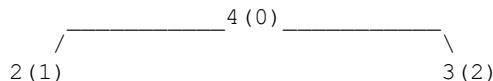
Element		4		2		3	
Node		0		1		2	



n is a leaf, so the loop terminates

To pop the top element, 4:

Element		4		2		3	
Node		0		1		2	



```
swap(_dequeue.front(), _dequeue.back());
```

```

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

      /   3(0)   \
      2(1)           \
                         \
                         4(2)

_deque.pop_back();

Element | 3 | 2 |
Node    | 0 | 1 |

      /   3(0)
      2(1)

Initializer:

Node n = 0;

Iteration 1:

Node nChild = _getChildForComparison(n); // nChild = 1

Element | 3 | 2 |
Node    | 0 | 1 |

      /   3(0) <- n
      2(1) <- nChild

if (_predicate(_deque[n], _deque[nChild])) // Is 3 less than 2?
{
    swap(_deque[n], _deque[nChild]);
    n = nChild;
}
else
{
    break; // No, terminate loop
}

```

To pop the top element, 3:

```

Element | 3 | 2 |
Node    | 0 | 1 |

      /   3(0)
      2(1)

swap(_deque.front(), _deque.back());

```

```

Element | 2 | 3 |
Node    | 0 | 1 |

      _____ 2(0)
      /
3(1)

_deque.pop_back();

Element | 2 |
Node    | 0 |

2(0)

Initializer:

Node n = 0;

2(0) <- n

n is a leaf, so the loop terminates

```

To pop the final element, 2:

```

Element | 2 |
Node    | 0 |

2(0)

swap(_deque.front(), _deque.back());           // The front and back
                                                // are the same element,
                                                // so nothing changes
Element | 2 |
Node    | 0 |

2(0)

_deque.pop_back();                           // _deque is now empty

Initializer:

Node n = 0;

n is a leaf, so the loop terminates

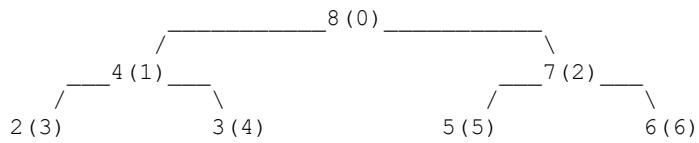
```

Our test program (*main.cpp*) begins by constructing the heap (lines 10-18)

```

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

```



We then print the top element and pop it until the heap is empty (lines 20-24), generating the output

```
top 8
top 7
top 6
top 5
top 4
top 3
top 2
```

1.3: Rearranging an Existing Sequence into a Heap

Source files and folders

- *heapTransform/I*
- *heapTransform/common/getChildForComparison.h*
- *heapTransform/common/makeHeap.h*
- *heapTransform/common/moveElementDownTree.h*
- *heapTransform/common/relatives.h*

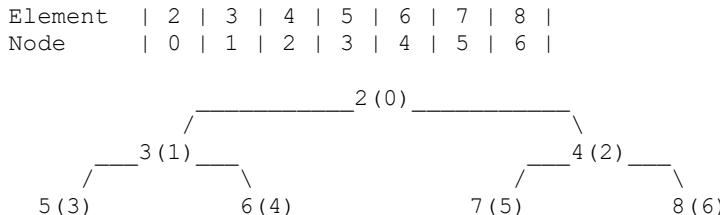
Chapter outline

- Rearranging an existing sequence of elements into a heap

In this chapter we'll create a stand-alone function, *makeHeap*, which rearranges an existing sequence of elements into a heap (*makeHeap.h*, lines 9-10):

```
template <class RanIt, class Predicate>
void makeHeap(RanIt begin, RanIt end, Predicate predicate);
```

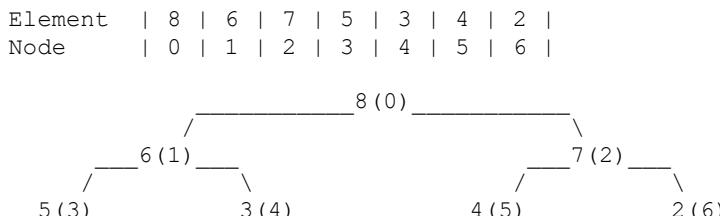
A completely generic solution, this function works for any sequence bound by random access iterators. The template parameter *RanIt* represents the iterators' type. As with the *Heap* class, using a less-than *predicate* creates a max heap, while using a greater-than *predicate* creates a min heap. Given a *vector<int>* *v*



for example, the function call

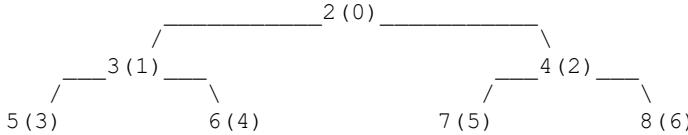
```
makeHeap(v.begin(), v.end(), less<int>());
```

rearranges *v* into the heap



To rearrange the elements, we start at the parent of the bottom node (i.e. the parent of the back element) and work our way right-to-left. In the above example, we begin at node 2 (the parent of the bottom node, 6), followed by nodes 1 and 0. For each node n (2, 1, 0), we move the element at node n down the tree until it's at the correct position. For the tree

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



the entire procedure is

- Move the element at node 2 (4) down the tree to the correct position
 - This turns the subtree rooted at node 2 into a heap
- Move the element at node 1 (3) down the tree to the correct position
 - This turns the subtree rooted at node 1 into a heap
- Move the element at node 0 (2) down the tree to the correct position
 - This turns the tree rooted at node 0 into a heap

The move procedure itself is identical to that of the *Heap* class' *pop* method: we compare the current element to its largest child and swap them if necessary, until it's at the correct position or we've reached the bottom of the tree. But before writing that, let's reimplement the *Heap* class' private helper methods as stand-alone functions (*relatives.h*, lines 10-17, 25-41):

```

template <class Node>           // The index value of n's parent
Node parent(Node n);           // (n/2 - 1, rounded up to the nearest index value)

template <class Node>           // The index value of n's left child (2n + 1)
Node leftChild(Node n);

template <class Node>           // The index value of n's right child
Node rightChild(Node n);       // (leftChild(n) + 1)
  
```

The functions (lines 19-23)

```

template <class Node, class Size>
bool isLeaf(Node n, Size size);

template <class Node, class Size>
bool hasTwoChildren(Node n, Size size);
  
```

differ slightly from those of the *Heap* class in that they require a parameter for the *size* of the underlying container (*array*, *vector*, etc.). Their implementations, however, are identical (lines 43-53).

Note that we've placed all of these functions into their own sub-namespace, *ht* (short for "heap transform") (lines 8-9, 54). We're keeping them out of the main *ds2* namespace because they're lower-level components of *makeHeap*, not meant to be used directly.

We can now use the functions in *relatives.h* to reimplement the *Heap* class' *_getChildForComparison* method as a stand-alone function (*getChildForComparison.h*, lines 10-17):

```
template <class Node,
         class RanIt,
         class Size,
         class Predicate>
Node getChildForComparison(Node n,           // Returns the index value of n's
                           RanIt begin,      // larger child (if n has 2 children),
                           Size size,        // or the index value of n's left child
                           Predicate predicate); // (if n doesn't have 2 children)
```

Compared to *Heap::_getChildForComparison*, this version has two additional parameters:

- *begin* (an iterator to the front element), used to access *n*'s left / right child elements
- *size* (the size of the underlying container), used to determine whether *n* has 2 children

Other than that, the implementation is identical (lines 19-42). We can now use *isLeaf* and *getChildForComparison* to write the function (*moveElementDownTree.h*, lines 13-20)

```
template <class Node,
         class RanIt,
         class Size,
         class Predicate>
void moveElementDownTree(Node n,           // Move the element at node n down the
                           RanIt begin,      // tree to the correct position
                           Size size,
                           Predicate predicate);
```

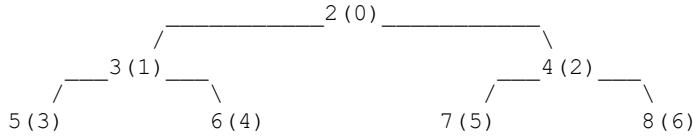
The implementation (lines 22-48) is virtually identical to the move procedure in *Heap::pop* (*Heap/common/memberFunctions_2.h*, lines 13-26). The only difference is that in *Heap::pop*, we moved the element at node 0 down the tree; in this version, we're moving the element at node *n* down the tree.

Writing the *makeHeap* function is now relatively simple, and our code (*makeHeap.h*, lines 9-10, 12-28) directly maps to the description from the beginning of the chapter:

- Start at the parent of the bottom node (i.e. the parent of the back element) (lines 19-21)
- For each node *n* from the starting point to the root (node 0) (line 21), move the element at node *n* down the tree to the correct position (lines 23-26)

To demonstrate the code, let's walk through the example at the beginning of this chapter:

```
Element | 2 | 3 | 4 | 5 | 6 | 7 | 8 |           // vector<int> v
Node    | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
```



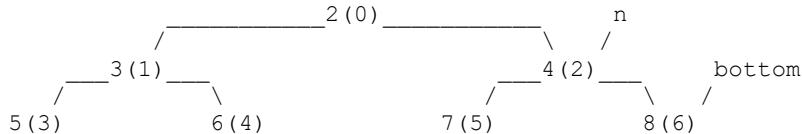
```
makeHeap(v.begin(), v.end(), less<int>());
```

```
Node bottom = end - begin - 1;           // bottom = 6
```

Initializer:

```
Node n = parent(bottom);           // n = 2
```

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



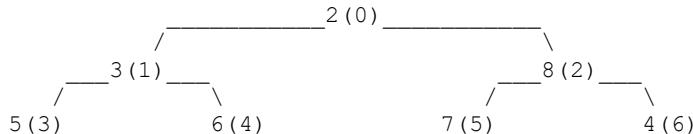
Iteration 1:

```
moveElementDownTree(n,           // Move the element at node 2 (4)
begin,                         // down the tree to the correct
end - begin,                   // position
predicate);
```



```
// Swap the current element (4)
// with its larger child (8)
```

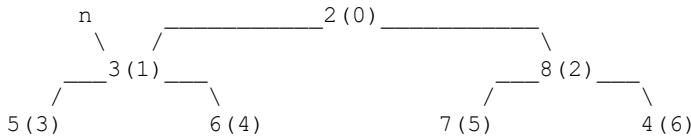
```
Element | 2 | 3 | 8 | 5 | 6 | 7 | 4 |
Node    | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
```



```
// The current element (4) is a
// leaf, so the move is complete
```

```
--n;           // n = 1
```

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



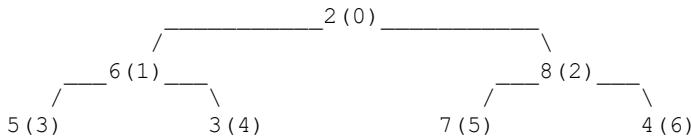
Iteration 2:

```

moveElementDownTree(n,
begin,
end - begin,
predicate);           // Move the element at node 1 (3)
// down the tree to the correct
// heap position

// Swap the current element (3)
// with its larger child (6)
  
```

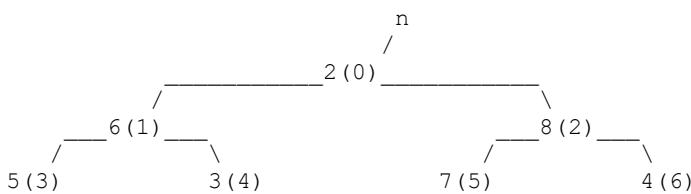
Element	2	6	8	5	3	7	4	
Node	0	1	2	3	4	5	6	



// The current element (3) is a
// leaf, so the move is complete

--n; // n = 0

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



Iteration 3:

```

moveElementDownTree(n,
begin,
end - begin,
predicate);           // Move the element at node 0 (2)
// down the tree to the correct
// heap position

// Swap the current element (2)
// with its larger child (8)
  
```

```

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


$$\begin{array}{c}
& & 8(0) & \\
& / \quad \backslash & & \\
6(1) & & 2(2) & \\
/ \quad \backslash & / \quad \backslash & \\
5(3) & 3(4) & 7(5) & 4(6)
\end{array}$$


// Swap the current element (2)
// with its larger child (7)

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


$$\begin{array}{c}
& & 8(0) & \\
& / \quad \backslash & & \\
6(1) & & 7(2) & \\
/ \quad \backslash & / \quad \backslash & \\
5(3) & 3(4) & 2(5) & 4(6)
\end{array}$$


// The current element (2) is a
// leaf, so the move is complete

--n;           // n = -1

Terminate loop

```

For now the header file *heapTransform.h* (*main.cpp*, line 6) only includes *makeHeap.h*, but we'll be adding to it in the next two chapters. Our test program (lines 8-21) demonstrates the above example, generating the output

```

2 3 4 5 6 7 8      // Original sequence (v)
8 6 7 5 3 2 4      // makeHeap(v.begin(), v.end(), less<int>());

```

1.4: The *pushHeap* Function

Source files and folders

- *heapTransform/2*
- *heapTransform/common/pushHeap.h*

Chapter outline

- *Implementing a stand-alone function that pushes a new element onto an existing heap*

Recall from Chapter 1.1 that pushing a new element onto a heap consists of two main steps:

- Insert the new element at the bottom of the tree (place it at the back of the underlying *vector*, *deque*, etc.)
- Move the new element up the tree (by comparing it to its parent) until it's at the correct position

The function (*pushHeap.h*, lines 10-11)

```
template <class RanIt, class Predicate>
void pushHeap(RanIt begin, RanIt end, Predicate predicate);
```

performs the second step, where the element at (*end* – 1), the back element, is the newly inserted element. Given

```
vector<int> v = {2, 3, 4, 5, 6, 7, 8};
less<int> predicate;

makeHeap(v.begin(), v.end(), predicate); // Rearrange v into a heap
```

for example, we can push a new element (9) onto the heap by writing

```
v.push_back(9); // Insert the new element at the
// bottom of the tree

push_heap(v.begin(), v.end(), predicate); // Move the new element up the
// tree until it's at the
// correct position
```

The process is identical to *Heap::push* (*Heap/common/memberFunctions_1.h*, lines 31-44). We start with the bottom element (*pushHeap.h*, line 18) and compare it with its parent (lines 20-22). If the current element is less than its parent, it's at the correct position and we're done (lines 22-25); otherwise, we swap the elements and continue up the tree (lines 26-30).

To demonstrate the code, let's walk through the above example:

```

Element | 8 | 6 | 7 | 5 | 3 | 2 | 4 |
Node   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |           // vector<int> v,
                                                 // arranged as a heap

          8(0)
          /   \
       6(1)   7(2)
      / \   / \
    5(3) 3(4) 2(5) 4(6)

v.push_back(9);

Element | 8 | 6 | 7 | 5 | 3 | 2 | 4 | 9 |
Node   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |           // vector<int> v,
                                                 // arranged as a heap

          8(0)
          /   \
       6(1)   7(2)
      / \   / \
    5(3) 3(4) 2(5) 4(6)
   /
  9(7)

pushHeap(v.begin(), v.end(), predicate);

```

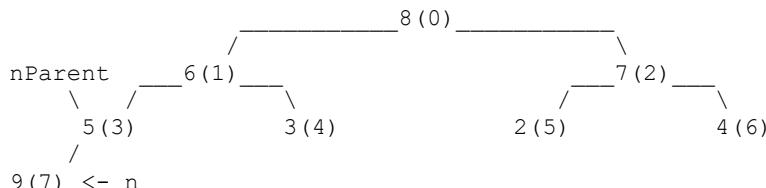
Initializer:

```
Node n = end - begin - 1;           // n = 7
```

Iteration 1:

```
Node nParent = ht::parent(n);           // nParent = 3
```

```
Element | 8 | 6 | 7 | 5 | 3 | 2 | 4 | 9 |
Node   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |           // vector<int> v,
                                                 // arranged as a heap
```

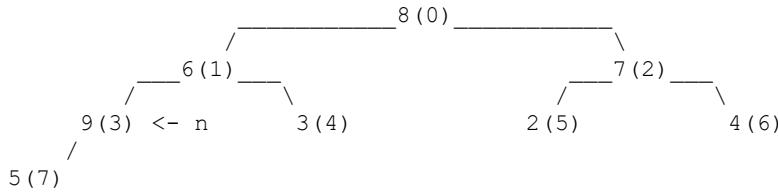


```

if (predicate(begin[n], begin[nParent]))           // Is 9 less than 5?
{
  break;
}
else
{
  std::swap(begin[n], begin[nParent]);           // Swap 9 and 5
  n = nParent;                                // n = 3
}

```

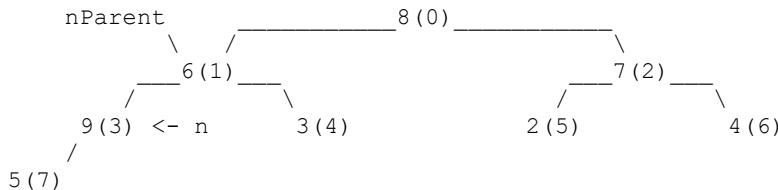
Element	8	6	7	9	3	2	4	5	
Node	0	1	2	3	4	5	6	7	



Iteration 2:

```
Node nParent = ht::parent(n); // nParent = 1
```

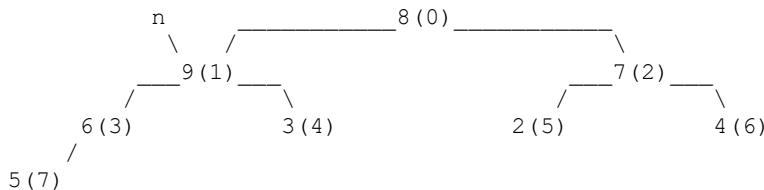
Element	8	6	7	9	3	2	4	5	
Node	0	1	2	3	4	5	6	7	



```

if (predicate(begin[n], begin[nParent])) // Is 9 less than 6?
{
    break;
}
else
{
    std::swap(begin[n], begin[nParent]); // Swap 9 and 6
    n = nParent; // n = 1
}
  
```

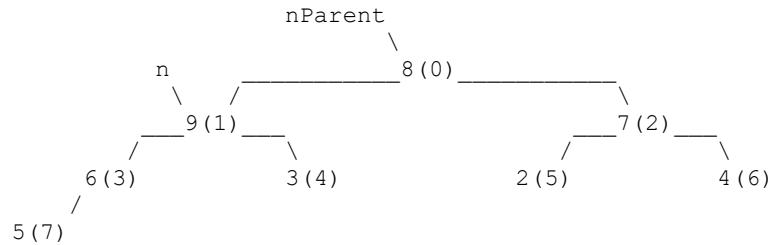
Element	8	9	7	6	3	2	4	5	
Node	0	1	2	3	4	5	6	7	



Iteration 3:

```
Node nParent = ht::parent(n); // nParent = 0
```

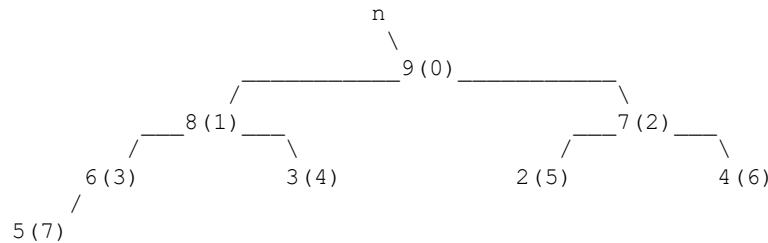
Element	8	9	7	6	3	2	4	5
Node	0	1	2	3	4	5	6	7



```

if (predicate(begin[n], begin[nParent]))      // Is 9 less than 8?
{
    break;
}
else
{
    std::swap(begin[n], begin[nParent]);        // Swap 9 and 8
    n = nParent;                                // n = 0
}
  
```

Element	9	8	7	6	3	2	4	5
Node	0	1	2	3	4	5	6	7



Terminate loop

The *heapTransform* header file (*main.cpp*, line 6) now provides *pushHeap* in addition to *makeHeap*. Our test program (lines 14-24) demonstrates the above example, generating the output

```

8 6 7 5 3 2 4          // makeHeap(v.begin(), v.end(), predicate);
8 6 7 5 3 2 4 9        // v.push_back(9);
9 8 7 6 3 2 4 5        // pushHeap(v.begin(), v.end(), predicate);
  
```

1.5: The *popHeap* Function

Source files and folders

- *heapTransform/3*
- *heapTransform/common/popHeap.h*

Chapter outline

- *Implementing a stand-alone function that pops an element from a heap*

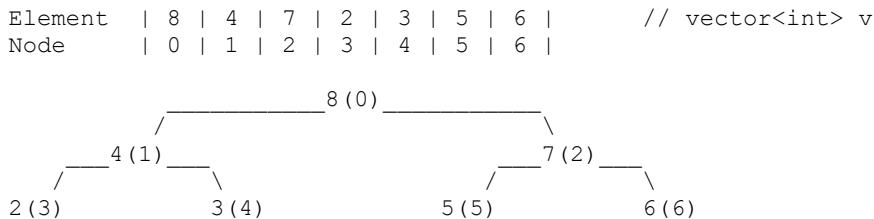
Recall from Chapter 1.2 that popping the top element from a heap consists of three main steps:

- Swap the top and bottom elements (the front and back elements of the underlying container)
- Move the top element down the tree (by comparing it to its largest child) until it's at the correct position
- Erase the bottom element

The function (*popHeap.h*, lines 10-11)

```
template <class RanIt, class Predicate>
void popHeap(RanIt begin, RanIt end, Predicate predicate);
```

performs the first two steps, after which we can remove the bottom element from the underlying container. The implementation is simple: after swapping the top and bottom elements (lines 18-20), we call *moveElementDownTree* (lines 22-25) to handle the second step. Given the heap

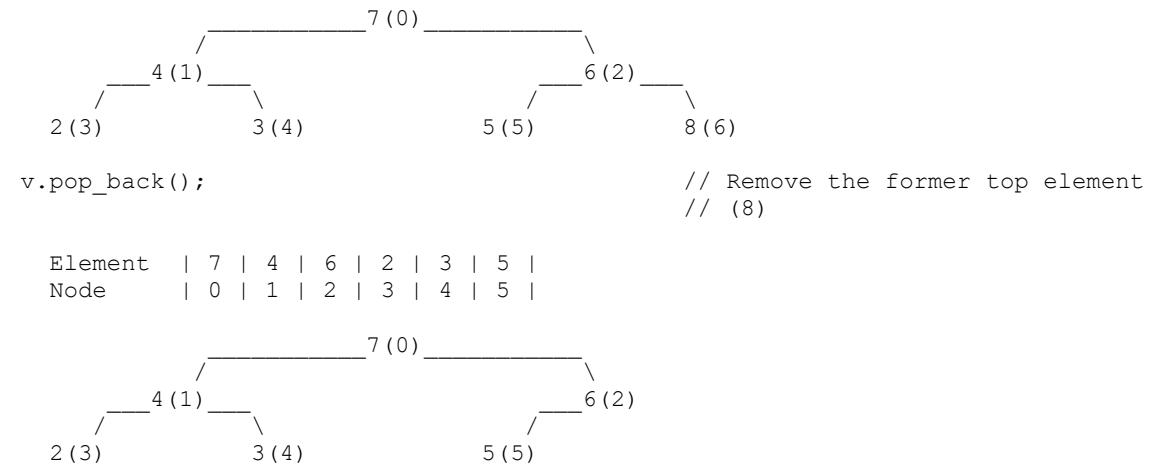


for example, we can pop the top element (8) by writing

```
popHeap(v.begin(), v.end(), less<int>()); // Swap elements 8 and 6, then
// move element 6 down to the
// correct position
```

after which *v* becomes

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



The *heapTransform* header file (*main.cpp*, line 6) now includes all the components we need to work with heaps using any type of random access container. Our test program (lines 8-25) demonstrates the above example, then continues popping the heap until it's empty. The resultant output is

```

8 4 7 2 3 5 6      // Original heap
7 4 6 2 3 5      // Pop 8
6 4 5 2 3      // Pop 7
5 4 3 2      // Pop 6
4 2 3      // Pop 5
3 2      // Pop 4
2      // Pop 3
    
```

The Standard Library versions of our *heapTransform* functions,

```

template <class RanIt, class Predicate>
void make_heap(RanIt begin, RanIt end, Predicate predicate);

template <class RanIt, class Predicate>
void push_heap(RanIt begin, RanIt end, Predicate predicate);

template <class RanIt, class Predicate>
void pop_heap(RanIt begin, RanIt end, Predicate predicate);
    
```

are provided by the *<algorithm>* header. The Standard Library version of our *Heap* class is called *priority_queue*, provided by the *<queue>* header:

```

template <class T,
         class Container = vector<T>,
         class Predicate = less<typename Container::value_type>>
class priority_queue
{
public:
    // Member types, push / pop methods, etc.,
    // identical to those of our Heap class

private:
    Container _container;      // As opposed to the _deque in our Heap class
    Predicate _predicate;
}

```

The *Container* template parameter allows a *priority_queue* to use any type of random access container to store the elements:

```

priority_queue<int> x;           // x._container is a vector<int>
priority_queue<int, array<int>> y; // y._container is an array<int>
priority_queue<int, deque<int>> z; // z._container is a deque<int>

```

Internally, *priority_queue*::*push* and *priority_queue*::*pop* manipulate the underlying container by simply calling *push_heap* and *pop_heap*. Although this implementation is more flexible than that of our *Heap* class, it's also a bit more complex, which is why we chose a more self-contained implementation before breaking it down into stand-alone functions.

1.6: Heap Sort

Source files and folders

- *heapSort*

Chapter outline

- *Implementing the heap sort algorithm*

The *makeHeap* and *popHeap* functions allow us to easily implement the *heap sort* algorithm, invented by John Williams in 1964. To perform a heap sort on a sequence S ,

- Rearrange S into a heap H
- Pop the top element from H and place it at the front of a new sequence N , until H is empty
- Once H is empty, N will contain the in-order sequence

If H is a max heap, N will be sorted in ascending order (least to greatest); if H is a min heap, N will be sorted in descending order (greatest to least). Consider, for example, the sequence

	<-----S----->									
Element	5	3	7	2	4	8	6			
Index	0	1	2	3	4	5	6			

To sort S in ascending order, we first rearrange it into a max heap H (via *makeHeap*), which puts the largest element (8) at the top:

	<-----H----->									
Element	8	-	-	-	-	-	-	-	-	
Index	0	1	2	3	4	5	6			
	begin								end	

// begin / end refer to the first / one-past-the-last elements of the heap (H)

All of the elements are currently part of the heap. For the sake of simplicity, we need not show the values of the remaining (non-top) elements; all that matters is that they form a heap. If we then call *popHeap* on H , it places element 8 at the back and element 7 at the top:

	<-----H-----> <-N->									
Element	7	-	-	-	-	-	-	-	8	
Index	0	1	2	3	4	5	6			
	begin								end	

Element 8, which is no longer part of the heap, is now the front of the new sequence N and the one-past-the-last element of H . If we call *popHeap* again on H , it places element 7 at the back of H (index 5) and element 6 at the top:

	<-----H----->						<----N---->		
Element	6	-	-	-	-	7	8		
Index	0	1	2	3	4	5	6		
	begin					end			

Element 7, which is no longer part of the heap, becomes the front of N and one-past-the-last element of H . If we call *popHeap* once again on H , element 6 ends up at the back (index 4) and element 5 rises to the top:

	<-----H----->						<----N---->		
Element	5	-	-	-	-	6	7	8	
Index	0	1	2	3	4	5	6		
	begin					end			

Element 6 is no longer part of the heap, and becomes the new front element of N . If we repeat the process until H is empty, N will eventually contain the entire sorted sequence:

popHeap (H);

	<-----H----->						<-----N----->		
Element	4	-	-	-	-	5	6	7	8
Index	0	1	2	3	4	5	6		
	begin					end			

popHeap (H);

	<-----H----->						<-----N----->		
Element	3	-	4	5	6	7	8		
Index	0	1	2	3	4	5	6		
	begin		end						

popHeap (H);

	<-H->						<-----N----->		
Element	2	3	4	5	6	7	8		
Index	0	1	2	3	4	5	6		
	begin	end							

popHeap (H);

	<-----N----->								
Element	2	3	4	5	6	7	8		
Index	0	1	2	3	4	5	6		
	begin								

The heap sort algorithm is implemented as the function (*heapSort.h*, lines 8-9)

```
template <class RanIt, class Predicate>
void heapSort(RanIt begin, RanIt end, Predicate predicate);
```

where *begin* and *end* bind the original sequence *S*. If we use a less-than *predicate*, *S* is rearranged into

a max heap, thereby sorting the final sequence (N) in ascending order; using a greater-than *predicate* will have the opposite effect.

We first rearrange S into the heap H (line 14), then call *popHeap* on H until H is empty (lines 16-20). After each iteration we decrement *end* (line 19), which shrinks H and expands N by a single element. When *begin* and *end* refer to the same element, H is empty so we're done.

Our test program (*main.cpp*) demonstrates the above example, generating the output

```
5 3 7 2 4 8 6      // v (original sequence)
2 3 4 5 6 7 8      // v (sorted)
```


Part 2: Selection Sort

2.1: Finding the Smallest and Largest Element

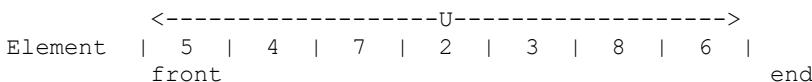
Source files and folders

- *findExtreme*

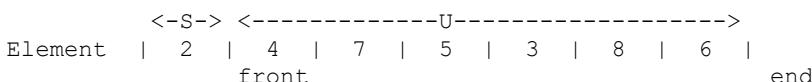
Chapter outline

- *An overview of the selection sort algorithm*
- *Finding the smallest (or largest) element of a given sequence*

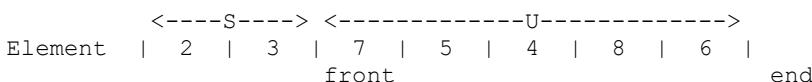
In this section we'll implement the *selection sort* algorithm, which is actually a more primitive (and less efficient) method than heap sort. With heap sort, the final sorted sequence grew from the back, one element at a time, until the unsorted sequence (heap) was empty. With selection sort, the final sorted sequence grows from the front. Consider, for example, the unsorted sequence U



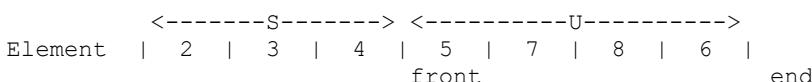
To sort U in ascending order, we first find the smallest element in U (2) then move it to the front by swapping it with the front element (5). This grows the sorted sequence S and shrinks the unsorted sequence U by 1 element:



We then find the smallest element in U (3) and swap it with the front of U (4). Elements 2 and 3 are now sorted:



We once again find the smallest element in U (4) and swap it with the front of U (7). Elements 2, 3, and 4 are now sorted:



If we repeat the process until U contains 1 element, all of the elements will be sorted:

```
// Find the smallest element in U (5) and swap it with the front of U (5)
```

Element		2		3		4		5		7		8		6		
										front						end

```
// Find the smallest element in U (6) and swap it with the front of U (7)
```

Element		2		3		4		5		6		8		7		
										front						end

```
// Find the smallest element in U (7) and swap it with the front of U (8)
```

Element		2		3		4		5		6		7		8		
										front					end	

```
// U now contains 1 element, which means that all the other elements are at
// their correct positions. The last element (8) is therefore also at its
// correct position.
```

To sort the sequence in descending order, we simply move the largest (rather than the smallest) element in each iteration. Most of the work therefore involves finding the smallest (or largest) element in U . The function (*findExtreme.h*, lines 6-7)

```
template <class Iter, class Predicate>
Iter findExtreme(Iter begin, Iter end, Predicate predicate);
```

returns an iterator to the “extreme” (smallest or largest) element in the given sequence. Using a less-than *predicate* will find the smallest element, while a greater-than *predicate* will find the largest.

The function traverses the entire sequence, keeping track of the smallest (or largest) element found so far. If we’re using a less-than predicate, *extreme* (line 12) points to the smallest element found so far, and *current* (line 14) points to the currently selected element. In each iteration, we compare the currently selected element to the smallest element found so far. If the current element is less than the smallest element found so far (line 15), we update *extreme* to point to the current element (line 16). When the traversal is complete, *extreme* will point to the smallest element. Given the sequence

Element		5		4		7		2		3		8		6		
		begin														end

for example, the function performs the following operations:

```
Iter extreme = begin; // e denotes extreme
```

Element		5		4		7		2		3		8		6		
										e						end

```

Iter current = ++begin;           // c denotes current

Element | 5 | 4 | 7 | 2 | 3 | 8 | 6 | end
        begin
        e
          c

Iteration 1:

if (predicate(*current, *extreme))      // Is 4 less than 5?
    extreme = current;                  // extreme points to element 4

++current;                            // current points to element 7

Element | 5 | 4 | 7 | 2 | 3 | 8 | 6 | end
        begin
        e
          c

Iteration 2:

if (predicate(*current, *extreme))      // Is 7 less than 4?
    extreme = current;

++current;                            // current points to element 2

Element | 5 | 4 | 7 | 2 | 3 | 8 | 6 | end
        begin
        e
          c

Iteration 3:

if (predicate(*current, *extreme))      // Is 2 less than 4?
    extreme = current;                  // extreme points to element 2

++current;                            // current points to element 3

Element | 5 | 4 | 7 | 2 | 3 | 8 | 6 | end
        begin
        e
          c

Iteration 4:

if (predicate(*current, *extreme))      // Is 3 less than 2?
    extreme = current;

++current;                            // current points to element 8

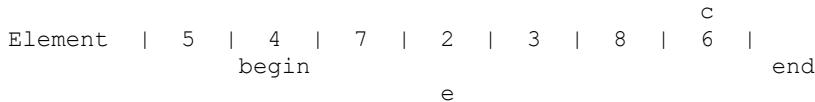
Element | 5 | 4 | 7 | 2 | 3 | 8 | 6 | end
        begin
        e
          c

```

Iteration 5:

```
if (predicate(*current, *extreme))      // Is 8 less than 2?
    extreme = current;

++current;                                // current points to element 6
```



Iteration 5:

```
if (predicate(*current, *extreme))      // Is 6 less than 2?
    extreme = current;

++current;                                // current is at the end

Element | 5 | 4 | 7 | 2 | 3 | 8 | 6 | end
        begin           e             c
                                         |
                                         e
```

// Terminate loop

```
return extreme;                           // Return an iterator to element 2
```

Our test program (*main.cpp*) demonstrates the above example, then uses a greater-than predicate to find the largest element. The resultant output is

```
5 4 7 2 3 8 6
The smallest element is 2
The largest element is 8
```

The Standard Library *<algorithm>* header provides the functions

```
template <class Iter, class Predicate>
Iter min_element(Iter begin, Iter end, Predicate predicate);

template <class Iter, class Predicate>
Iter max_element(Iter begin, Iter end, Predicate predicate);
```

which return iterators to the smallest / largest element in the given sequence. Their implementations are identical to that of our *findExtreme* function. The Standard Library also provides the overloads

```
template <class Iter>
Iter min_element(Iter begin, Iter end);

template <class Iter>
Iter max_element(Iter begin, Iter end);
```

which simply use the less-than / greater-than operators instead of a user-supplied predicate.

2.2: Completing the Implementation

Source files and folders

- ### - *selectionSort*

Chapter outline

- *Implementing the selectionSort function*

The `findExtreme` and `swap` functions allow us to easily implement selection sort. The procedure is

```
While the unsorted sequence U contains more than 1 element
{
    Find the smallest element in U and swap it with the front;

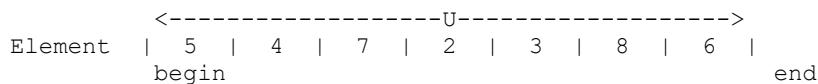
    Remove the front element from U and place it at the back of the new
        sorted sequence S;
}
```

The *selectionSort* function (*selectionSort.h*, lines 10-11),

```
template <class Iter, class Predicate>
void selectionSort(Iter begin, Iter end, Predicate predicate);
```

sorts the given sequence in ascending order (using a less-than *predicate*) or descending order (using a greater-than *predicate*). *back* (lines 18-19) points to the back element of U , and *front* (line 21) points to the front element of U . We traverse U until it contains 1 element, i.e. until *front* and *back* point to the same element. In each iteration, we find the smallest element in U and move it to the front via swapping (line 22). We then remove that element from U by pointing *front* to the next element.

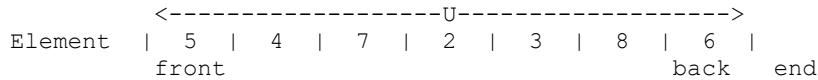
Consider, for example, the sequence



and suppose we're using a less-than *predicate*:

```
Iter back = end;  
--back;
```

```
Iter front = begin;
```



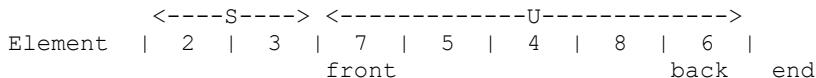
Iteration 1:

```
swap(*findExtreme(front, end, predicate), *front); // Swap 2 and 5
++front; // front points to 4
```

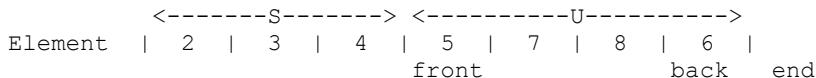


Iteration 2:

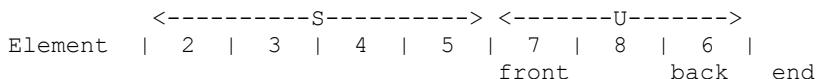
```
swap(*findExtreme(front, end, predicate), *front); // Swap 3 and 4  
++front; // front points to 7
```



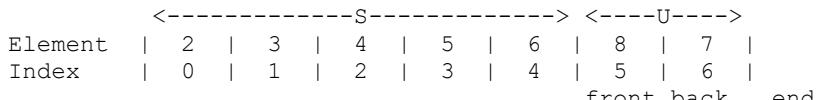
Iteration 3:



Iteration 4:



Iteration 5:



Iteration 6:

```
<-----S-----> <-U->
Element | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
Index   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
                           back   end
                           front
```

// U contains 1 element (front == back), so we're done

Our test program (*main.cpp*) demonstrates the above example, then sorts the sequence in descending order by using a greater-than predicate. The resultant output is

```
5 4 7 2 3 8 6      // Original sequence
2 3 4 5 6 7 8      // Ascending order (less-than predicate)
8 7 6 5 4 3 2      // Descending order (greater-than predicate)
```


Part 3: Shell Sort

3.1: Subsequence Sorting

Source files and folders

- *shellSort/I*
- *shellSort/common/sortSubsequence.h*

Chapter outline

- *An overview of the Shell sort algorithm*
- *Sorting individual subsequences within a given sequence*

The *Shell sort* algorithm, named after its inventor Donald Shell, improves the efficiency of the insertion sort algorithm we studied in Volume 1. Shell sort is similar to quick sort in that it divides the given sequence into smaller parts and sorts each one individually. Consider, for example, the sequence

21	20	17	12	14	18	16	11	22	19	15	13
----	----	----	----	----	----	----	----	----	----	----	----

We can break this down using a fixed *gap size* (number of elements), where a gap size of N elements produces a total of N subsequences. A gap size of 5 elements, for example, produces 5 subsequences:

21	--	--	--	--	18	--	--	--	--	15	--
--	20	--	--	--	--	16	--	--	--	--	13
--	--	17	--	--	--	--	11	--	--	--	--
--	--	--	12	--	--	--	--	22	--	--	--
--	--	--	--	14	--	--	--	--	19	--	--

Each subsequence is formed by taking every 5th element from the starting point: the first subsequence, {21, 18, 15}, consists of every 5th element beginning at 21, the second subsequence, {20, 16, 13}, consists of every 5th element beginning at 20, etc.

We then sort each subsequence, using any algorithm we want; in this implementation we'll use insertion sort. Performing an insertion sort on the first subsequence, we have

21	--	--	--	--	18	--	--	--	--	15	--
<i>Swap 18 and 21</i>											
18	--	--	--	--	21	--	--	--	--	15	--
<i>Swap 15 and 21</i>											
18	--	--	--	--	15	--	--	--	--	21	--
<i>Swap 15 and 18</i>											
15	--	--	--	--	18	--	--	--	--	21	--

15	--	--	--	--	18	--	--	--	--	21	--
----	----	----	----	----	----	----	----	----	----	----	----

After sorting each subsequence, we end up with

15	--	--	--	--	18	--	--	--	--	21	--
--	13	--	--	--	--	16	--	--	--	--	20
--	--	11	--	--	--	--	17	--	--	--	--
--	--	--	12	--	--	--	--	22	--	--	--
--	--	--	--	14	--	--	--	--	19	--	--

and the entire sequence becomes

15	13	11	12	14	18	16	17	22	19	21	20
----	----	----	----	----	----	----	----	----	----	----	----

Although we aren't done yet, the table below shows that many of the elements are now much closer to their correct positions:

Original sequence													
21	20	17	12	14	18	16	11	22	19	15	13		
After subsequence sorting (gap size = 5)													
15	13	11	12	14	18	16	17	22	19	21	20		
Correct positions (fully sorted sequence)													
11	12	13	14	15	16	17	18	19	20	21	22		
Element	Distance to correct position (Original sequence)						Distance to correct position (After subsequence sorting)						
11	7						2						
13	9						1						
15	6						4						
17	4						1						
20	8						2						
21	10						0						

In the original sequence, for example, element 11 is 7 elements away from its correct position. With a traditional insertion sort, we would therefore need to perform 7 consecutive swaps to move it there. Our subsequence sort, however, covered most of that distance in just 1 swap. That's the advantage of subsequence sorting: it can reduce the total number of swaps required compared to a traditional insertion sort.

We then repeat the process, using a progressively smaller gap size each time. We can use any series of gap sizes, as long as the final gap size is 1. Proceeding with the current example, let's use a gap size of 3, which produces 3 subsequences:

15	--	--	12	--	--	16	--	--	19	--	--	
--	13	--	--	14	--	--	17	--	--	21	--	
--	--	11	--	--	18	--	--	22	--	--	20	

After sorting each subsequence, we end up with

12	--	--	15	--	--	16	--	--	19	--	--
--	13	--	--	14	--	--	17	--	--	21	--
--	--	11	--	--	18	--	--	20	--	--	22

and the entire sequence becomes

12	13	11	15	14	18	16	17	20	19	21	22
----	----	----	----	----	----	----	----	----	----	----	----

All of the elements are now at most 2 elements away from their correct positions; most, in fact, are only 1 element away:

After subsequence sorting (gap size = 3)												
12	13	11	15	14	18	16	17	20	19	21	22	
Correct positions (fully sorted sequence)												
11	12	13	14	15	16	17	18	19	20	21	22	
Element	Distance to correct position (After subsequence sorting)											
11	2											
12	1											
13	1											
14	1											
15	1											
16	1											
17	1											
18	2											
19	1											
20	1											
21	0											
22	0											

After the final subsequence sort, which always uses a gap size of 1, each element will be at the correct position. The question then becomes, what is the optimal series of gap sizes? We'll discuss that later, but for now let's implement subsequence sorting. As mentioned earlier, we'll use insertion sort as the underlying algorithm. The function (*sortSubsequence.h*, lines 16-20)

```
template <class RanIt, class Predicate>
void moveElement(RanIt element,
                 RanIt subsequenceBegin,
                 typename RanIt::difference_type gapSize,
                 Predicate predicate);
```

moves the given *element* to its correct position in the subsequence. *subsequenceBegin* is the first element of the subsequence, *gapSize* is the distance between each element, and *predicate* is the function object used to compare elements.

The function is defined in lines 39-59. Consider, for example, the subsequence

```
-- 21 -- -- 14 -- -- 17 -- -- 13 --
```

To move element 13 to its correct position in the subsequence, we perform the following operations:

```
// element points to 13
// subsequenceBegin points to 21
// gapSize is 3
// Assume we're using a less-than predicate
```

```
SB
-- 21 -- -- 14 -- -- 17 -- -- 13 --
E
```

```
// E = element
// SB = subsequenceBegin
```

Iteration 1:

```
RanIt leftNeighbor = element - gapSize;
```

```
SB
-- 21 -- -- 14 -- -- 17 -- -- 13 --
LN E
```

```
// LN = leftNeighbor (the next element to the left of the current element)
```

```
if (predicate(*element, *leftNeighbor)) // Is 13 less than 17?
{
    std::swap(*element, *leftNeighbor); // Swap 13 and 17
    element = leftNeighbor; // Proceed to the next element
}
else
{
    break;
}
```

```
SB
-- 21 -- -- 14 -- -- 13 -- -- 17 --
E
```

Iteration 2:

```
RanIt leftNeighbor = element - gapSize;
```

```
SB
-- 21 -- -- 14 -- -- 13 -- -- 17 --
LN E
```

```

if (predicate(*element, *leftNeighbor))           // Is 13 less than 14?
{
    std::swap(*element, *leftNeighbor);          // Swap 13 and 14
    element = leftNeighbor;                     // Proceed to the next element
}
else
{
    break;
}

```

	SB											
--	21	--	--	13	--	--	14	--	--	17	--	
				E								

Iteration 3:

```
RanIt leftNeighbor = element - gapSize;
```

	SB											
--	21	--	--	13	--	--	14	--	--	17	--	
				LN	E							

```

if (predicate(*element, *leftNeighbor))           // Is 13 less than 21?
{
    std::swap(*element, *leftNeighbor);          // Swap 13 and 21
    element = leftNeighbor;                     // Proceed to the next element
}
else
{
    break;
}

```

	SB											
--	13	--	--	21	--	--	14	--	--	17	--	
				E								

// We've reached the beginning of the subsequence (element and
// subsequenceBegin point to the same element), so terminate the loop

// Element 13 is now at its correct position in the subsequence

To sort an entire subsequence, we simply call *moveElement* for each element in the subsequence (except for the first element, which can't be moved leftward at all). To sort the subsequence

	21	--	--	14	--	--	17	--	--	13	--
--	----	----	----	----	----	----	----	----	----	----	----

for example, we call *moveElement* on 14, 17, then 13. The function (lines 10-14)

```

template <class RanIt, class Predicate>
void sortSubsequence(RanIt subsequenceBegin,
RanIt sequenceEnd,
typename RanIt::difference_type gapSize,
Predicate predicate);

```

does exactly that. Note, however, that *sequenceEnd* refers to the end (one-past-the-last element) of the *whole* sequence, not the subsequence:

```

SSB                               SE
-- 21  --  -- 14  --  -- 17  --  -- 13  --
// SSB = subsequenceBegin (first element of the subsequence)
// SE = sequenceEnd (one-past-the-last element of the whole sequence)

```

To sort the above subsequence, we perform 3 iterations (lines 28-36):

```

// gapSize is 3

RanIt element = subsequenceBegin + gapSize;

SSB                               SE
-- 21  --  -- 14  --  -- 17  --  -- 13  --
E

// E = element

Iteration 1:

moveElement(element,           // Move 14 leftward to the correct spot
            subsequenceBegin,
            gapSize,
            predicate);

SSB                               SE
-- 14  --  -- 21  --  -- 17  --  -- 13  --
E

element += gapSize;

SSB                               SE
-- 14  --  -- 21  --  -- 17  --  -- 13  --
E

Iteration 2:

moveElement(element,           // Move 17 leftward to the correct spot
            subsequenceBegin,
            gapSize,
            predicate);

SSB                               SE
-- 14  --  -- 17  --  -- 21  --  -- 13  --
E

element += gapSize;

```

```

SSB                                     SE
-- 14 -- -- 17 -- -- 21 -- -- 13 --  

                                         E

Iteration 3:

moveElement(element,           // Move 13 leftward to the correct spot
            subsequenceBegin,
            gapSize,
            predicate);

SSB                                     SE
-- 13 -- -- 14 -- -- 17 -- -- 21 --  

                                         E

element += gapSize;

SSB                                     SE
-- 13 -- -- 14 -- -- 17 -- -- 21 --  

                                         E

// element is not less than sequenceEnd, so terminate the loop
// The entire subsequence is now sorted

```

We can now easily sort each subsequence, using whatever gap size we want. Our test program (*main.cpp*) demonstrates this on the sequence

```
21 20 17 12 14 18 16 11 22 19 15 13
```

Using a gap size of 3, the subsequences are

```
21 -- -- 12 -- -- 16 -- -- 19 -- --  

-- 20 -- -- 14 -- -- 11 -- -- 15 --  

-- -- 17 -- -- 18 -- -- 22 -- -- 13
```

and the loop (lines 19-27) performs 3 iterations:

- Iteration 1 ($s = 0$) sorts the subsequence beginning at element 21 ($v.begin() + 0$)
- Iteration 2 ($s = 1$) sorts the subsequence beginning at element 20 ($v.begin() + 1$)
- Iteration 3 ($s = 2$) sorts the subsequence beginning at element 17 ($v.begin() + 2$)

The resultant output is

```
21 20 17 12 14 18 16 11 22 19 15 13 // Original sequence
12 20 17 16 14 18 19 11 22 21 15 13 // Subsequence 0 {12,16,19,21} sorted
12 11 17 16 14 18 19 15 22 21 20 13 // Subsequence 1 {11,14,15,20} sorted
12 11 13 16 14 17 19 15 18 21 20 22 // Subsequence 2 {13,17,18,22} sorted
```


3.2: Choosing a Series of Gap Sizes

Source files and folders

- *shellSort/2*
- *shellSort/common/generateGapSizes.h*
- *shellSort/common/sortAllSubsequences.h*

Chapter outline

- *Sorting all the subsequences of a particular gap size*
- *Generating a series of gap sizes using Tokuda's sequence*

Now that we've written *sortSubsequence*, the next step to implementing Shell sort is having a way to sort all the subsequences of a particular gap size. The function (*sortAllSubsequences.h*, lines 10-14)

```
template <class RanIt, class Predicate>
void sortAllSubsequences(RanIt sequenceBegin,
    RanIt sequenceEnd,
    typename RanIt::difference_type gapSize,
    Predicate predicate);
```

does exactly that, where *sequenceBegin* and *sequenceEnd* mark the beginning and end of the entire sequence. The function simply executes the loop from the previous chapter's test program, calling *sortSubsequence* on each starting element (*sortAllSubsequences.h*, lines 22-26). Given a *vector v*

```
21      20      17      12      14      18      16      11      22      19      15      13
```

for example, the function call

```
sortAllSubsequences(v.begin(),
    v.end(),
    3,
    less<int>());
```

sorts all the subsequences of gap size 3:

```
21      --      --      12      --      --      16      --      --      19      --      --
--      20      --      --      14      --      --      11      --      --      15      --
--      --      17      --      --      18      --      --      22      --      --      13
```

Performing a complete Shell sort is now fairly simple: we repeatedly call *sortAllSubsequences*, using a descending series of gap sizes that ends with 1. Unfortunately, however, there is no single “correct” series of gap sizes, though many have been researched and proposed. In this implementation we'll use *Tokuda's sequence*, a series named after its inventor Naoyuki Tokuda. The first number in the sequence (G_1) is 1, and the n^{th} number (G_n) is $(2.25G_{n-1} + 1)$:

$$\begin{aligned}
 G_1 &= 1 \\
 G_2 &= 2.25G_{2-1} + 1 = 2.25(1) + 1 &= 3.25 \\
 G_3 &= 2.25G_{3-1} + 1 = 2.25(3.25) + 1 &= 8.3125 \\
 G_4 &= 2.25G_{4-1} + 1 = 2.25(8.3125) + 1 &= 19.703125
 \end{aligned}$$

We'll use the ceiling of each value for the corresponding gap size:

$$\{\text{ceil}(G_1), \text{ceil}(G_2), \text{ceil}(G_3), \text{ceil}(G_4), \dots\} = \{1, 4, 9, 20, \dots\}$$

The starting gap size depends on the total number of elements: in an N -element sequence, any gap size greater than or equal to N will have no effect. In a 5-element sequence, for example, any gap size greater than or equal to 5 won't do anything:

Original sequence (5 elements):

21 20 17 12 14

Subsequences formed by a gap size of 5 or more:

21	--	--	--	--	// None of these subsequences
--	20	--	--	--	// can be sorted because they
--	--	17	--	--	// only contain 1 element
--	--	--	12	--	
--	--	--	--	14	

Subsequence sorting, in other words, only works when the gap size is smaller than the total number of elements. We'll therefore start with the largest Tokuda value that is smaller than the total number of elements and work our way down to 1. Given a 12-element sequence, for example, we would use the gap sizes $\{9, 4, 1\}$; for a 25-element sequence, we'd use $\{20, 9, 4, 1\}$.

The function (*generateGapSizes.h*, lines 11-12)

```
template <class Size>
void generateGapSizes(std::vector<Size>* gapSizes, Size totalElements);
```

generates a series of gap sizes (Tokuda values). The template parameter *Size* is the type used to represent gap size / number of elements. *gapSizes* points to an empty *vector* that will store the series, and *totalElements* is the total number of elements to be sorted.

We start the series with a gap size of 1 (line 17). *g* (line 19) represents the current Tokuda value (G_n). In each iteration of the loop, we calculate G_n (line 23), then convert it to the actual gap size by taking the ceiling and explicitly converting that value to an integer (line 25). If the actual gap size is less than the total number of elements, we add it to the series; otherwise, we're done (lines 27-30).

Our test program (*main.cpp*) demonstrates *generateGapSizes* for a 5-element sequence (lines 14-18), followed by a 35-element sequence (lines 20-24). Note that because we stored the gap sizes using *push_back*, we need to traverse the vector in reverse order (back-to-front). The resultant output is

Gap sizes for a 5-element sequence:

4 1

Gap sizes for a 35-element sequence:

20 9 4 1

3.3: Completing the Implementation

Source files and folders

- *shellSort/3*

Chapter outline

- *Implementing the shellSort function*

We now have everything we need to complete our Shell sort implementation. The procedure is

```
Generate a series of gap sizes;
For each gap size in the series,
    Sort all the subsequences of that gap size;
```

The *shellSort* function is (*shellSort.h*, lines 11-12)

```
template <class RanIt, class Predicate>
void shellSort(RanIt begin, RanIt end, Predicate predicate);
```

GapSize (line 19) is an alias of the iterator's *difference_type*, and the expression (line 23)

```
end - begin
```

gets us the total number of elements in the sequence. Because the vector contains the gap sizes in ascending order, we need to traverse it backwards (in descending order). Line 25,

```
for (auto g = gapSizes.crbegin(); g != gapSizes.crend(); ++g)
```

is equivalent to

```
for (vector<GapSize>::const_reverse_iterator g = gapSizes.rbegin();
     g != gapSizes.rend();
     ++g)
```

The *crbegin / crend* methods, introduced by the C++11 Standard, return *const_reverse_iterators*. The *auto* keyword, also introduced in C++11, tells the compiler to automatically deduce *g*'s type: the compiler knows the return type of *crbegin (vector<GapSize>::const_reverse_iterator)*, so it automatically uses that as the type for *g*.

Our test program (*main.cpp*) sorts a 12-element sequence in ascending and descending order, generating the output

```
21 20 17 12 14 18 16 11 22 19 15 13      // Original sequence
11 12 13 14 15 16 17 18 19 20 21 22      // Sorted (ascending order)
22 21 20 19 18 17 16 15 14 13 12 11      // Sorted (descending order)
```

In addition to the *crbegin* / *crend* methods, C++11 introduced the *cbegin* / *cend* methods, which return const iterators. These methods, along with automatic type deduction, allow us to avoid writing out the full type names:

```
vector<int> v = {1, 3, 5};

auto i = v.begin();           // i is a vector<int>::iterator
auto k = v.cbegin();         // k is a vector<int>::const_iterator

auto x = v.rbegin();          // x is a vector<int>::reverse_iterator
auto y = v.crbegin();        // y is a vector<int>::const_reverse_iterator
```

They also give us more precise control when passing iterators to generic functions. Consider, for example, the function

```
template <class Iter>                                // Print all elements in the
void printSequence(Iter begin, Iter end)             // set [begin, end)
{
    while (begin != end)                            // Doesn't modify any elements
    {
        std::cout << *begin << ' ';
        ++begin;
    }
}
```

Using the above vector *v*, if we make the call

```
printSequence(v.begin(), v.end());
```

we're actually passing *vector<int>::iterators*. *v* is a non-*const* object, so the compiler selects the non-*const* overloads of *begin* and *end*. The above call, in other words, is equivalent to

```
printSequence<vector<int>::iterator>(v.begin(), v.end());
```

That's okay, but ideally we shouldn't pass non-*const* iterators around if we don't actually intend to modify any elements. Furthermore, if *printSequence* mistakenly contained any element-modifying code, the compiler wouldn't catch it.

Before C++11, if we wanted to pass *const_iterators* to *printSequence*, we would've had to include the template argument, as in

```
printSequence<vector<int>::const_iterator>(v.begin(), v.end());
```

Using *cbegin* and *cend*, however, we can simply write

```
printSequence(v.cbegin(), v.cend());
```

Part 4: Merge Sort

4.1: Splitting a Sequence in Half

Source files and folders

- *mergeSort/I*
- *mergeSort/common/split.h*

Chapter outline

- *An overview of the merge sort algorithm*
- *Getting an iterator to the midpoint of a sequence*

The *merge sort* algorithm, invented by John von Neumann, seeks to improve upon quick sort. Recall that quick sort recursively partitions a sequence until all the subsequences contain 1 or 0 elements, at which point the entire sequence is sorted. To partition a sequence, we choose a *pivot* element, then rearrange the sequence such that all elements less than the pivot are to the left of the pivot, and all elements not less than the pivot are to the right of the pivot. Given the sequence

5 7 6 1 3 2 4

for example, suppose that we use the back element, 4, as the pivot. Partitioning results in

```

pivot
|
1 3 2 4 7 6 5      // The pivot element, 4, is now at the correct position:
|__|   |__|      // All elements less than 4 have been moved to the left
|   |   |      // of 4, and all elements not less than 4 have been moved
left    right      // to the right of 4
subseq  subseq      // We then recursively partition the left and right
                     // subsequences, {1 3 2} and {7 6 5}...

```

Partitioning is most effective when the resulting subsequences are of equal size. In the above example, our chosen pivot (4) happened to evenly divide the sequence, but that won't always be the case. Given the sequence

5 7 6 4 3 2 1

for example, if we again use the back element (1) as the pivot, partitioning results in

```

pivot
|
1 5 7 6 4 3 2      // The left and right subsequences, {} and {5 7 6 4 3 2},
|_| |_____|
|   |           |
left    right
subseq  subseq

```

which offers no improvement over a simple insertion sort of element 1.

Merge sort specifically address this problem. Rather than use arbitrarily chosen pivots, merge sort recursively splits the original sequence in half until each subsequence contains 1 element, then merges (recombines) the subsequences into a fully sorted sequence. The procedure is

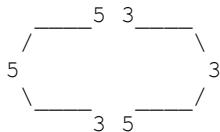
```

Sort(Sequence S)
{
  If S contains more than 1 element
  {
    Sort(The left half of S);
    Sort(The right half of S);

    Merge(The sorted halves);
  }
}

```

Consider, for example, the sequence {5 3}. The following diagram and pseudocode illustrate the recursion:



```

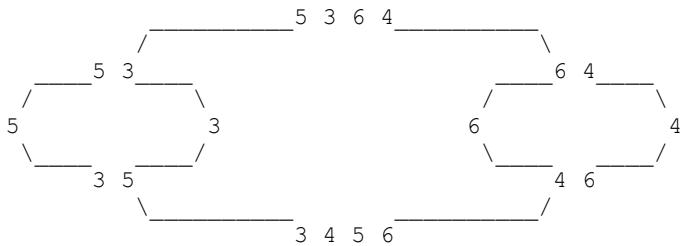
Sort(5 3)                                // Sort S
{
  Sort(5)                                 // Sort the left half of S
  {
    return;                               // A 1-element sequence, by definition,
  }                                       // is already sorted

  Sort(3)                                 // Sort the right half of S
  {
    return;                               // A 1-element sequence, by definition,
  }                                       // is already sorted

  Merge (5) and (3) into (3 5);          // Merge the sorted halves into a fully
}                                         // sorted sequence

```

To sort the sequence {5 3 6 4}:



```

Sort(5 3 6 4)                                // Sort S
{
  Sort(5 3)                                    // Sort the left half of S
  {
    Sort(5)                                     // Sort the left half of
    {                                           // the left half of S
      return;
    }
    Sort(3)                                     // Sort the right half of
    {                                           // the left half of S
      return;
    }

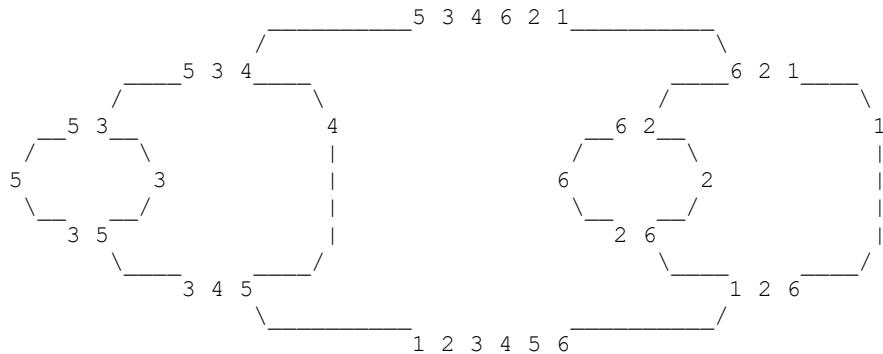
    Merge (5) and (3) into (3 5);             // The left half of S is now
  }                                             // fully sorted

Sort(6 4)                                      // Sort the right half of S
{
  Sort(6)                                       // Sort the left half of
  {                                             // the right half of S
    return;
  }
  Sort(4)                                       // Sort the right half of
  {                                             // the left half of S
    return;
  }

  Merge (6) and (4) into (4 6);             // The right half of S is now
}                                             // fully sorted

Merge (3 5) and (4 6) into (3 4 5 6);       // S is now fully sorted
}
  
```

To sort the sequence {5 3 4 6 2 1}:



```

Sort(5 3 4 6 2 1)
{
    Sort(5 3 4)
    {
        Sort(5 3)
        {
            Sort(5)
            {
                return;
            }
            Sort(3)
            {
                return;
            }

            Merge (5) and (3) into (3 5);
        }
        Sort(4)
        {
            return;
        }

        Merge (3 5) and (4) into (3 4 5);
    }

    Sort(6 2 1)
    {
        Sort(6 2)
        {
            Sort(6)
            {
                return;
            }
            Sort(2)
            {
                return;
            }

            Merge (6) and (2) into (2 6);
        }
    }
}

```

```

Sort(1)
{
    return;
}

Merge (2 6) and (1) into (1 2 6);
}

Merge (3 4 5) and (1 2 6) into (1 2 3 4 5 6);
}

```

To implement merge sort, we first need a way to split a sequence in half. As the above example shows, we'll split any odd-sized sequences by putting the extra element in the left subsequence. Given a 5-element sequence such as {1 3 5 7 9}, for example, we'll split it into {1 3 5} and {7 9}, as opposed to {1 3} and {5 7 9}. The function (*split.h*, lines 10-11)

```

template <class RanIt>
RanIt split(RanIt begin, RanIt end);

```

returns an iterator to the midpoint of the given sequence, where “midpoint” is defined as the first element of the right subsequence. *Distance* (line 16) is an alias of the iterator's *difference_type*, and *midpointDistance* (line 18) is the distance (number of elements) from the first element to the midpoint. In a 4-element sequence, for example, the midpoint is 2 elements away from the first element:

```

midpointDistance = static_cast<Distance>(ceil((end - begin) / 2.0));
= static_cast<Distance>(ceil(4 / 2.0));
= static_cast<Distance>(ceil(2));
= static_cast<Distance>(2);
= 2;

```

In a 5-element sequence, the midpoint is 3 elements away from the first element:

```

midpointDistance = static_cast<Distance>(ceil((end - begin) / 2.0));
= static_cast<Distance>(ceil(5 / 2.0));
= static_cast<Distance>(ceil(2.5));
= static_cast<Distance>(3);
= 3;

```

Our test program (*main.cpp*) demonstrates this by splitting *x* (a 4-element sequence) and *y* (a 5-element sequence). *xMid* (line 18) refers to the midpoint of *x* (element 6), and *yMid* (line 27) refers to the midpoint of *y* (element 7). The resultant output is

```

2 4 6 8      // [x.begin(), x.end()), the whole sequence
2 4            // [x.begin(), xMid), the left subsequence
6 8            // [xMid, x.end()), the right subsequence

1 3 5 7 9      // [y.begin(), y.end()), the whole sequence
1 3 5          // [y.begin(), yMid), the left subsequence
7 9            // [yMid, y.end()), the right subsequence

```


4.2: Merging Sorted Halves

Source files and folders

- *mergeSort/2*
- *mergeSort/common/merge.h*

Chapter outline

- *Combining the two sorted halves of a sequence into a fully sorted sequence*

Now that we can split a sequence by taking the midpoint, the next step is implementing the merge operation. Suppose, for example, that we have a sequence that's split into two sorted halves:

```
3 4 5 1 2 6
B       M       E
```

```
// B = begin (The beginning of the whole sequence, {3 4 5 1 2 6}, and the
// beginning of the left subsequence, {3 4 5})
```

```
// M = midpoint (The first element of the right subsequence, {1 2 6})
// E = end (The end of the whole sequence)
```

The merge operation combines the two sorted subsequences, {3 4 5} and {1 2 6}, into one fully sorted sequence, {1 2 3 4 5 6}. Let x be the first element of the left subsequence, and let y be the first element of the right subsequence. We'll also need a temporary buffer with enough room for the sorted elements:

```
x      y
3 4 5 1 2 6
B       M       E
-
- - - - -
Buffer (6-element capacity)
```

To perform the merge, we simultaneously traverse the left and right subsequences, comparing the current elements x and y :

- If x is less than y , we copy x to the buffer and point to the next element in the left subsequence
- Otherwise, we copy y to the buffer and point to the next element in the right subsequence

Upon reaching the end of either subsequence, we copy the remaining elements to the buffer, then overwrite the original sequence with the contents of the buffer. Let's walk through the above example:

Iteration 1:

`x` is not less than `y`,
 so copy `y` to the buffer and point to the next element in the right
 subsequence;

<code>x</code>	<code>y</code>
3 4 5 1 2 6	
B	M
	E

1 - - - -
 Buffer

Iteration 2:

`x` is not less than `y`,
 so copy `y` to the buffer and point to the next element in the right
 subsequence;

<code>x</code>	<code>y</code>
3 4 5 1 2 6	
B	M
	E

1 2 - - -
 Buffer

Iteration 3:

`x` is less than `y`,
 so copy `x` to the buffer and point to the next element in the left
 subsequence;

<code>x</code>	<code>y</code>
3 4 5 1 2 6	
B	M
	E

1 2 3 - -
 Buffer

Iteration 4:

`x` is less than `y`,
 so copy `x` to the buffer and point to the next element in the left
 subsequence;

<code>x</code>	<code>y</code>
3 4 5 1 2 6	
B	M
	E

1 2 3 4 - -
 Buffer

Iteration 5:

```
x is less than y,
so copy x to the buffer and point to the next element in the left
subsequence;
```

	x	y
3 4 5 1 2 6		
B	M	E

1 2 3 4 5 -
Buffer

We've reached the end of the left subsequence (x is at the midpoint), so copy the remaining elements in the right subsequence to the buffer:

Iteration 1:

Copy y to the buffer and point to the next element in the right subsequence;

	x	y
3 4 5 1 2 6		
B	M	E

1 2 3 4 5 6
Buffer

We've reached the end of the right subsequence (y is at the end), so the buffer contains all of the sorted elements.

We then overwrite the original sequence, [begin, end) with the contents of the buffer:

1 2 3 4 5 6
B M E

1 2 3 4 5 6
Buffer

Before writing the *merge* function, let's walk through another example:

```
x      y          // The two sorted halves are {2 5 6} (left
2 5 6 1 3 4          // subsequence) and {1 3 4} (right subsequence)
B      M      E

- - - - -
Buffer (6-element capacity)
```

Iteration 1:

```
x is not less than y,
so copy y to the buffer and point to the next element in the right
subsequence;
```

x		y
2 5 6 1 3 4		
B	M	E

1 - - - -
Buffer

Iteration 2:

x is less than **y**,
so copy **x** to the buffer and point to the next element in the left subsequence;

x		y
2 5 6 1 3 4		
B	M	E

1 2 - - -
Buffer

Iteration 3:

x is not less than **y**,
so copy **y** to the buffer and point to the next element in the right subsequence;

x		y
2 5 6 1 3 4		
B	M	E

1 2 3 - - -
Buffer

Iteration 4:

x is not less than **y**,
so copy **y** to the buffer and point to the next element in the right subsequence;

x		y
2 5 6 1 3 4		
B	M	E

1 2 3 4 - -
Buffer

We've reached the end of the right subsequence (**y** is at the end), so copy the remaining elements in the left subsequence to the buffer:

Iteration 1:

Copy **x** to the buffer and point to the next element in the left subsequence;

	x	
2 5 6 1 3 4		Y
B	M	E

1 2 3 4 5 -
Buffer

Iteration 2:

Copy *x* to the buffer and point to the next element in the left subsequence;

	x	
2 5 6 1 3 4		Y
B	M	E

1 2 3 4 5 6
Buffer

We've reached the end of the left subsequence (*x* is at the end), so the buffer contains all of the sorted elements.

We then overwrite the original sequence, [begin, end) with the contents of the buffer:

1 2 3 4 5 6
B M E

1 2 3 4 5 6
Buffer

Our *merge* function is (*merge.h*, lines 10-14)

```
template <class RanIt, class Predicate>
void merge(RanIt begin,
           RanIt midpoint,
           RanIt end,
           Predicate predicate);
```

where *begin* / *end* point to the first / one-past-the-last elements of the whole sequence, and *midpoint* refers to the beginning (first element) of the right subsequence.

We begin by initializing the buffer, which is simply a *vector* with the same capacity as the total number of elements in our sequence (lines 22-25).

Lines 27-28 initialize *x* and *y* to the beginning of the left and right subsequences, respectively. We then simultaneously traverse the left and right subsequences, until we've reached the end of either one (line 30). In each iteration of the loop,

- If *x* is less than *y*, we copy *x* to the buffer and point to the next element in the left subsequence (lines 32-33)

- Otherwise, we copy y to the buffer and point to the next element in the right subsequence (lines 34-35)

Once the loop terminates, we copy the remaining elements to the buffer (lines 38-42):

- If we've already copied all the elements from the right subsequence, then only the first loop will perform any iterations (copying the remaining elements from the left subsequence) (lines 38-39)
- If we've already copied all the elements from the left subsequence, then only the second loop will perform any iterations (copying the remaining elements from the right subsequence) (lines 41-42)

Finally, we overwrite the original sequence with the contents of the buffer by simply assigning the value of each element in the buffer to the corresponding element in the original sequence (lines 44-45):

```
for (const Element& e : buffer)
    *begin++ = e;
```

This is an example of a *range-based for loop*, another C++11 feature designed to simplify code. It traverses the sequence [buffer.begin(), buffer.end()), and in each iteration, e is a *const* reference to the current element. This is equivalent to the traditional *for* loop

```
for (std::vector<Element>::const_iterator e = buffer.begin();
     e != buffer.end();
     ++e)
{
    *begin++ = *e;
}
```

With range-based *for* loops, we can also access elements by non-*const* reference, or by value:

```
for (Element& e : buffer)
{
    // e is a non-const reference to the current element
}

for (Element e : buffer)
{
    // e is a separate, temporary copy of the current element
}
```

That second range-based *for* loop is equivalent to something like

```

for (std::vector<Element>::const_iterator i = buffer.begin();
     i != buffer.end();
     ++i;
{
    Element e = *i;      // The current element (e) is a separate, temporary
                          // copy of the one in the underlying container (buffer)
}

```

We can also use automatic type deduction instead of writing out the typename:

```

for (const auto& e : buffer)      // for (const Element& e : buffer)
{
    // ...
}

for (auto& e : buffer)           // for (Element& e : buffer)
{
    // ...
}

for (auto e : buffer)            // for (Element e : buffer)
{
    // ...
}

```

Our test program (*main.cpp*) demonstrates the example from the beginning of the chapter. We begin by constructing the sequence {3 4 5 1 2 6}, in which the two halves, {3 4 5} and {1 2 6}, are sorted. We then merge the two sorted halves, generating the output

```

3 4 5 1 2 6    // Original sequence x
1 2 3 4 5 6    // x after merging

```


4.3: Completing the Implementation

Source files and folders

- *mergeSort/3*

Chapter outline

- *Implementing the mergeSort function*

We now have everything we need to complete our merge sort implementation. Recall the procedure from Chapter 4.1:

```
Sort(Sequence S)
{
    If S contains more than 1 element
    {
        Sort(The left half of S);           // Recursive call
        Sort(The right half of S);         // Recursive call

        Merge(The sorted halves);
    }
}
```

The *mergeSort* function, defined in the header file *mergeSort.h*, directly maps to the above pseudocode. The range $[begin, midpoint)$ (line 21) denotes the left half of *S*, while the range $[midpoint, end)$ (line 22) denotes the right half.

Our test program (*main.cpp*) sorts the sequence {1 6 4 8 2 7 3 0 9 5} in ascending and descending order, generating the output

```
1 6 4 8 2 7 3 0 9 5      // Original sequence
0 1 2 3 4 5 6 7 8 9      // Sorted (ascending order)
9 8 7 6 5 4 3 2 1 0      // Sorted (descending order)
```


Part 5: Binary Search

5.1: Inheritance and Iterator Tags

Source files and folders

- *advance*
- *distance*

Chapter outline

- *An introduction to the concept of inheritance, and how it applies to iterator tags*
- *Implementing generic functions for advancing an iterator and obtaining the distance between two iterators*

In Volume 1 we worked with 2 types of iterator tags, *bidirectional_iterator_tag* and *random_access_iterator_tag*. Recall that a bidirectional iterator can be:

- Dereferenced (*)
- Compared with another iterator (==, !=), to determine whether it points to the same element
- Incremented (++) / decremented (--), to point to the next / previous element

and that a random access iterator, in addition to all of the above, can be:

- Compared with another iterator (<, >, <=, >=), to determine its relative position (whether its current position comes before or after that of the other iterator)
- Incremented (+) / decremented (-) by an integral value, to point to any element (in constant time)
- Subtracted (-) from another iterator, to obtain the distance (number of elements) between it and the other iterator (in constant time)

A random access iterator extends the capabilities of a bidirectional iterator. The Standard Library models this relationship using a concept known as *inheritance*. Inheritance is a powerful tool with a wide variety of uses, but here we'll focus on its application to iterator tags. *random_access_iterator_tag*, for example, is defined as

```
struct random_access_iterator_tag : public bidirectional_iterator_tag
{
    // ...
};
```

where the

```
: public bidirectional_iterator_tag
```

indicates that *random_access_iterator_tag* is a *subclass* (subtype) of *bidirectional_iterator_tag*. In this case, *bidirectional_iterator_tag* is the *base class* (parent class), and *random_access_iterator_tag* is the *derived class* (child class).

The Standard Library defines 5 types of iterators:

Iterator Type	Supported Operations
Output	Dereference (*) (can write to the referent element once) Comparison (==, !=) Increment (++)
Input	Dereference (*) (can read the referent element once) Comparison (==, !=) Increment (++)
Forward	All operations of an input iterator, in addition to: Dereference (*) (can read or write the referent element, more than once)
Bidirectional	All operations of a forward iterator, in addition to: Decrement (--)
Random Access	All operations of a bidirectional iterator, in addition to: Comparison (<, >, <=, >=) Random offset by an integral value (+, -, +=, -=) Subtraction (-) (to obtain the distance between two iterators)

The following diagram shows the corresponding iterator tags and inheritance hierarchy. No classes are derived from *output_iterator_tag* (it has no subclasses):

```
output_iterator_tag

    |
    |
input_iterator_tag
    |
    |
forward_iterator_tag : public input_iterator_tag
    |
    |
bidirectional_iterator_tag : public forward_iterator_tag
    |
    |
random_access_iterator_tag : public bidirectional_iterator_tag
```

Inheritance allows us to pass an object of derived class type to a function that requires an object of base class type. Consider, for example, the function (*distance.h*, lines 8-10)

```
template <class Iter>
typename std::iterator_traits<Iter>::difference_type distance(
    Iter begin,
    Iter end);
```

which returns the distance (number of elements) from *begin* to *end*.

- If *Iter*'s *iterator_category* is *input_iterator_tag*, *forward_iterator_tag*, or *bidirectional_iterator_tag*, then we have to increment *begin* (one element at a time) until we reach *end*, while manually counting the number of elements.
- If *Iter*'s *iterator_category* is *random_access_iterator_tag*, then we can simply subtract *begin* from *end*.

We achieve this by writing a separate function, *_distance*, with two overloads (lines 31-54):

```
// "Input overload"

template <class Iter>
typename std::iterator_traits<Iter>::difference_type _distance(
    Iter begin,
    Iter end,
    std::input_iterator_tag iteratorTag)
{
    // Increment begin until we reach end, while counting
    // the total number of elements traversed, then return the count
}

// "Random access overload"

template <class Iter>
inline typename std::iterator_traits<Iter>::difference_type _distance(
    Iter begin,
    Iter end,
    std::random_access_iterator_tag iteratorTag)
{
    return end - begin;
}
```

The main function, *distance* (lines 22-29), simply calls *_distance*, passing the *Iter*'s *iterator_tag*:

```
template <class Iter>
inline typename std::iterator_traits<Iter>::difference_type distance(
    Iter begin,
    Iter end)
{
    return _distance(begin,
        end,
        std::iterator_traits<Iter>::iterator_category());
```

If we pass a *random_access_iterator_tag* to *_distance* (line 28), the compiler selects the random access

overload because it's an exact match for the parameter type. Similarly, if we pass an *input_iterator_tag*, the compiler selects the input overload. If we pass a *forward_iterator_tag* or *bidirectional_iterator_tag*, the compiler will recognize it as a subtype of *input_iterator_tag*, and select the input overload accordingly.

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

```
template <class Iter, class Distance>
void advance(Iter& iter, Distance distance);
```

moves the given iterator (*iter*) forward by the given *distance* (number of elements), and is implemented the exact same way. We create a separate function, *_advance*, with two overloads:

- One for input, forward, and bidirectional iterators (lines 29-36), which increments the iterator one element at a time until the desired distance is covered
- One for random access iterators (lines 38-44), which advances the iterator by simply adding the desired distance

The *advance* function (lines 21-27) then calls *_advance*, and the compiler selects the appropriate overload based on the type of *Iter*'s *iterator_tag*.

We'll use these two functions (*distance* and *advance*) in the next chapter, to write generic versions of the upper and lower bound algorithms.

5.2: Finding the Upper and Lower Bound

Source files and folders

- *bound*

Chapter outline

- *Linear vs. binary search*
- *Implementing upper and lower bound in terms of a common function*

In an unordered sequence, such as

7 4 12 9 14 2 10 16 13 15 8 11 3 1 6 5

the only way to search for a given element is to start at the beginning and check each one, until we find a match or reach the end (a linear time operation). If we first sort the elements, however,

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

we can perform a *binary search*, which is a much more efficient (logarithmic time) method. Binary search utilizes a concept known as the *lower bound*. In an ordered (sorted) sequence,

- The *lower bound* of a value v is the first element *not less than* v (i.e. the first element *greater than or equal to* v)
- The *upper bound* of a value v is the first element *greater than* v

In the above sequence, for example,

```
lowerBound(0) = 1
lowerBound(1) = 1
lowerBound(7) = 7
lowerBound(16) = 16
lowerBound(17) = end (non-existent)

upperBound(0) = 1
upperBound(1) = 2
upperBound(7) = 8
upperBound(16) = end (non-existent)
upperBound(17) = end (non-existent)
```

To find the lower bound of a value v , we repeatedly divide the original sequence into left and right halves, comparing v to the midpoint element each time. If the midpoint is less than v , we narrow the search down to the right half; otherwise, we proceed to the left half. The search is over when the length of the current subsequence is 0.

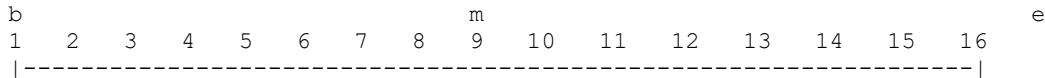
Suppose, for example, that we'd like to find the lower bound of 12:

```

b = begin      (Iterator to the first element of the current subsequence)
e = end        (Iterator to the one-past-the-last element of the entire
                  sequence)
m = midpoint   (Iterator to the element separating the two halves of the
                  current subsequence)

subsequenceLength = end - begin             = 16
midpointDistance = subsequenceLength / 2 = 8

```

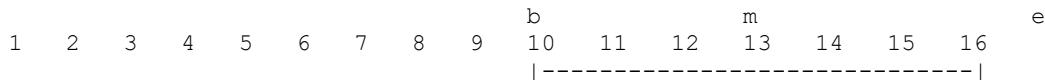


The midpoint (9) is less than v (12), so we proceed to the right half:
begin becomes the element after the midpoint
subsequenceLength decreases by (midpointDistance + 1)

```

subsequenceLength = 16 - (8 + 1)           = 7
midpointDistance = subsequenceLength / 2 = 3

```

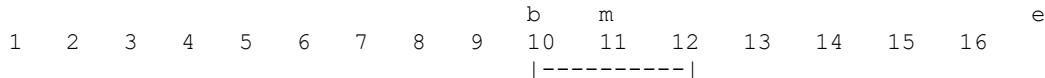


The midpoint (13) is not less than v (12), so we proceed to the left half:
begin is unchanged
subsequenceLength decreases by half

```

subsequenceLength = midpointDistance       = 3
midpointDistance = subsequenceLength / 2 = 1

```



The midpoint (11) is less than v (12), so we proceed to the right half:
begin becomes the element after the midpoint
subsequenceLength decreases by (midpointDistance + 1)

```

subsequenceLength = 3 - (1 + 1)           = 1
midpointDistance = subsequenceLength / 2 = 0

```



The midpoint (12) is not less than v (12), so we proceed to the left half:
begin is unchanged
subsequenceLength decreases by half

```

subsequenceLength = midpointDistance       = 0

```

subsequenceLength is 0, so we're done; the lower bound of v (12) is b (12)

We can summarize the lower bound algorithm with the following pseudocode:

```

lowerBound(begin, end, value)
{
    subsequenceLength = end - begin;

    while (subsequenceLength > 0)
    {
        midpointDistance = subsequenceLength / 2;
        midpoint = begin + midpointDistance;

        if (*midpoint < value)
        {
            begin = midpoint + 1;                                // Proceed to the right
            subsequenceLength -= (midpointDistance + 1);          // half
        }
        else
        {
            subsequenceLength = midpointDistance;                // Proceed to the left
        }
    }

    return begin;
}

```

The upper bound algorithm is nearly identical; the only difference is that in each iteration, we determine whether the midpoint is *less than or equal to* the search value:

```

if (*midpoint <= value)
{
    // Proceed to the right half
}
else
{
    // Proceed to the left half
}

```

To find the upper bound of 12, for example:

```

subsequenceLength = end - begin      = 16
midpointDistance = subsequenceLength / 2 = 8

```

b							m							e	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

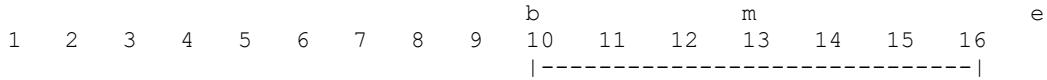
The midpoint (9) is less than or equal to v (12), so we proceed to the right half:

begin becomes the element after the midpoint
subsequenceLength decreases by (midpointDistance + 1)

```

subsequenceLength = 16 - (8 + 1)      = 7
midpointDistance = subsequenceLength / 2 = 3

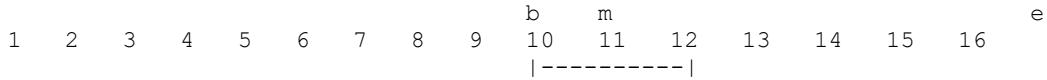
```



The midpoint (13) is not less than or equal to v (12), so we proceed to the left half:

```
begin is unchanged
subsequenceLength decreases by half
```

```
subsequenceLength = midpointDistance      = 3
midpointDistance = subsequenceLength / 2 = 1
```



The midpoint (11) is less than or equal to v (12), so we proceed to the right half:

```
begin becomes the element after the midpoint
subsequenceLength decreases by (midpointDistance + 1)
```

```
subsequenceLength = 3 - (1 + 1)      = 1
midpointDistance = subsequenceLength / 2 = 0
```



The midpoint (12) is less than or equal to v (12), so we proceed to the right half:

```
begin becomes the element after the midpoint
subsequenceLength decreases by (midpointDistance + 1)
```

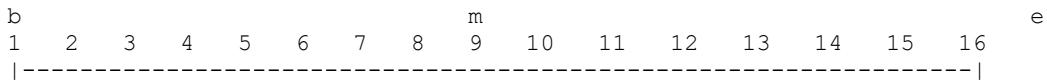
```
subsequenceLength = 1 - (0 + 1)      = 0
```



subsequenceLength is 0, so we're done; the upper bound of v (12) is b (13)

When searching for the lower bound of 0 (or any value less than 1), we go left each time:

```
subsequenceLength = end - begin      = 16
midpointDistance = subsequenceLength / 2 = 8
```



The midpoint (9) is not less than v (0), so we proceed to the left half:

```
begin is unchanged
subsequenceLength decreases by half
```

```
subsequenceLength = midpointDistance      = 8
midpointDistance = subsequenceLength / 2 = 4
```

b		m														e
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	

The midpoint (5) is not less than v (0), so we proceed to the left half:
begin is unchanged
subsequenceLength decreases by half

```
subsequenceLength = midpointDistance      = 4
midpointDistance = subsequenceLength / 2 = 2
```

b		m													e	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	

The midpoint (3) is not less than v (0), so we proceed to the left half:
begin is unchanged
subsequenceLength decreases by half

```
subsequenceLength = midpointDistance      = 2
midpointDistance = subsequenceLength / 2 = 1
```

b	m														e	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	

The midpoint (2) is not less than v (0), so we proceed to the left half:
begin is unchanged
subsequenceLength decreases by half

```
subsequenceLength = midpointDistance      = 1
midpointDistance = subsequenceLength / 2 = 0
```

b															e	
m																
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
-																

The midpoint (1) is not less than v (0), so we proceed to the left half:
begin is unchanged
subsequenceLength decreases by half

```
subsequenceLength = midpointDistance      = 0
```

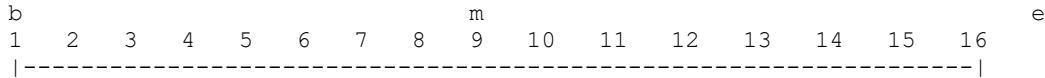
subsequenceLength is 0, so we're done; the lower bound of v (0) is b (1)

Conversely, when searching for the upper bound of 16 (or any value greater than 16), we go right each time:

```

subsequenceLength = end - begin           = 16
midpointDistance = subsequenceLength / 2 = 8

```



The midpoint (9) is less than or equal to v (16), so we proceed to the right half:

```

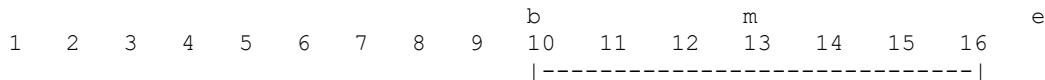
begin becomes the element after the midpoint
subsequenceLength decreases by (midpointDistance + 1)

```

```

subsequenceLength = 16 - (8 + 1)           = 7
midpointDistance = subsequenceLength / 2 = 3

```



The midpoint (11) is less than or equal to v (16), so we proceed to the right half:

```

begin becomes the element after the midpoint
subsequenceLength decreases by (midpointDistance + 1)

```

```

subsequenceLength = 7 - (3 + 1)           = 3
midpointDistance = subsequenceLength / 2 = 1

```



The midpoint (15) is less than or equal to v (16), so we proceed to the right half:

```

begin becomes the element after the midpoint
subsequenceLength decreases by (midpointDistance + 1)

```

```

subsequenceLength = 3 - (1 + 1)           = 1
midpointDistance = subsequenceLength / 2 = 0

```



The midpoint (16) is less than or equal to v (16), so we proceed to the right half:

```

begin becomes the element after the midpoint
subsequenceLength decreases by (midpointDistance + 1)

```

```

subsequenceLength = 1 - (0 + 1)           = 0

```



```
subsequenceLength is 0, so we're done; the upper bound of v (16) is b (end)
```

As mentioned earlier, the procedures for finding the upper and lower bound are nearly identical; the only difference lies in how we compare the midpoint element with the search value. We can therefore write a single function (*bound.h*, lines 31-35),

```
template <class Iter, class T, class Predicate>
Iter bound(Iter begin,
           Iter end,
           const T& value,
           Predicate predicate);
```

to do most of the work for both algorithms. The implementation (lines 73-95) closely resembles the pseudocode from earlier, but we're using the *distance* function (line 74)

```
Length subsequenceLength = ds2::distance(begin, end);
```

instead of

```
Length subsequenceLength = end - begin;
```

since that would only work for random access iterators. Similarly, we're using the *advance* function (line 81)

```
ds2::advance(midpoint, midpointDistance);
```

instead of

```
midpoint = begin + midpointDistance;
```

to allow compatibility with non-random access iterators.

To compare the midpoint element with the search value (and decide whether to proceed left or right), we use the given *predicate* (line 83); this lets us find either the upper or lower bound, depending on the type of *predicate* used. The first version of our lower bound function (lines 13-14),

```
template <class Iter, class T>
Iter lowerBound(Iter begin, Iter end, const T& value);
```

simply calls *bound*, using a less-than *predicate* (lines 37-41). The expression (line 83)

```
if (predicate(*midpoint, value))
    // Proceed to the right half
```

becomes

```
if (std::less(*midpoint, value))
    // Proceed to the right half
```

The second version of *lowerBound* lets the caller use their own type of less-than predicate, directly passing it to *bound* (lines 43-50).

Recall that to find the upper bound, we test whether the midpoint element is less than or equal to the search value:

```
if (*midpoint <= value)
    // Proceed to the right half
```

We can “rephrase” that into an equivalent expression, using a less-than operator to perform the same test:

```
if (!(value < *midpoint))
    // Proceed to the right half
                                // "If value is not less than midpoint,
                                // then value is greater than or equal
                                // to midpoint; midpoint is therefore
                                // less than or equal to value, so
                                // proceed to the right half"
```

Let's write a function object to apply this technique (*UpperBoundPredicate.h*, lines 6-21). An *UpperBoundPredicate* contains a less-than predicate (line 20), which we use to test if the midpoint element is less than or equal to the search value (lines 16-17, 36-42).

To find the upper bound, we wrap a less-than predicate inside an *UpperBoundPredicate* and pass it to *bound* (*bound.h*, lines 52-56). In the second version of *upperBound* (lines 58-65), we simply construct an *UpperBoundPredicate* from the caller's own (less-than) predicate.

Without *UpperBoundPredicate*, we would need two types of predicates: a less-than (to find the lower bound), and a less-than-or-equal-to (to find the upper bound). That approach could easily cause confusion (and errors) when users pass their own predicates to *upperBound* and *lowerBound*; someone might, for example, accidentally pass a less-than predicate to *upperBound* (and vice-versa). With *UpperBoundPredicate*, users need only concern themselves with a single type of predicate (less-than), regardless of whether they're calling *upperBound* or *lowerBound*.

Our test program (*main.cpp*) demonstrates the examples shown earlier. After populating a list and a vector with the set [1, 16] (lines 12-19), we find and print the upper and lower bound of 12, the lower bound of 0, and the upper bound of 16 (lines 21-36). The resultant output is

```
lowerBound(12) = 12
upperBound(12) = 13
lowerBound(0) = 1
upperBound(16) = end
```

We've used both a list and a vector to demonstrate the *distance* and *advance* functions from the previous chapter (called by *bound*). For list iterators, the compiler selects the input overloads of *_distance* and *_advance*; for vector iterators, the compiler selects the random access versions.

We'll use *lowerBound* to implement binary search in the next chapter, and again in the next section when working with B-trees.

5.3: Completing the Implementation

Source Files and Folders

- *binarySearch*

Chapter outline

- *Implementing binary search*

Now that we can find the lower bound, implementing binary search is easy. To search for a given value v , we find the lower bound of v . If the lower bound exists and v is not less than the lower bound, we've found v (the lower bound of v is v); otherwise, v doesn't exist in the sequence. Consider, for example, the sequence

```
2   4   6   8   10  12  14  16  18  20  22  24  26  28  30
```

and suppose that we'd like to search for the value 8:

```
v = 8
lowerBound(v) = 8
lowerBound(v) exists and v is not less than lowerBound(v), so we've found v
```

Now suppose that we search for the nonexistent values 21, 0, and 32:

```
v = 21
lowerBound(v) = 22
lowerBound(v) exists, but v is less than lowerBound(v), so v doesn't exist

v = 0
lowerBound(v) = 2
lowerBound(v) exists, but v is less than lowerBound(v), so v doesn't exist

v = 32
lowerBound(v) = end
lowerBound(v) doesn't exist, so v doesn't exist
```

We can also prove that this works using the definition of the lower bound. Recall from the previous chapter that $\text{lowerBound}(v)$ is the first element greater than or equal to v . v must therefore be less than or equal to $\text{lowerBound}(v)$:

```
lowerBound(v) >= v
v           <= lowerBound(v)
```

From this it follows that if v is not less than $\text{lowerBound}(v)$, then v must be equal to $\text{lowerBound}(v)$, which means that we've found v .

Like the upperBound and lowerBound functions from the previous chapter, our binary search function

has two versions. The first one (*binarySearch.h*, lines 13-17)

```
template <class Iter, class T, class Predicate>
Iter binarySearch(Iter begin,
                  Iter end,
                  const T& value,
                  Predicate predicate);
```

lets the caller pass their own less-than *predicate*. The implementation (lines 31-36) maps directly to the description from the beginning of the chapter (and as usual, we return *end* to indicate that the *value* wasn't found). The second version (lines 19-23) simply calls the first, passing a *std::less* predicate.

Our test program (*main.cpp*) demonstrates the above examples. We begin by populating a *vector* with the even numbers in the set [0, 30] (lines 11-14), then search for the values 8, 21, 0, and 32 (lines 16-33). The resultant output is

```
Found 8
21 not found
0 not found
32 not found
```

Part 6: B-Trees

6.1: Introducing the *KmPair* Class

Source files and folders

- *KmPair*
- *stdPairIo*
- *Traceable/I*

Chapter outline

- *An overview of B-Tree structure and terminology*
- *Creating the KmPair class to store const keys in a deque*
- *Overloading the stream operators for std::pair*
- *Updating the Traceable class to verify the destruction of all dynamically-allocated elements*

In this section we'll create a new type of data structure called a *B-tree*, invented by Rudolf Bayer and Edward M. McCreight. Like binary trees, B-trees consist of parent / child nodes, but each node can store more than 1 element and have more than 2 children.

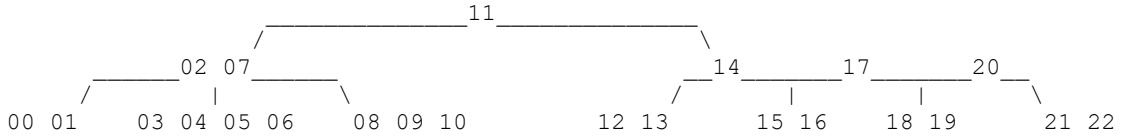
The *order* of a B-tree is the maximum allowable number of children per node. A B-Tree of order m with a total number of elements t must satisfy the following requirements:

- If t is less than m , then all elements reside in the root node (the root has 0 children); otherwise, the root has at least 2 children.
- Each inner node (non-root / non-leaf) is at least half full (having c children and $c - 1$ elements, where $m/2 \leq c \leq m$).
- All leaf nodes are at the same level, and at least half full (having n elements, where $n \geq m/2$).

In a B-tree of order 5, for example:

- If the tree contains less than 5 elements, then all elements reside in the root node (which has 0 children). If the tree contains 5 or more elements, then the root has at least 2 children.
- Each inner node (non-root / non-leaf) is at least half full, having
 - 3 children (2 elements),
 - 4 children (3 elements), or
 - 5 children (4 elements).
- All leaf nodes are at the same level, and at least half full, having 2, 3, or 4 elements.

The following diagram shows a B-tree of order 5:



For the sake of simplicity, we'll refer to each node by its first (leftmost) key value. "Node 2," for example, refers to the node containing the keys {2, 7}. Similarly, "node 14" refers to the node containing {14, 17, 20}, and "node 3" refers to the node containing {3, 4, 5, 6}. The root, "node 11," contains {11}.

We'll also use the terms *left child* and *right child* to describe a node's position relative to its parent key. Node 2, for example, has 3 children: nodes 0, 3, and 8:

- At key 2, the left and right children are nodes 0 and 3
- At key 7, the left and right children are nodes 3 and 8
- At *end* (the nonexistent key after 7), the left child is 8; there is no right child

Similarly, node 14 has 4 children, nodes 12, 15, 18, and 21:

- At key 14, the left and right children are nodes 12 and 15
- At key 17, the left and right children are nodes 15 and 18
- At key 20, the left and right children are nodes 18 and 21
- At *end* (the nonexistent key after 20), the left child is node 21; there is no right child

To find a given key value K , we begin at the root node and find the lower bound of K . If the lower bound is equal to K , then we've found K ; otherwise, we proceed to the left child of the lower bound and repeat the process. If we can no longer descend the tree, then K doesn't exist.

To find key 10, for example:

- Beginning at the root, $lowerBound(10)$ is 11. 11 is not equal to 10, so we go left (from 11) to node 2.
- At node 2, $lowerBound(10)$ is *end*. *end* is not equal to 10, so we go left (from *end*) to node 8.
- At node 8, $lowerBound(10)$ is 10. 10 is equal to 10, so we've found K .

To find key 4:

- Beginning at the root, $lowerBound(4)$ is 11. 11 is not equal to 4, so we go left (from 11) to node 2.
- At node 2, $lowerBound(4)$ is 7. 7 is not equal to 4, so we go left (from 7) to node 3.
- At node 3, $lowerBound(4)$ is 4. 4 is equal to 4, so we've found K .

To find key -1:

- Beginning at the root, $lowerBound(-1)$ is 11. 11 is not equal to -1, so we go left (from 11) to node 2.
- At node 2, $lowerBound(-1)$ is 2. 2 is not equal to -1, so we go left to (from 2) to node 0.
- At node 0, $lowerBound(-1)$ is 0. 0 is not equal to -1, and we can no longer descend the tree, so K doesn't exist.

To find key 15:

- Beginning at the root, $lowerBound(15)$ is end . end is not equal to 15, so we go left (from end) to node 14.
- At node 14, $lowerBound(15)$ is 17. 17 is not equal to 15, so we go left (from 17) to node 15.
- At node 15, $lowerBound(15)$ is 15. 15 is equal to 15, so we've found K .

To find key 19:

- Beginning at the root, $lowerBound(19)$ is end . end is not equal to 19, so we go left (from end) to node 14.
- At node 14, $lowerBound(19)$ is 20. 20 is not equal to 19, so we go left (from 20) to node 18.
- At node 18, $lowerBound(19)$ is 19. 19 is equal to 19, so we've found K .

To find key 23:

- Beginning at the root, $lowerBound(23)$ is end . end is not equal to 23, so we go left (from end) to node 14.
- At node 14, $lowerBound(23)$ is end . end is not equal to 23, so we go left (from end) to node 21.
- At node 21, $lowerBound(23)$ is end . end is not equal to 23, and we can no longer descend the tree, so K doesn't exist.

In terms of implementation, each node contains two deques: one for elements (key-mapped pairs), and one for (pointers to) child nodes. The reason we're using deques (as opposed to say, vectors) is that in addition to constant time random access (for $lowerBound$), we'll need to insert and remove elements from the front, back, and middle of the containers.

But like all the other associative data structures in the Standard Library, B-Tree's *value_type* (element type) will be *std::pair<const key_type, mapped_type>*. And because the keys are *const*, we can't directly store them in a deque because deque's *insert* and *erase* methods only work on writable (non-*const*) elements. Recall from Volume 1 that *deque::insert* actually places the new element at the front or back, then moves it to the desired location by repeatedly swapping it with its neighbor:

```
i
1 2 3 5 6 7          // deque<int> d, contains {1, 2, 3, 5, 6, 7}
// i is an iterator to element 5

d.insert(i, 4);
```

```

1 2 3 5 6 7 4      // push_back(4), then swap 4 and 7
1 2 3 5 6 4 7      // swap 4 and 6
1 2 3 5 4 6 7      // swap 4 and 5
1 2 3 4 5 6 7

```

Similarly, `deque::erase` moves the desired element to the front or back (again, by repeatedly swapping it with its neighbor), then removes it using `pop_front` or `pop_back`:

```

          i
1 2 3 4 5 6 7

d.erase(i);

1 2 3 4 5 6 7      // swap 4 and 5
1 2 3 5 4 6 7      // swap 4 and 6
1 2 3 5 6 4 7      // swap 4 and 7
1 2 3 5 6 7 4      // pop_back()
1 2 3 5 6 7

```

We can't swap *const* elements, however, since we can't reassign their values. So rather than store our `std::pairs` of `<const key_type, mapped_type>` directly in the deque, we'll create a new class, *KmPair* (short for "Key-Mapped Pair"), to store them indirectly.

A *KmPair<Key, Mapped>* (*KmPair.h*, lines 19-20) contains a pointer to a single, dynamically-allocated `std::pair<const Key, Mapped>` (lines 25, 48). *StdPair* is simply an alias of `std::pair<const Key, Mapped>`. The *allocator* (line 49) manages the *StdPair* (`std::allocator`, provided by the `<memory>` header, is the Standard Library version of the *Allocator* class that we wrote in Volume 1).

In the default constructor (line 31), we allocate a block of memory for a single *StdPair* and construct it at address *_p*, using *StdPair*'s default constructor (lines 66-71).

The next constructor (line 32) creates a *KmPair* from a *StdPair*, in which the dynamically-allocated *StdPair* is a copy of the given *stdPair* (lines 73-78). This also allows for the implicit conversion of a `std::pair<const Key, Mapped>` (B-Tree's *value_type*) to a *KmPair<Key, Mapped>*.

The third constructor (line 33) initializes the dynamically-allocated *StdPair*'s *first* and *second* members to the given *key* and *mapped* values (lines 80-86).

The *const*-overloaded *stdPair* method (lines 38, 42) provides direct access to the dynamically-allocated *StdPair* (lines 102-107, 123-127). Similarly, the *key* and *mapped* methods (lines 39-40, 43-44) provide access to the *StdPair*'s *first* and *second* members (lines 109-121, 129-139).

The copy constructor (line 34) creates a *KmPair* from another *KmPair*. The new *KmPair* contains a copy of the original *KmPair*'s *StdPair* (lines 88-93).

The destructor (line 36) destroys the dynamically-allocated *StdPair* and deallocates the memory block (lines 95-100).

The assignment operator (line 45) sets the key and mapped values of the left operand to those of the right operand. But since the key value,

```
_p->first
```

is *const*, we can't just assign the new value, as in

```
_p->first = rhs._p->first;           // Compiler error: _p->first is const, so we
                                         // can't change its value

_p->second = rhs._p->second;         // Okay: _p->second is not const, so we can
                                         // change its value
```

What we can do, however, is destroy the left operand's entire *StdPair* (without deallocating the memory), then construct a copy of the right operand's *StdPair* in its place (lines 141-148). Given

```
KmPair<int, int> x(1, 2);
KmPair<int, int> y(3, 4);
```

for example, there are two *std::pairs* of *<const int, int>*, at addresses *x._p* and *y._p*:

```
x._p ---> std::pair<const int, int>(1, 2)
y._p ---> std::pair<const int, int>(3, 4)
```

The expression

```
x = y
```

destroys the entire *std::pair* at address *x._p*,

```
x._p --->
y._p ---> std::pair<const int, int>(3, 4)
```

then constructs a copy of *y*'s *std::pair* in its place (at the same address, *x._p*):

```
x._p ---> std::pair<const int, int>(3, 4)
y._p ---> std::pair<const int, int>(3, 4)
```

We can now indirectly store *std::pairs* of *<const int, int>* in a deque by creating a *deque<KmPair<int, int>>*. For a B-tree of *<const int, int>* pairs, the *value_type* from the user's perspective is *std::pair<const int, int>*, just like any other associative container. Internally, however, each node stores its elements in a *deque<KmPair<int, int>>*. This lets us manipulate the deques as needed on the implementation side (within *insert* and *erase*) while maintaining the *const*-ness of the key values (*std::pair::first*) on the user side.

The output stream operator (lines 13-14) simply inserts the *StdPair* into the *ostream* (lines 52-57), while the input stream operator (lines 16-17) writes to the *StdPair* from the *istream* (lines 59-64). For this to work, however, we need to overload the stream operators for *std::pair* (*stdPairIo.h*). This is nearly identical to what we wrote in Volume 1 (*dss/Pair/PairIo.h*); the only difference is that this

version uses `std::pair` instead of `dss::Pair`. These operators will also be useful later on, since all of the associative data structures developed in this book will use `std::pair` (rather than `dss::Pair`) as their `value_type`.

Finally, we'll make a small update to the `Traceable` class from Volume 1 (`Traceable.h`). The only addition here is that this version keeps track of the total number of `Traceables` via a static data member, `_totalObjects` (lines 43, 48-49). Whenever a `Traceable` is constructed, we increment `_totalObjects` (lines 51-73). The destructor (lines 75-85) decrements `_totalObjects`, and if the last object has been destroyed, prints the message

```
All Traceables destroyed
```

This will provide a convenient way to verify that our data structures have destroyed all of their dynamically-allocated elements.

To test the functionality of the `KmPair` class, we construct a deque of `KmPair<Traceable<int>, int>` (`main.cpp`, lines 16, 19). Each `KmPair` in the deque contains a dynamically-allocated `std::pair<const Traceable<int>, int>` (for which we've declared `StdPair` as an alias, in line 17). We then insert the `StdPairs`

```
(0,0) (1,0) (2,0) (4,0) (5,0) (6,0)
```

using `push_front` and `push_back` (lines 21-25). The call to `printContainer` (line 29) demonstrates `KmPair`'s `ostream` operator (which calls the corresponding function in `stdPairIo.h`). We then insert the `StdPair` (3,0) in the middle (line 32), generating the sequence

```
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0)
```

Erasing the middle element (line 38) leaves us with the original sequence,

```
(0,0) (1,0) (2,0) (4,0) (5,0) (6,0)
```

Upon termination of `main`, the deque destroys the remaining `KmPairs` (which destroy their dynamically-allocated `StdPairs`), generating the output

```
~ 6
~ 5
~ 4
~ 2
~ 1
~ 0
```

```
All Traceables destroyed
```

6.2: Implementing Upper and Lower Bound for *KmPairs*

Source files and folders

- *KmPairBound*

Chapter outline

- *Finding the upper and lower bound of a given key value, in a sequence of KmPairs*

As shown in the previous chapter, searching a B-tree entails finding the lower bound of the desired key value (in the root node) and descending the tree accordingly. In Chapter 5.2, we wrote the functions

```
template <class Iter, class T>
Iter lowerBound(Iter begin, Iter end, const T& value);

template <class Iter, class T>
Iter upperBound(Iter begin, Iter end, const T& value);
```

which simply forward their arguments to the *bound* function,

```
template <class Iter, class T, class Predicate>
Iter bound(Iter begin,
           Iter end,
           const T& value,
           Predicate predicate);
```

Recall that *bound* works by comparing the *midpoint* element with the given search *value* (*bound/bound.h*, lines 83-92):

```
if (predicate(*midpoint, value))
{
    // Proceed to the right half
}
else
{
    // Proceed to the left half
}
```

When searching for the lower bound, the *predicate* checks whether **midpoint* is less than *value*; when searching for the upper bound, the *predicate* checks whether *value* is not less than **midpoint*.

lowerBound passes a *std::less<T>* for the *predicate* (line 40), while *upperBound* passes an *UpperBoundPredicate<std::less<T>>* (line 55) – both of which only work if *value* and **midpoint* are the same type. When searching for an *int* value in a *vector<int>*, for example, **midpoint* and *value* are both *ints*. Similarly, when searching for a *string* value in a *deque<string>*, **midpoint* and *value* are both *strings*.

In our B-tree, implementation, however, **midpoint* and *value* have different types:

- **midpoint*'s type is *KmPair<key_type, mapped_type>*
- *value*'s type is *key_type*

When using *bound* to search a B-tree containing *std::pairs* of *<const string, int>*, for example,

- **midpoint*'s type is *KmPair<string, int>* (since the *std::pairs* are stored in *deques* of *KmPairs*)
- *value*'s type is *string*

We therefore can't simply use a *std::less<string>* predicate to check whether **midpoint* is less than *value*. Similarly, we can't use an *UpperBoundPredicate<std::less<string>>* to check whether *value* is not less than **midpoint*.

To work around this we need to create a new type of predicate, specifically designed to compare a *KmPair<string, int>* (**midpoint*) with a *string* (*value*) – or more generally, a *KmPair<Key, Mapped>* with a *Key*. To compare **midpoint* and *value*, this new predicate will simply compare **midpoint*'s key (*midpoint->key()*, or *midpoint->stdPair()->first*) with *value*.

Let's create two new types of predicate, one for finding the lower bound and one for finding the upper bound. Our first new predicate, *KmPairLowerBoundPred*, performs a less-than comparison between the *midpoint* element's key and the desired search *value* (*KmPairBound/KmPairPred.h*, lines 6-19). The private member *_keyPred* (line 18) is a less-than predicate that compares two keys.

The default constructor (line 12) initializes *_keyPred* using its own default constructor (lines 36-40), while the other constructor (line 13) lets us pass in our own *_keyPred* (lines 42-48).

If *_keyPred* is a *std::less<key_type>*, the function call operator (line 15) will return true if *midpoint.key()* is less than the search *value* (lines 50-56), thereby finding the lower bound when applied to the *bound* function.

Continuing with our example of a B-tree containing *<string, int>* pairs,

- The B-tree's *key_type* is *string*, its *mapped_type* is *int*, and its *value_type* (element type) is *std::pair<const string, int>*.
- Each node stores its elements (*std::pair<const string, int>*) in a *deque<KmPair<string, int>>*.
- To search for a given key, we call the *bound* function, in which the *predicate* is a *KmPairLowerBoundPred*, whose
 - *KmPair* type is *KmPair<string, int>* (**midpoint*'s type),
 - *key_type* is *string* (*value*'s type), and
 - *KeyPred* type is *std::less<string>*

- The *predicate*'s full typename is therefore `KmPairLowerBoundPred<KmPair<string, int>, less<string>>`.
- In each iteration of *bound*, the *predicate* checks whether *midpoint.key()* (a *string*) is less than *value* (also a *string*).

Our other new predicate, `KmPairUpperBoundPred` (lines 21-34), follows the same design pattern as `KmPairLowerBoundPred`. It also uses a less-than predicate for the `_keyPred`, but its function call operator (line 30) returns true if *value* is not less than *midpoint.key()* (lines 72-78), thereby finding the upper bound when applied to the *bound* function.

Now that we have compatible predicates, let's write a pair of simple wrapper functions for convenience and readability (`KmPairBound/KmPairBound.h`, lines 9-19):

```
template <class KmPairIter, class KeyPred>
KmPairIter kmPairLowerBound(KmPairIter begin,
    KmPairIter end,
    const typename KmPairIter::value_type::key_type& key,
    KeyPred keyPred);

template <class KmPairIter, class KeyPred>
KmPairIter kmPairUpperBound(KmPairIter begin,
    KmPairIter end,
    const typename KmPairIter::value_type::key_type& key,
    KeyPred keyPred);
```

`kmPairLowerBound` finds the lower bound of the desired *key* value in a sequence of *KmPairs*, using the given *keyPred*, a less-than predicate that compares two key values. `KmPairIter` is the type of iterator used to traverse the sequence of *KmPairs*. The function simply forwards its arguments to *bound*, wrapping the *keyPred* in a `KmPairLowerBoundPred` (lines 21-34).

Similarly, `kmPairUpperBound` wraps the *keyPred* in a `KmPairUpperBoundPred` and calls *bound* (lines 36-49).

Our test program (`main.cpp`) begins by populating a `deque<KmPair<int, int>>` with the sequence $[(0,0), (19,0)]$ (lines 11-20). We then call `kmPairLowerBound`, which returns an iterator to the *KmPair* (7,0) (the *KmPair* whose key value is the lower bound of 7) (lines 22-27). Similarly, `kmPairUpperBound` returns an iterator to the *KmPair* (8,0) (the *KmPair* whose key value is the upper bound of 7) (lines 29-34).

6.3: Introducing the *BTreeNode* Class

Source files and folders

- *BTree/common/BTreeNode.h*

Chapter outline

- *Creating a class for B-tree nodes*

Now that we can store *std::pairs* of *<const Key, Mapped>* in a *deque* and find the lower bound of a desired key value, we're ready to implement the nodes for our B-tree. The *BTreeNode* class (*BTreeNode.h*, line 12) has 4 template parameters:

- *Key*, the type of key values
- *Mapped*, the type of value mapped to each key
- *KeyPred*, the type of less-than predicate used to compare two key values
- *Order*, the maximum allowable number of children per node

Each node contains a *deque* of *KmPairs* (*_kmPairs*), a *deque* of pointers to child nodes (*_children*), and a pointer to its parent node (*_parent*) (lines 79-81).

The default constructor (line 30) initializes the parent pointer to null and leaves both *deques* empty (lines 84-89).

The *totalElements* and *totalChildren* methods (lines 32-33) simply return the total number of elements / children (the sizes of *_kmPairs* and *_children*) (lines 91-103).

Recall from Chapter 6.1 that each non-root node must be at least half full. *isAtLeastHalfFull* (line 34) returns *true* if the total number of elements is greater than or equal to half the capacity (lines 105-109). Similarly, *isMoreThanHalfFull* (line 35) returns *true* if the total number of elements is greater than half the capacity (lines 111-115). In an *Order 7* node, for example,

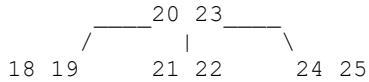
- The maximum allowable number of elements is 6
- *isAtLeastHalfFull* returns *true* if the node has 3 or more elements
- *isMoreThanHalfFull* returns *true* if the node has more than 3 elements.

hasOverflow (line 36) returns *true* if the node's total number of elements exceeds the capacity (lines 117-121). In an *Order 5* node, for example, *hasOverflow* returns *true* if the node has more than 4 elements.

hasElement (line 37) returns *true* if the node has an element at the given *index* (lines 123-128). In a node containing 3 elements, for example, the elements reside at *_kmPairs[0]*, *_kmPairs[1]*, and *_kmPairs[2]*. *hasElement* therefore returns *true* for *index* values of 0, 1, and 2, and *false* for all other

index values (which are out-of-bounds).

hasLeftChild / hasRightChild (lines 38-39) return *true* if the node has a left / right child at the element with the given index (lines 130-142). Consider, for example, node 20 in the following diagram:



- Element 0 (*_kmPairs[0]*) is 20
 - *hasLeftChild(0)* is *true* because the node has a left child at element 0 (20's left child is node 18)
 - *hasRightChild(0)* is *true* because the node has a right child at element 0 (20's right child is node 21)
- Element 1 (*_kmPairs[1]*) is 23
 - *hasLeftChild(1)* is *true* because the node has a left child at element 1 (23's left child is node 21)
 - *hasRightChild(1)* is *true* because the node has a right child at element 1 (23's right child is node 24)
- Element 2 (*_kmPairs[2]*) is *end*
 - *hasLeftChild(2)* is *true* because the node has a left child at element 2 (*end*'s left child is node 24)
 - *hasRightChild(2)* is *false* because the node does not have a right child at element 2 (*end* doesn't have a right child)

isRoot (line 40) returns *true* if the node is the root of the tree (lines 144-148). *isLeaf* (line 41) returns *true* if the node is a leaf (has no children) (lines 150-154).

frontChild (line 54) returns a pointer to the node's front (leftmost) child (the *front* element's left child) (lines 249-254). *backChild* (line 55) returns a pointer to the node's back (rightmost) child (the *back* element's right child) (lines 256-261). In the above diagram, node 20's *frontChild* is node 18 and its *backChild* is node 24.

isLeftChild / isRightChild (lines 42-43) return *true* if the node is a left / right child of its parent (lines 156-172). For any non-root node *n*,

- If *n* is the *frontChild* of its parent, then *n* is a left child only (*n* is not the right child of any element)
- If *n* is the *backChild* of its parent, then *n* is a right child only (*n* is not the left child of any element)
- If *n* is neither the *frontChild* nor *backChild* of its parent, then *n* is both a left and right child (*n* is the left child of one element and the right child of another)

In the above diagram, for example,

- Node 18 is the left child of element 0 (20), so *isLeftChild* is *true*
- Node 18 is not the right child of any element, so *isRightChild* is *false*
- Node 24 is the right child of element 1 (23), so *isRightChild* is *true*
- Node 24 is not the left child of any element, so *isLeftChild* is *false*
- Node 21 is the left child of element 1 (23), so *isLeftChild* is *true*
- Node 21 is the right child of element 0 (20), so *isRightChild* is *true*

frontElement / *backElement* (lines 44-45) return *const* references to the node's front / back *std::pairs* (lines 174-186), while *element* (line 46) returns a *const* reference to the *std::pair* at the given *index* (lines 188-193). The non-*const* versions of these methods return references (lines 59-61, 295-314).

key (line 47) returns a *const* reference to the key value at the given *index* (lines 195-200).

backElementIndex (line 48) returns the index value of the node's back (rightmost) element (lines 202-207). In the above diagram, for example, node 20's *backElementIndex* is 1 (the back element, 23, is at index 1).

lowerBound (line 49) returns the index value of the element whose key value is the lower bound of the given *key* (lines 209-219). *kmPairLowerBound* gets us an iterator *i* to the lower bound, from which we subtract *_kmPairs.begin()* to obtain the offset (index value of *i*).

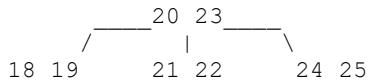
parent (line 50) returns a pointer to the node's parent (lines 221-226).

child (line 51) returns a pointer to the child node at the given *index* (lines 228-233). *leftChild* / *rightChild* (lines 52-53) return pointers to the left / right children of the element with the given index (lines 235-247). Consider, for example, node 20 in the above diagram:

- *child(0)* returns a pointer to node 18 (*_children[0]*)
- *child(1)* returns a pointer to node 21 (*_children[1]*)
- *child(2)* returns a pointer to node 24 (*_children[2]*)
- *leftChild(0)* returns a pointer to node 18 (the left child at element 0 (20))
- *leftChild(1)* returns a pointer to node 21 (the left child at element 1 (23))
- *leftChild(2)* returns a pointer to node 24 (the left child at element 2 (*end*))
- *rightChild(0)* returns a pointer to node 21 (the right child at element 0 (20))
- *rightChild(1)* returns a pointer to node 24 (the right child at element 1 (23))

leftSibling / rightSibling (lines 56-57) return pointers to the node's left / right siblings (or null if the node doesn't have a left / right sibling) (lines 263-293). The *left sibling* of a node *n* is the left child of *n*'s parent element, while the *right sibling* is the right child of *n*'s parent element. The exceptions are the root node (which has no siblings), the *frontChild* (which has no left sibling), and the *backChild* (which has no right sibling); all other nodes are guaranteed to have both a left and right sibling.

To find the left / right sibling of a non-root node *n*, we first find the index of *n*'s parent element (*parentElementIndex*, lines 269, 285) by searching *n*'s parent for the *lowerBound* of *n*'s leftmost key (*key(0)*). We can then call *leftChild / rightChild* on *n*'s parent node, using *parentElementIndex*. Once again using the tree



as an example,

- Node 20 has no siblings because it is the root (lines 267, 283)
- Node 18 has no left sibling because it is the *frontChild* of its parent (line 267)
- Node 18's right sibling is node 21:
 - The index of node 18's parent element (20) is 0 (line 285)
 - The parent's right child at element 0 (20) is node 21 (line 287)
- Node 21's left sibling is node 18:
 - The index of node 21's parent element (23) is 1 (line 269)
 - The parent's left child at element 0 (20) is node 18 (line 271)
- Node 21's right sibling is node 24:
 - The index of node 21's parent element (23) is 1 (line 285)
 - The parent's right child at element 1 (23) is node 24 (line 287)
- Node 24's left sibling is node 21:
 - The index of node 24's parent element (*end*) is 2 (line 269)
 - The parent's left child at element 1 (23) is node 21 (line 271)
- Node 24 has no right sibling because it is the *backChild* of its parent (line 283)

kmPair (line 62) returns a reference to the *KmPair* at the given *index* (lines 316-321).

insertElement (line 63) inserts the *newElement* before the element at the given *index*, and returns an iterator to the newly inserted element (lines 323-329).

overwriteElement (line 64) overwrites the element at the given *index* with the *newElement* (lines 331-

337).

eraseElement (line 65) erases the element at the given *index* (lines 339-343).

insertChild (line 66) inserts the given *child* pointer before the child at the given *index* (lines 345-350).

eraseChild (line 67) erases the child pointer at the given *index* (lines 352-356).

setParent (line 68) sets the node's *_parent* pointer to the *newParent* (lines 358-362).

pushFrontElement / pushBackElement (lines 69-70) insert the *newElement* at the front / back of the *deque* (*_kmPairs*) (lines 364-376).

pushFrontChild / pushBackChild (lines 71-72) insert the *child* pointer at the front / back of the *deque* (*_children*) (lines 378-390).

popFrontElement / popBackElement (lines 73-74) erase the element at the front / back of the *deque* (lines 392-402).

popFrontChild / popBackChild (lines 75-76) erase the child pointer at the front / back of the *deque* (lines 404-414).

6.4: Recursive In-Order Traversal

Source files and folders

- *BTreeInOrderRecursive*
- *PrintBTreeNode*

Chapter outline

- *Implementing a recursive function to perform in-order traversal of a B-tree*
- *Printing B-tree nodes*

Before we begin writing the B-tree class, we need a way to verify the contents and structure of the tree.

In Volume 1, we performed recursive in-order traversal of binary trees via the function

```
template <class Node, class Function>
void traverseInOrder(Node* n, Function visit)
{
    if (n != nullptr)
    {
        traverseInOrder(n->left, visit);           // Traverse n's left subtree
        visit(n);                                // Visit (e.g. print) n
        traverseInOrder(n->right, visit);          // Traverse n's right subtree
    }
}
```

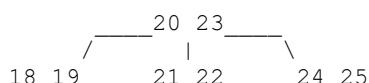
In this chapter we'll apply the same technique to B-tree nodes. The algorithm for traversing a tree rooted at node *n* is

```
traverse(Node n)
{
    for each element e in n
    {
        if n has a left child at e,
            traverse(the left child);

        visit n at element e;

        if e is the back (rightmost) element in n, and n has a right child at e,
            traverse(the right child);
    }
}
```

Consider, for example, the tree



Beginning at the root (node 20), the algorithm generates the proper in-order sequence of visits:

```

traverse(node 20)
{
    element 0 (20) has a left child, so traverse(node 18)
    {
        element 0 (18) does not have a left child;
        visit(node 18, element 0);
        element 0 (18) is not the back element;

        element 1 (19) does not have a left child;
        visit(node 18, element 1);
        element 1 (19) is the back element, but does not have a right child;
    }
    visit(node 20, element 0);
    element 0 is not the back element;

    element 1 (23) has a left child, so traverse(node 21)
    {
        element 0 (21) does not have a left child;
        visit(node 21, element 0);
        element 0 (21) is not the back element;

        element 1 (22) does not have a left child;
        visit(node 21, element 1);
        element 1 (22) is the back element, but does not have a right child;
    }
    visit(node 20, element 1);
    element 1 is the back element, and has a right child, so traverse(node 24)
    {
        element 0 (24) does not have a left child;
        visit(node 24, element 0);
        element 0 (24) is not the back element;

        element 1 (25) does not have a left child;
        visit(node 24, element 1);
        element 1 (25) is the back element, but does not have a right child;
    }
}

```

The function (*BTreeInOrderRecursive.h*, lines 6-7)

```

template <class Node, class Function>
void traverseBTreeInOrder(const Node* n, Function visit);

```

performs a recursive in-order traversal of the B-tree rooted at *n*, applying the given *visit* function to each element (lines 9-25). The loop variable *i* (line 14) is the index value of the current element.

Our *visit* function is (*PrintBTreeNode.h*, lines 8-14)

```
template <class Node>
struct PrintBTreeNode
{
    typedef typename Node::Index Index;

    void operator()(const Node* n, Index element);
};
```

The function call operator (lines 16-33) prints the key value of the current element, followed by the front keys of its parent and children (where applicable). The extra whitespace (lines 21, 24, 27, 30) simply allows the printed key values to line up, for improved readability. For the above tree, the resulting output would be

```
key          18
parent->front 20

key          19
parent->front 20

key          20
left         18
right        21

key          21
parent->front 20

key          22
parent->front 20

key 23
left         21
right        24

key          24
parent->front 20

key          25
parent->front 20
```


6.5: Introducing the *BTree* Class

Source files and folders

- *BTree/I*
- *BTree/common/memberFunctions_I.h*
- *BTree/insert/insert_I.h*

Chapter outline

- *Implementing the constructor, destructor, basic accessor methods, and insertion*

Having implemented our nodes and testing tools, we're ready to begin writing the *BTree* class (*BTree.h*). By default, a *BTree<Key, Mapped>* uses a *std::less<Key>* to compare key values, with each node having up to 5 children (lines 13-17). *Node* (line 30) is an alias of *BTreeNode<Key, Mapped, KeyPred, Order>*.

The private data members (lines 77-82) are:

- *_root*, a pointer to the root node
- *_head*, a pointer to the leftmost node
- *_tail*, a pointer to the rightmost node
- *_size*, the total number of elements
- *_keyPred*, the predicate used to compare key values
- *_alloc*, an *allocator* for creating / destroying nodes

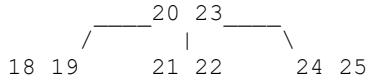
The default constructor (line 32) initializes all pointers to null, and the *_size* to 0 (*memberFunctions_I.h*, lines 5-13).

The accessor methods (*BTree.h*, lines 35-42, 44-45, 47-49 / *memberFunctions_I.h*, lines 21-109) are:

- *empty*, which returns *true* if the tree contains 0 elements
- *size*, which returns the total number of elements
- *front / back*, which return references to the front / back elements of the entire tree
- *key_comp*, which returns the predicate used to compare keys (*_keyPred*)
- *root / head / tail*, which return pointers to the root / head / tail nodes

The private member function *_createNode* (*BTree.h*, line 73) creates a new node and returns a pointer to the newly created node (*memberFunctions_I.h*, lines 226-234). *_destroyNode* (*BTree.h*, line 74) destroys the given node *n* (*memberFunctions_I.h*, lines 236-241).

_destroyTree (*BTree.h*, line 75) recursively destroys all nodes of the tree / subtree with the given *root* (*memberFunctions_I.h*, lines 243-256). Given the tree



for example, the function performs the following operations:

```

_destroyTree(node 20)
{
    (node 20)->totalChildren = 3
    _destroyTree((node 20)->frontChild)           // _destroyTree(node 18)
    {
        (node 18)->totalChildren = 0
        _destroyNode(node 18);
    }
    (node 20)->popFrontChild();                  // remove pointer to node 18

    (node 20)->totalChildren = 2
    _destroyTree((node 20)->frontChild)           // _destroyTree(node 21)
    {
        (node 21)->totalChildren = 0
        _destroyNode(node 21);
    }
    (node 20)->popFrontChild();                  // remove pointer to node 21

    (node 20)->totalChildren = 1
    _destroyTree((node 20)->frontChild)           // _destroyTree(node 24)
    {
        (node 24)->totalChildren = 0
        _destroyNode(node 24);
    }
    (node 20)->popFrontChild();                  // remove pointer to node 24

    _destroyNode(node 20);
}
  
```

The destructor (*BTree.h*, line 33) simply passes the *_root* pointer to *_destroyTree* (*memberFunctions_1.h*, lines 15-19).

Recall from Chapter 6.1 that for a tree to be balanced, all leaf nodes must be at the same level. To maintain this property when inserting a new element, we always place the new element in a leaf node *n*.

- If *n* doesn't overflow (*n->totalElements()* doesn't exceed the maximum allowable number), we're done.
- Otherwise, we “promote” *n*'s middle element (move it to *n*'s parent node) and split *n* in half. We then proceed to *n*'s parent node, check for overflow, and repeat the process if necessary.

Consider, for example, an empty B-Tree of order 5. The first 4 elements all fit in the root (which is

also a leaf node, until we insert the 5th element):

```
insert(10);

Beginning at the root, lowerBound(10) is end;

The root is a leaf node,
so insert the new element (10) before end;

          10

insert(0);

Beginning at the root, lowerBound(0) is 10;

The root is a leaf node,
so insert the new element (0) before 10;

          00 10

insert(15);

Beginning at the root, lowerBound(15) is end;

The root is a leaf node,
so insert the new element (15) before end;

          00 10 15

insert(5);

Beginning at the root, lowerBound(5) is 10;

The root is a leaf node,
so insert the new element (5) before 10;

          00 05 10 15
```

Once we insert the 5th element, the root overflows:

```
insert(20);

Beginning at the root, lowerBound(20) is end;

The root is a leaf node,
so insert the new element (20) before end;

          _root -> 00 05 10 15 20

The root has overflow,
so extract the middle element (10) and split the root in half;
```

```

10 <- middleElement

_root ->    00 05      15 20
              |          |
leftHalf     rightHalf

```

The left half is the `_root`,
so create a new root node (updating the `_root` pointer accordingly),
and insert the middle element (10) there;

```

_root ->           10
                  00 05      15 20
                  |          |
leftHalf     rightHalf

```

In the root, insert pointers to the left and right halves
and set the left and right halves' parent pointers to the root;

```

_root ->           10
                  / \
                  00 05      15 20

```

Node 15 now has room for 2 more elements:

```

insert(30);

Beginning at the root, lowerBound(30) is end;

The root is not a leaf node,
so proceed to end's left child (node 15);

Node 15 is a leaf,
so insert the new element before lowerBound(30), end;

```

```

          10
          / \
          00 05      15 20 30

```

```

insert(25);

Beginning at the root, lowerBound(25) is end;

The root is not a leaf node,
so proceed to end's left child (node 15);

Node 15 is a leaf,
so insert the new element before lowerBound(25), 30;

```

```

          10
          / \
          00 05      15 20 25 30

```

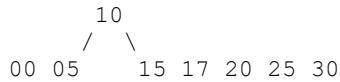
At this point, inserting any key value greater than 10 will overflow node 15:

```
insert(17);

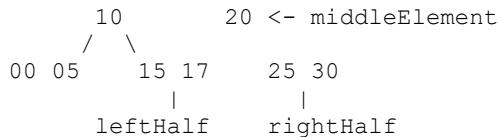
Beginning at the root, lowerBound(17) is end;

The root is not a leaf node,
so proceed to end's left child (node 15);

Node 15 is a leaf,
so insert the new element before lowerBound(17), 20;
```

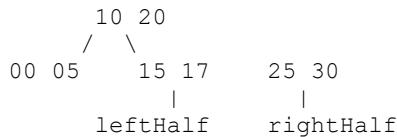


```
Node 15 has overflow,
so extract the middle element (20) and split the node in half;
```



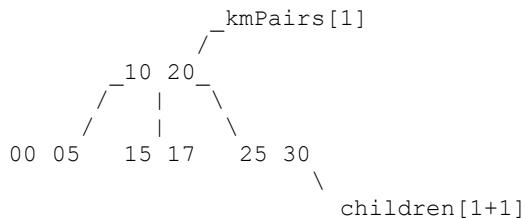
```
The left half is not the root,
so insert the middle element in the left half's parent (node 10);
```

```
In node 10, lowerBound(20) is end,
so insert 20 before end;
```



```
The right half becomes the right child of the promoted element (20);
```

```
In node 10,
insert a pointer to the right half at (index of promoted element) + 1,
and set the right half's parent pointer to node 10;
```



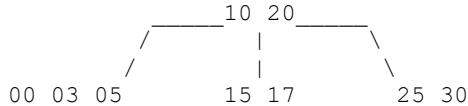
Nodes 0 and 15 each have room for 2 more elements:

```
insert(3);
```

Beginning at the root, lowerBound(3) is 10;

The root is not a leaf node,
so proceed to 10's left child (node 0);

Node 0 is a leaf,
so insert the new element before lowerBound(3), 5;

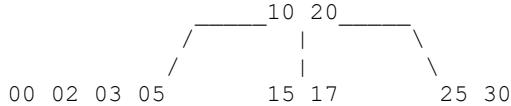


```
insert(2);
```

Beginning at the root, lowerBound(2) is 10;

The root is not a leaf node,
so proceed to 10's left child (node 0);

Node 0 is a leaf,
so insert the new element before lowerBound(2), 3;

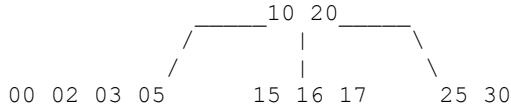


```
insert(16);
```

Beginning at the root, lowerBound(16) is 20;

The root is not a leaf node,
so proceed to 20's left child (node 15);

Node 15 is a leaf,
so insert the new element before lowerBound(16), 17;

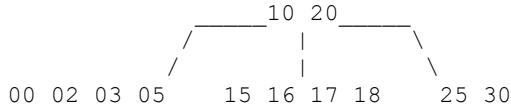


```
insert(18);
```

Beginning at the root, lowerBound(18) is 20;

The root is not a leaf node,
so proceed to 20's left child (node 15);

Node 15 is a leaf,
so insert the new element before lowerBound(18), end;



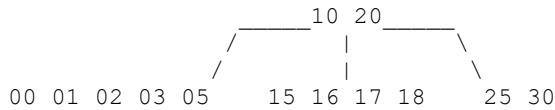
At this point, inserting any key value less than 10 will overflow node 0:

```
insert(1);
```

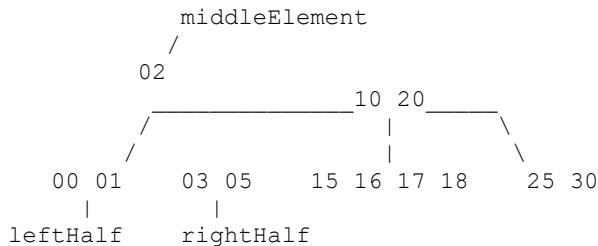
Beginning at the root, lowerBound(1) is 10;

The root is not a leaf node,
so proceed to 10's left child (node 0);

Node 0 is a leaf,
so insert the new element before lowerBound(1), 2;

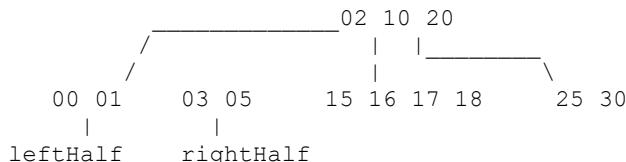


Node 0 has overflow,
so extract the middle element (2) and split the node in half;



The left half is not the root,
so insert the middle element in the left half's parent (node 10);

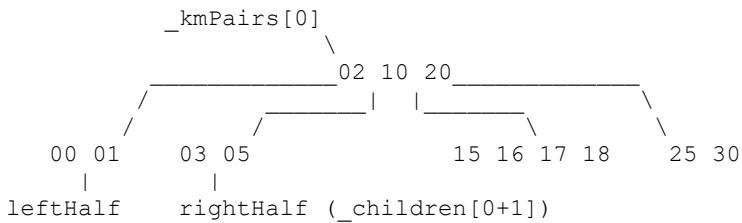
In node 10, lowerBound(2) is 10,
so insert 2 before 10;



The right half becomes the right child of the promoted element (2);

In node 2,
insert a pointer to the right half at (index of promoted element) + 1;

Set the right half's parent pointer to the left half's parent (node 2);



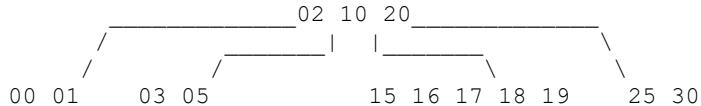
Inserting the key value 19 overflows node 15:

```
insert(19);
```

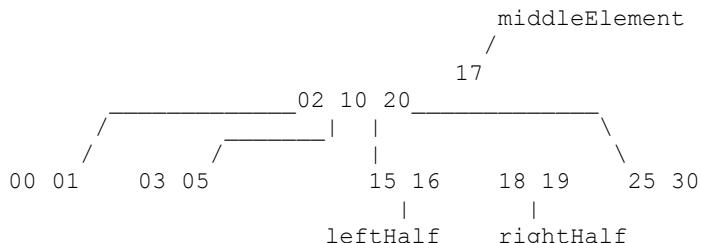
Beginning at the root, lowerBound(19) is 20;

The root is not a leaf node,
so proceed to 20's left child (node 15);

Node 15 is a leaf,
so insert the new element before lowerBound(19), end;

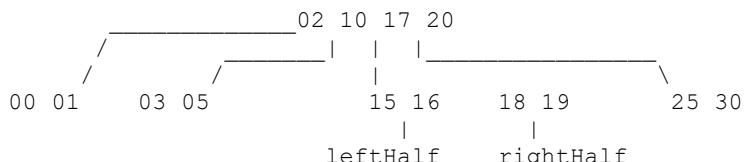


Node 15 has overflow,
so extract the middle element (17) and split the node in half;



The left half is not the root,
so insert the middle element in the left half's parent (node 2);

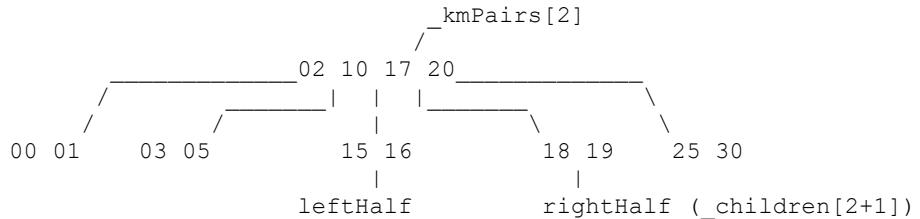
In node 2, lowerBound(17) is 20,
so insert 17 before 20;



The right half becomes the right child of the promoted element (17);

```
In node 2,
    insert a pointer to the right half at (index of promoted element) + 1;
```

Set the right half's parent pointer to the left half's parent (node 2);



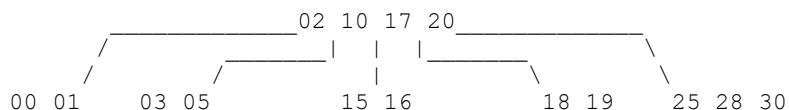
Inserting 3 new elements into node 25 will create another overflow, which will propagate up the tree and result in the creation of a new root:

```
insert(28);
```

Beginning at the root, lowerBound(28) is end;

The root is not a leaf node,
 so proceed to end's left child (node 25);

Node 25 is a leaf,
 so insert the new element before lowerBound(28), 30;

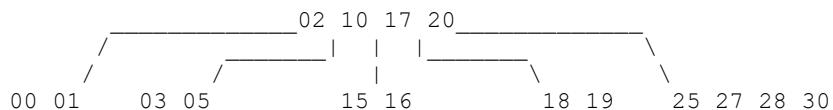


```
insert(27);
```

Beginning at the root, lowerBound(27) is end;

The root is not a leaf node,
 so proceed to end's left child (node 25);

Node 25 is a leaf,
 so insert the new element before lowerBound(27), 28;

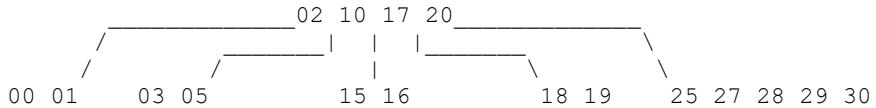


```
insert(29);
```

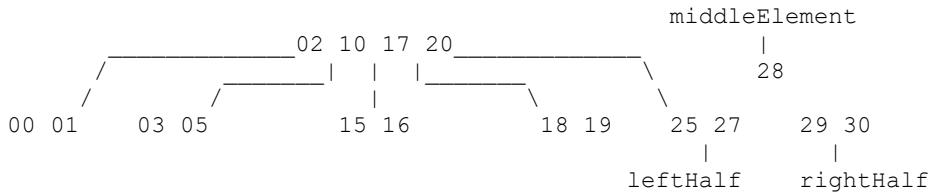
Beginning at the root, lowerBound(29) is end;

The root is not a leaf node,
 so proceed to end's left child (node 25);

Node 25 is a leaf,
so insert the new element before lowerBound(29), 30;

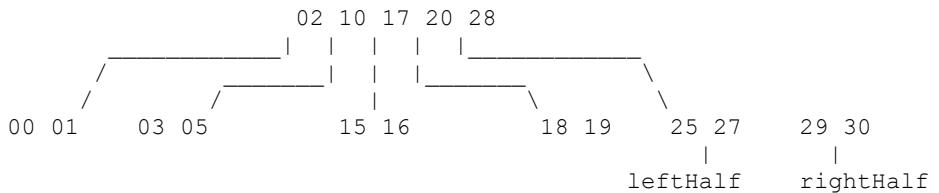


Node 25 has overflow,
so extract the middle element (28) and split the node in half;



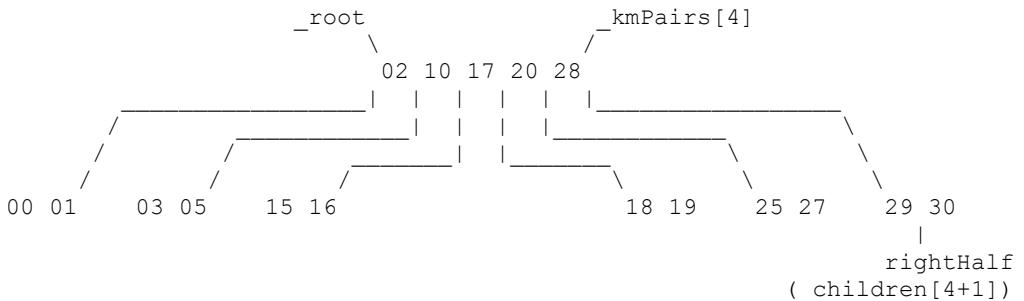
The left half is not the root,
so insert the middle element in the left half's parent (node 2);

In node 2, lowerBound(28) is end,
so insert 28 before end;



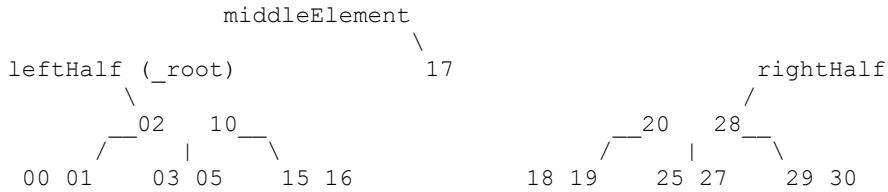
The right half becomes the right child of the promoted element (28);

In node 2,
insert a pointer to the right half at (index of promoted element) + 1;

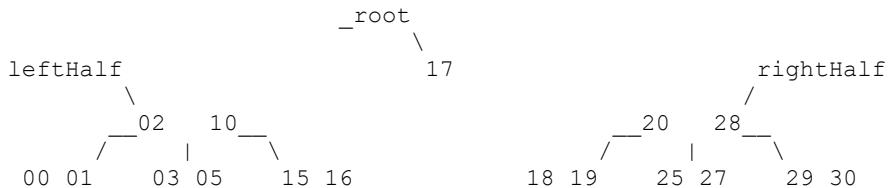


Set the right half's parent pointer to the left half's parent (node 2);

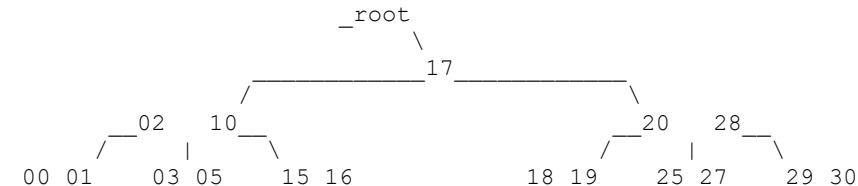
Node 2 has overflow,
so extract the middle element (17) and split the node in half;



The left half is the `_root`, so create a new root node (updating the `_root` pointer accordingly), and insert the middle element (17) there;



In the root, insert pointers to the left and right halves, and set the left and right halves' parent pointers to the root;



Keeping track of the head and tail nodes is relatively simple: insertion has no effect on the `_head`, and the `_tail` only changes when split (in which case the new tail is the right half of the original tail). To demonstrate this, let's review the previous sequence of insertions:

```

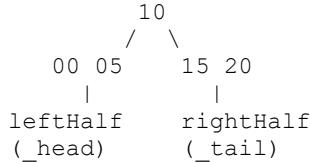
insert(10);
insert(0);
insert(15);
insert(5);

_head / _tail
\ 00 05 10 15

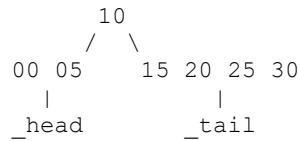
insert(20);

_head / _tail
\ 00 05 10 15 20
  
```

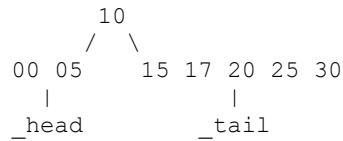
Promote 10, split node 0:
 $_head$ is unchanged;
 $_tail$ was split, so $_tail$ becomes the right half;



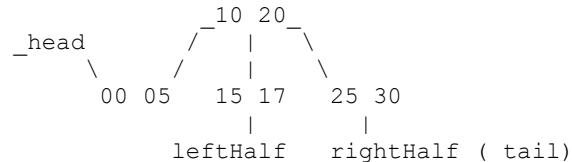
`insert(30);
insert(25);`



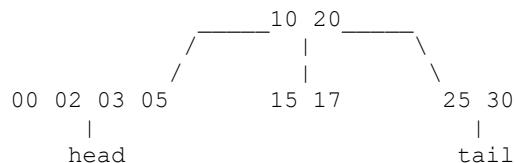
`insert(17);`



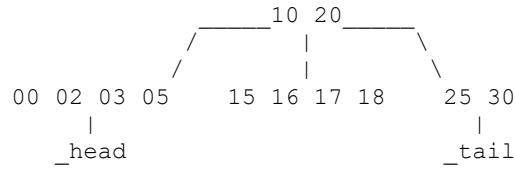
Promote 20, split node 15:
 $_head$ is unchanged;
 $_tail$ was split, so $_tail$ becomes the right half;



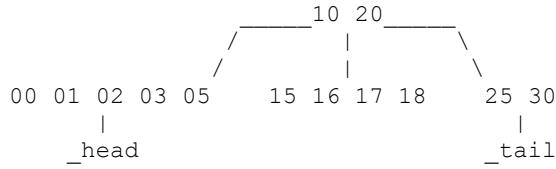
`insert(3);
insert(2);`



`insert(16);
insert(18);`

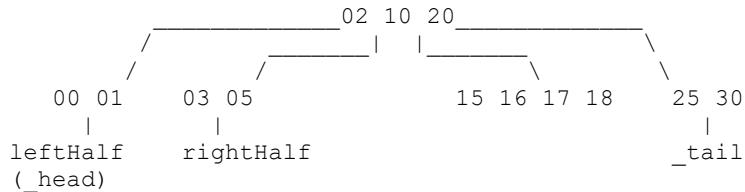


`insert(1);`

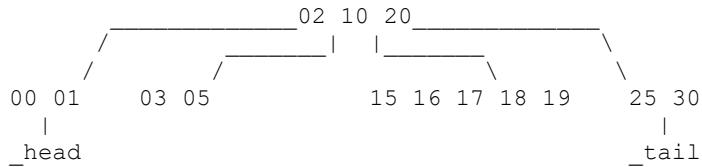


Promote 2, split node 0:

_head is unchanged;
`_tail` was not split, so `_tail` is unchanged;

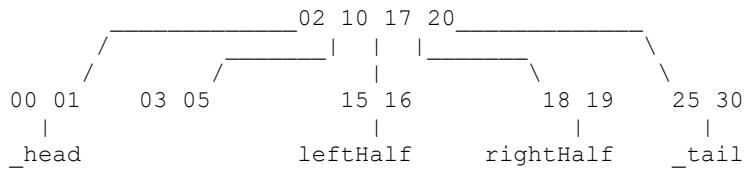


`insert(19);`

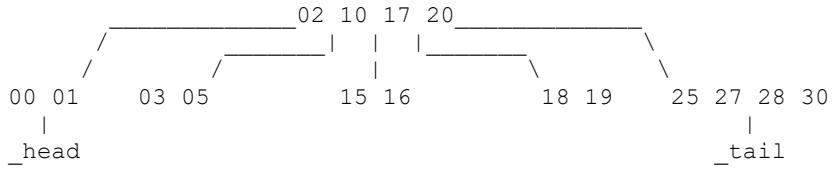


Promote 17, split node 15:

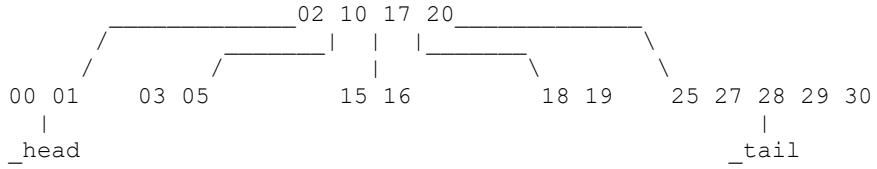
_head is unchanged;
`_tail` was not split, so `_tail` is unchanged;



`insert(28);`
`insert(27);`

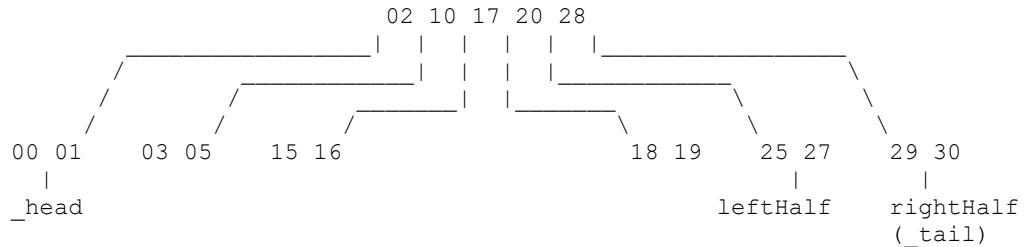


```
insert(29);
```



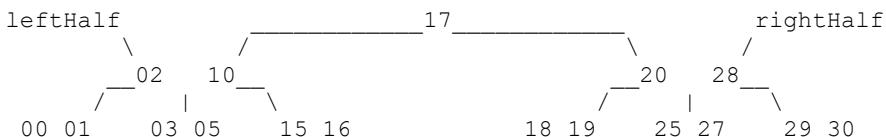
Promote 28, split node 25:

```
_head is unchanged;
_tail was split, so _tail becomes the right half;
```



Promote 17 (create new root), split node 2:

```
_head is unchanged;
_tail was not split, so _tail is unchanged;
```



We're now ready to implement *BTree*'s *insert* method (*BTree.h*, line 46),

```
void insert(const value_type& newElement);
```

For now the function doesn't return anything, but we'll update it with the standard return type (*pair<iterator, bool>*) later on (after creating *BTree::iterator*). That's why the implementation resides in a separate header file (*insert_1.h*), as opposed to *memberFunctions_1.h*.

If the tree is empty, we simply create a new root node, place the new element there, update the *_size*, and set the *_head* and *_tail* to the root (*insert_1.h*, lines 6-17).

If the tree isn't empty, we proceed to find the insertion point (leaf node) for the new element, based on its key value. This is accomplished via the function (*BTree.h*, line 69)

```
NewKeySearch _findInsertionPointForNewKey(const key_type& newKey) const;
```

This function returns a *NewKeySearch* object (lines 55-60), which contains a boolean flag *newKeyAlreadyExists*, along with a *node* pointer and *index* value.

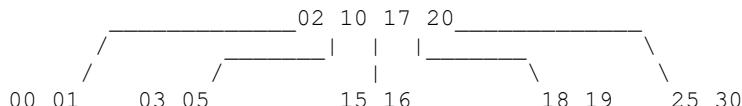
- If *newKeyAlreadyExists* is *true*, then *node* and *index* refer to the location of the preexisting element whose key value matches *newKey*
- If *newKeyAlreadyExists* is *false*, then *node* and *index* refer to the location where the new element should be inserted

If the new key already exists, then we don't insert the new element because we can't have duplicate keys (*insert_1.h*, lines 19-22); otherwise, we insert the new element at the appropriate location, update the *_size*, and balance the tree (lines 24-27).

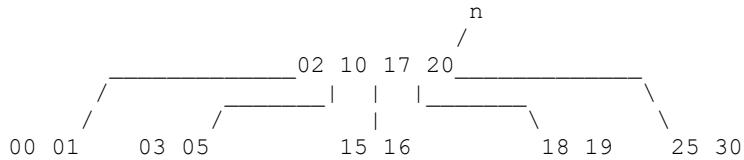
Before writing the *_balanceOnInsert* method, let's implement *_findInsertionPointForNewKey* (*memberFunctions_1.h*, lines 111-146). The *search* object (line 116) is our return value, while *n* and *i* (lines 118-119) are the current node (beginning at the root) and index. In each iteration of the loop, we find the lower bound of the *newKey* (line 123), then proceed along one of three branches:

- If the lower bound is a valid element (i.e. not *end*), and the lower bound's key value matches the *newKey*, then we set *newKeyAlreadyExists* to *true*, exit the loop, save the current node and index, and return the *search* result (lines 125-130, 142-145). To test the keys for equality (line 126), we're using the *isReflexivelyEqual* function from Volume 1 (provided by the header *predicates.h*, included in line 1).
- Otherwise, if the lower bound has a left child, we proceed to the left child for the next iteration of the loop (lines 131-134).
- Otherwise, the *newKey* doesn't already exist in the tree and we've reached a leaf node, so we set *newKeyAlreadyExists* to *false*, exit the loop, save the current node and index, and return the *search* result (lines 135-139, 142-145).

Consider, for example, the tree



```
_findInsertionPointForNewKey(28);
n = _root;
```



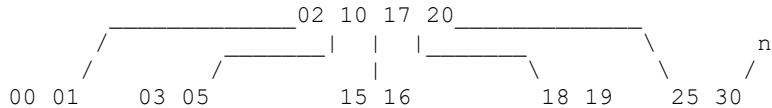
Iteration 1:

```

i = n->lowerBound(28);      // i = 4 (end);

n->hasElement(4) is false;

n->hasLeftChild(4) is true;
n = n->leftChild(4);
  
```



Iteration 2:

```

i = n->lowerBound(28);      // i = 1;

n->hasElement(1) is true;
isReflexivelyEqual(30, 28) is false;

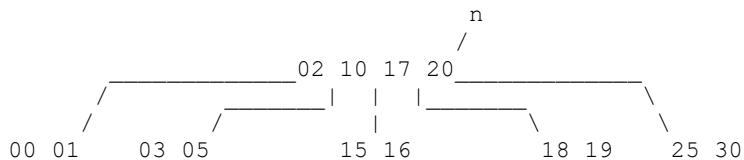
n->hasLeftChild(1) is false;

search.newKeyAlreadyExists = false;
break;

search.node = node 30;
search.index = 1;
  
```

_findInsertionPointForNewKey(15);

n = _root;



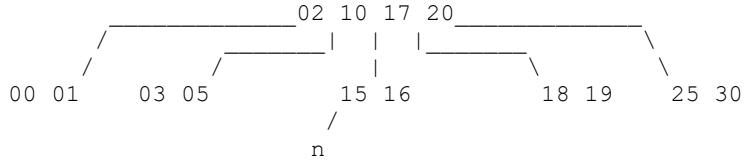
Iteration 1:

```

i = n->lowerBound(15);      // i = 2;

n->hasElement(2) is true;
isReflexivelyEqual(17, 15) is false;
  
```

```
n->hasLeftChild(2) is true;
n = n->leftChild(2);
```



Iteration 2:

```
i = n->lowerBound(15);      // i = 0;

n->hasElement(0) is true;
isReflexivelyEqual(15, 15) is true;
search.newKeyAlreadyExists = true;
break;

search.node = node 15;
search.index = 0;
```

After inserting the new element at the correct location (*insert_1.h*, line 24), we balance the tree (line 27) via the function (*BTree.h*, line 71)

```
void _balanceOnInsert(Node* n);
```

where *n* is the leaf node containing the newly inserted element. As shown earlier, the procedure is

```
_balanceOnInsert(Node* n)
{
    If n has overflow
    {
        Extract the middle element,
        and split n into left and right halves;

        If the left half is the tail,
        the tail becomes the right half;

        If the left half is not the root
        {
            Promote the middle element to the parent node;
            Attach the right half to the parent;
            Proceed to the parent and check for overflow;
        }
        else
        {
            Create a new root node;
            Promote the middle element to the root;
            Attach both halves to the root;
        }
    }
}
```

The first step to implementing `_balanceOnInsert` is to create a separate function for splitting nodes (`BTree.h`, line 72):

```
SplitNode _extractMiddleElementAndSplitNode (Node* n);
```

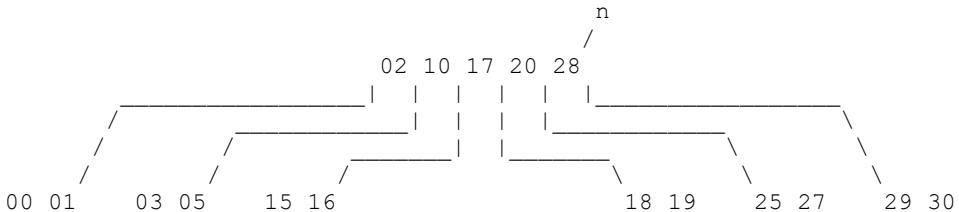
This function extracts the middle element from the given node `n`, splits `n`, and returns the middle element, left half, and right half in a `SplitNode` object (lines 62-67).

Note how the `middleElement`'s type is not `value_type` (`pair<const Key, Mapped>`), but rather `value_type_writable` (`pair<Key, Mapped>`) (lines 53, 64). This makes it a bit easier to work with `SplitNodes`, as it lets us use the compiler-generated default constructor and set the `middleElement` via simple assignment.

To implement `_extractMiddleElementAndSplitNode`, we begin by constructing a `SplitNode s` (`memberFunctions_1.h`, line 197), which will be our return value. We then find the `midpoint` index and use that to save the `middleElement` (lines 198-200). The `leftHalf` is the original node and the `rightHalf` is a newly created node (lines 201-202).

We then move the right-hand portion of elements from the `leftHalf` to the `rightHalf` (lines 204-211), after which we erase the middle element (line 213). Lastly, we move the right-hand portion of children from the `leftHalf` to the `rightHalf`, attaching each child to its new parent (lines 215-221).

Consider, for example, the tree



which has an overflow at node 2 (`n`).

```
_extractMiddleElementAndSplitNode (n);

midpoint = n->totalElements() / 2;           // midpoint = 2;

s.middleElement = n->element(midpoint);      // s.middleElement = 17;
s.leftHalf = n;
s.rightHalf = _createNode();

halfTotalElements = midpoint;                  // halfTotalElements = 2;
halfTotalChildren = midpoint + 1;             // halfTotalChildren = 3;

// The leftHalf keeps 2 elements (move the remainder to the rightHalf)

leftHalf->totalElements() != halfTotalElements // 5 != 2, continue
```

Iteration 1:

```
rightHalf->pushFrontElement(leftHalf->backElement());
leftHalf->popBackElement();

leftHalf                               rightHalf
    \                                /
      02      10      17      20
    / | / | / | / |
  00 01 03 05 15 16 18 19 25 27   28
                                         \ \
                                         29 30

leftHalf->totalElements() != halfTotalElements // 4 != 2, continue
```

Iteration 2:

```
rightHalf->pushFrontElement(leftHalf->backElement());
leftHalf->popBackElement();

leftHalf                               rightHalf
    \                                /
      02      10      17
    / | / | / |
  00 01 03 05 15 16 18 19 25 27   20 28
                                         \ \
                                         29 30

leftHalf->totalElements() != halfTotalElements // 3 != 2, continue
```

Iteration 3:

```
rightHalf->pushFrontElement(leftHalf->backElement());
leftHalf->popBackElement();

leftHalf                               rightHalf
    \                                /
      02      10
    / | / | / |
  00 01 03 05 15 16 18 19 25 27   17 20 28
                                         \ \
                                         29 30

leftHalf->totalElements() != halfTotalElements // 2 == 2, break

rightHalf->popFrontElement();

leftHalf                               rightHalf
    \                                /
      02      10
    / | / | / |
  00 01 03 05 15 16 18 19 25 27   20 28

leftHalf->totalChildren() > halfTotalChildren // 6 > 3, continue
```

Iteration 1:

```
rightHalf->pushFrontChild(leftHalf->backChild());      // Move node 29
leftHalf->popBackChild();
```

```
rightHalf->frontChild()->setParent(rightHalf);
```



```
leftHalf->totalChildren() > halfTotalChildren      // 5 > 3, continue
```

Iteration 2:

```
rightHalf->pushFrontChild(leftHalf->backChild());      // Move node 25
leftHalf->popBackChild();
```

```
rightHalf->frontChild()->setParent(rightHalf);
```

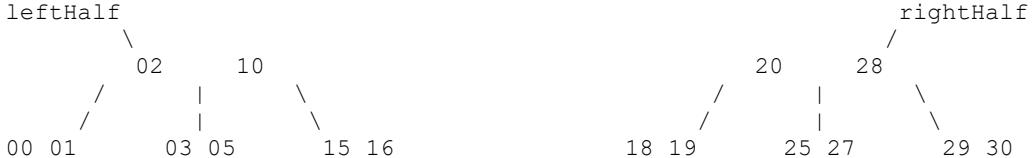


```
leftHalf->totalChildren() > halfTotalChildren      // 4 > 3, continue
```

Iteration 3:

```
rightHalf->pushFrontChild(leftHalf->backChild());      // Move node 18
leftHalf->popBackChild();
```

```
rightHalf->frontChild()->setParent(rightHalf);
```



```
leftHalf->totalChildren() > halfTotalChildren      // 3 = 3, break
```

Now that we can easily split overflowing nodes, we're ready to write `_balanceOnInsert`. The implementation (`memberFunctions_1.h`, lines 148-191) closely resembles the pseudocode shown earlier:

```

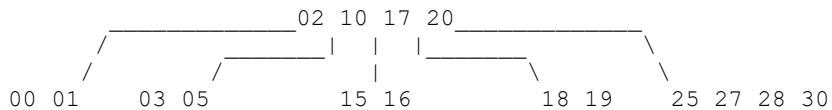
balanceOnInsert(Node* n)
{
    If n has overflow
    {
        Extract the middle element,
        and split n into left and right halves;

        If the left half is the tail,
        the tail becomes the right half;

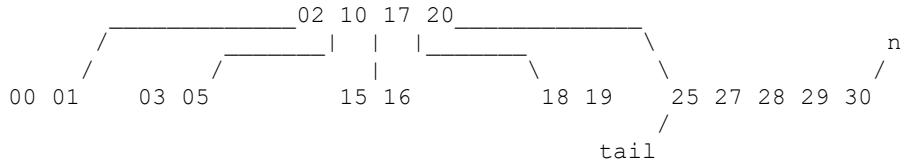
        If the left half is not the root
        {
            Promote the middle element to the parent node;      // lines 162-165
            Attach the right half to the parent;                // lines 166-168
            Proceed to the parent and check for overflow;      // line 170
        }
        else
        {
            Create a new root node;
            Promote the middle element to the root;           // line 176
            Attach both halves to the root;                   // lines 177-181
        }
    }
}
}

```

Consider, for example, the tree



`insert(29);`



`_balanceOnInsert(n);`

Iteration 1:

`n->hasOverflow() is true;`

`SplitNode s = _extractMiddleElementAndSplitNode(n);`

```

s.middleElement
      /   |   |   |
02 10 17 20
      /   |   |   \
00 01 03 05 15 16 18 19 25 27 28
                                         |
                                         s.leftHalf
                                         |
                                         _tail
                                         |
                                         s.rightHalf
                                         |
                                         29 30

s.leftHalf is the _tail;
_tail = s.rightHalf;

s.middleElement
      /   |   |   |
02 10 17 20
      /   |   |   \
00 01 03 05 15 16 18 19 25 27 28
                                         |
                                         s.leftHalf
                                         |
                                         _tail
                                         |
                                         s.rightHalf
                                         |
                                         29 30

s.leftHalf is not the _root;

Node* parent = s.leftHalf->parent();
Index i = parent->lowerBound(s.middleElement.first); // i = 4;

parent
      /   |   |   |
02 10 17 20
      /   |   |   \
00 01 03 05 15 16 18 19 25 27 28
                                         |
                                         s.leftHalf
                                         |
                                         s.rightHalf

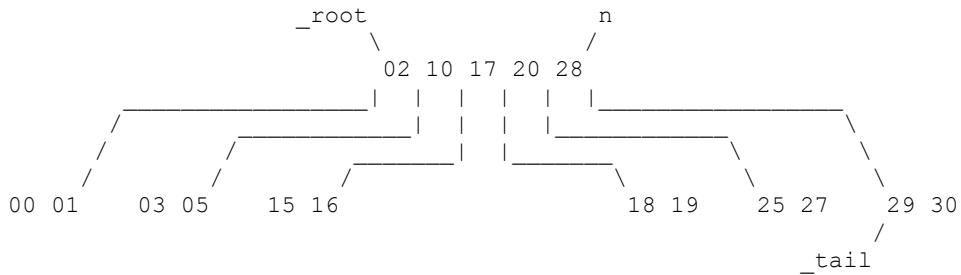
parent->insertElement(i, s.middleElement);

parent
      /   |   |   |   |
02 10 17 20 28
      /   |   |   |
00 01 03 05 15 16 18 19 25 27 29 30
                                         |
                                         s.rightHalf

parent->insertChild(i + 1, s.rightHalf);
s.rightHalf->setParent(parent);

n = parent;

```



Iteration 2:

```

n->hasOverflow() is true;

SplitNode s = _extractMiddleElementAndSplitNode(n);

    s.middleElement
    s.leftHalf
    s.root
        \ 17
            / \
            02   10
            |   |
            00   01   03   05   15   16
            |       |
            18   19   25   27
            |       |
            20   28
            |   |
            29   30
            / \
            _tail

s.leftHalf is not the _tail;
s.leftHalf is the _root;

_root = _createNode();
_root->pushBackElement(s.middleElement);

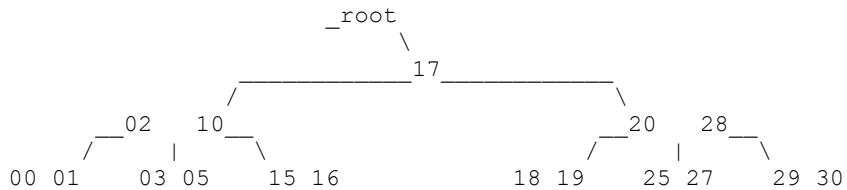
    s.leftHalf
    s.root
        \ 17
            / \
            02   10
            |   |
            00   01   03   05   15   16
            |       |
            18   19   25   27
            |       |
            20   28
            |   |
            29   30
            / \
            _tail

_root->pushBackChild(s.leftHalf);
_root->pushBackChild(s.rightHalf);

s.leftHalf->setParent(_root);
s.rightHalf->setParent(_root);

break;

```



Our test program (*main.cpp*) begins by constructing a *BTree<Traceable<int>, int>* (lines 12, 22). In each iteration of the loop, we prompt the user for a command (*i* to insert a key, *q* to quit) (lines 26-30) and branch accordingly (lines 32-37).

If the user enters *i*, we call the function *getKeyAndInsertElement* (line 14), which prompts the user for the desired key value (lines 51-54), inserts it into the tree (lines 55-59), and prints the structure of the entire tree, including the *front* and *back* elements (lines 62-65). The following sample run demonstrates the sequence of insertions that we stepped through earlier:

```

// insert 10, 0, 15, 5

key          0
key          5
key          10
key          15

Front element = (0,0)
Back element = (15,0)
  
```

```

// insert 20

key          0
parent->front 10

key          5
parent->front 10

key          10
left          0
right         15

key          15
parent->front 10

key          20
parent->front 10

Front element = (0,0)
Back element = (20,0)
  
```

```
// insert 30, 25

key          0
parent->front 10

key          5
parent->front 10

key          10
left         0
right        15

key          15
parent->front 10

key          20
parent->front 10

key          25
parent->front 10

key          30
parent->front 10

Front element = (0,0)
Back element = (30,0)
```

```
// insert 17

key          0
parent->front 10

key          5
parent->front 10

key          10
left         0
right        15

key          15
parent->front 10

key          17
parent->front 10

key          20
left         15
right        25

key          25
parent->front 10
```

```
key          30
parent->front 10

Front element = (0,0)
Back element = (30,0)
```

```
// insert 3, 2, 16, 18
```

```
key          0
parent->front 10

key          2
parent->front 10

key          3
parent->front 10

key          5
parent->front 10

key          10
left        0
right       15

key          15
parent->front 10

key          16
parent->front 10

key          17
parent->front 10

key          18
parent->front 10

key          20
left        15
right       25

key          25
parent->front 10

key          30
parent->front 10

Front element = (0,0)
Back element = (30,0)
```

```
// insert 1

key          0
parent->front 2

key          1
parent->front 2

key          2
left         0
right        3

key          3
parent->front 2

key          5
parent->front 2

key          10
left         3
right        15

key          15
parent->front 2

key          16
parent->front 2

key          17
parent->front 2

key          18
parent->front 2

key          20
left         15
right        25

key          25
parent->front 2

key          30
parent->front 2

Front element = (0,0)
Back element = (30,0)
```

```
// insert 19

key          0
parent->front 2
```

```

key          1
parent->front 2

key          2
left         0
right        3

key          3
parent->front 2

key          5
parent->front 2

key          10
left         3
right        15

key          15
parent->front 2

key          16
parent->front 2

key          17
left         15
right        18

key          18
parent->front 2

key          19
parent->front 2

key          20
left         18
right        25

key          25
parent->front 2

key          30
parent->front 2

Front element = (0,0)
Back element = (30,0)

```

```
// insert 28, 27
```

```

key          0
parent->front 2

key          1
parent->front 2

```

```

key          2
left         0
right        3

key          3
parent->front 2

key          5
parent->front 2

key          10
left         3
right        15

key          15
parent->front 2

key          16
parent->front 2

key          17
left         15
right        18

key          18
parent->front 2

key          19
parent->front 2

key          20
left         18
right        25

key          25
parent->front 2

key          27
parent->front 2

key          28
parent->front 2

key          30
parent->front 2

Front element = (0,0)
Back element = (30,0)

```

```
// insert 29
```

```

key          0
parent->front 2

```

```
key          1
parent->front 2

key          2
parent->front 17
left         0
right        3

key          3
parent->front 2

key          5
parent->front 2

key          10
parent->front 17
left         3
right        15

key          15
parent->front 2

key          16
parent->front 2

key          17
left         2
right        20

key          18
parent->front 20

key          19
parent->front 20

key          20
parent->front 17
left         18
right        25

key          25
parent->front 20

key          27
parent->front 20

key          28
parent->front 17
left         25
right        29

key          29
parent->front 20
```

```
key          30
parent->front 20

Front element = (0,0)
Back element = (30,0)
```

```
// quit

~ 1
~ 0
~ 5
~ 3
~ 16
~ 15
~ 10
~ 2
~ 19
~ 18
~ 27
~ 25
~ 30
~ 29
~ 28
~ 20
~ 17

All Traceables destroyed
```


6.6: Iterative In-Order Traversal

Source files and folders

- *BTree/2*
- *BTree/common/Location.h*
- *BTreeInOrderIterative*

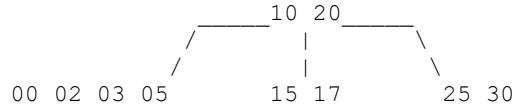
Chapter outline

- *Using iteration to find the in-order predecessor / successor of a given element*

The *Location* class (*BTree.h*, lines 33-46) stores the location of a particular element in the tree:

- *node* (line 44) is a pointer to the node containing the element
- *index* (line 45) is the element's relative position (offset) within the *node*

In the tree



for example,

- the *Location* of element 5 is (*node 0, index 3*)
- the *Location* of element 17 is (*node 15, index 1*)
- the *Location* of element 25 is (*node 25, index 0*)

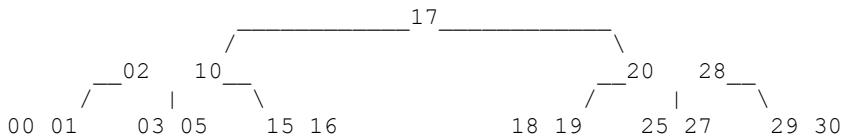
The default constructor (*BTree.h*, line 38) creates a *Location* with a null *node* and *index* of 0 (*Location.h*, lines 3-9). Alternatively, we can construct a *Location* from our own *node* and *index* (*BTree.h*, line 39 / *Location.h*, lines 11-18).

The equality and inequality operators (*BTree.h*, lines 41-42) let us compare *Locations* to determine whether they refer to the same element (*Location.h*, lines 20-32).

The function (*BTreeInOrderIterative.h*, lines 9-10)

```
template <class Location>
Location bTreeInOrderSuccessor(Location current);
```

takes the location of the *current* element and returns the location of the next element in the in-order sequence (lines 49-84). The variables *n* and *i* (lines 52-55) are initialized to the current element's node and index. To demonstrate the 4 main branches of the algorithm, let's use the tree



Case 1: The current element does not reside in a leaf node (lines 57-65)

To find the successor:

- Proceed to the element's right child
- Descend the tree leftward, until reaching a leaf node
- The successor is the front element of that leaf node

From element 2:

- Proceed to the element's right child (node 3)
- Descend the tree leftward, until reaching a leaf (node 3)
- The successor is the front element of that leaf (element 3)

From element 10:

- Proceed to the element's right child (node 15)
- Descend the tree leftward, until reaching a leaf (node 15)
- The successor is the front element of that leaf (element 15)

From element 17:

- Proceed to the element's right child (node 20)
- Descend the tree leftward, until reaching a leaf (node 18)
- The successor is the front element of that leaf (element 18)

From element 20:

- Proceed to the element's right child (node 25)
- Descend the tree leftward, until reaching a leaf (node 25)
- The successor is the front element of that leaf (element 25)

From element 28:

- Proceed to the element's right child (node 29)
- Descend the tree leftward, until reaching a leaf (node 29)
- The successor is the front element of that leaf (element 29)

Case 2: The current element resides in a leaf node,
and it is not the back element (lines 66-69)

The successor is the next element in the node:

- From element 0, the successor is element 1

- From element 3, the successor is element 5
 - From element 15, the successor is element 16
 - From element 18, the successor is element 19
 - From element 25, the successor is element 27
 - From element 29, the successor is element 30
-

Case 3: The current element resides in a leaf node,
it is the back element,
and its node is a left child (lines 70-74)

To find the successor:

- Proceed to the parent node
- The successor is the lower bound of the current element's key value

From element 1:

- Proceed to the parent (node 2)
- The successor is the lower bound of 1 (element 2)

From element 5:

- Proceed to the parent (node 2)
- The successor is the lower bound of 5 (element 10)

From element 19:

- Proceed to the parent (node 20)
- The successor is the lower bound of 19 (element 20)

From element 27:

- Proceed to the parent (node 20)
 - The successor is the lower bound of 27 (element 28)
-

Case 4: The current element resides in a leaf node,
it is the back element,
and its node is not a left child (lines 75-83)

To find the successor:

- Climb the tree, until reaching a node that is a left child, then proceed to that node's parent. The successor is the lower bound of the current element's key value.
- If we climb past the root before finding a left child, then the current element is the final one in the sequence and the successor is null.

From element 16:

- Climb the tree, until reaching a node that is a left child (node 2), then proceed to that node's parent (node 17). The successor is the lower bound of 16 (element 17).

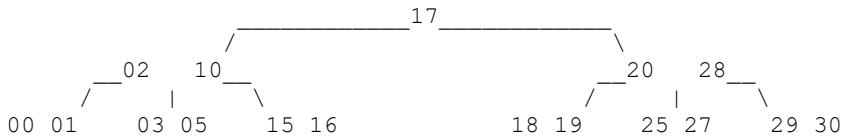
From element 30:

- Climbing all the way up the tree (past node 20 and the root, node 17), no left child was found. Element 30 is therefore the final element in the sequence, and the successor is null.

The function (*BTreeInOrderIterative.h*, lines 6-7)

```
template <class Location>
Location bTreeInOrderPredecessor(Location current);
```

takes the location of the *current* element and returns the location of the previous element in the in-order sequence (lines 12-47). This function is the mirror image of *bTreeInOrderSuccessor*. To demonstrate the 4 main branches, we'll once again use the tree



Case 1: The current element does not reside in a leaf node (lines 20-28)

To find the successor:

- Proceed to the element's left child
- Descend the tree rightward, until reaching a leaf node
- The predecessor is the back element of that leaf node

From element 2:

- Proceed to the element's left child (node 0)
- Descend the tree rightward, until reaching a leaf node (node 0)
- The predecessor is the back element of that leaf node (element 1)

From element 10:

- Proceed to the element's left child (node 3)
- Descend the tree rightward, until reaching a leaf node (node 3)
- The predecessor is the back element of that leaf node (element 5)

From element 17:

- Proceed to the element's left child (node 2)
- Descend the tree rightward, until reaching a leaf node (node 15)

- The predecessor is the back element of that leaf node (element 16)

From element 20:

- Proceed to the element's left child (node 18)
- Descend the tree rightward, until reaching a leaf node (node 18)
- The predecessor is the back element of that leaf node (element 19)

From element 28:

- Proceed to the element's left child (node 25)
 - Descend the tree rightward, until reaching a leaf node (node 25)
 - The predecessor is the back element of that leaf node (element 27)
-

Case 2: The current element resides in a leaf node,
and it is not the front element (lines 29-32)

The predecessor is the previous element in the node:

- From element 1, the predecessor is element 0
 - From element 5, the predecessor is element 3
 - From element 16, the predecessor is element 15
 - From element 19, the predecessor is element 18
 - From element 27, the predecessor is element 25
 - From element 30, the predecessor is element 29
-

Case 3: The current element resides in a leaf node,
it is the front element,
and its node is a right child (lines 33-37)

To find the predecessor:

- Proceed to the parent node
- Find the lower bound of the current element's key value
- The predecessor is the preceding element

From element 3:

- Proceed to the parent (node 2)
- Find the lower bound of the current element's key value (element 10)
- The predecessor is the preceding element (element 2)

From element 15:

- Proceed to the parent (node 2)
- Find the lower bound of the current element's key value (end)
- The predecessor is the preceding element (element 10)

From element 25:

- Proceed to the parent (node 20)
- Find the lower bound of the current element's key value (element 28)
- The predecessor is the preceding element (element 20)

From element 29:

- Proceed to the parent (node 20)
 - Find the lower bound of the current element's key value (end)
 - The predecessor is the preceding element (element 28)
-

Case 4: The current element resides in a leaf node,
it is the front element,
and its node is not a right child (lines 38-46)

To find the predecessor:

- Climb the tree, until reaching a node that is a right child, then proceed to that node's parent. Find the lower bound of the current element's key value. The predecessor is the preceding element.
- If we climb past the root before finding a right child, then the current element is the first one in the sequence and the predecessor is null.

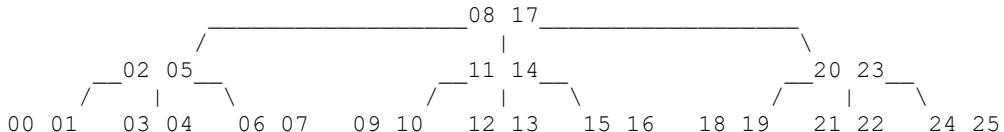
From element 18:

- Climb the tree, until reaching a node that is a right child (node 20), then proceed to that node's parent (node 17). Find the lower bound of the current element's key value (end). The predecessor is the preceding element (element 17).

From element 0:

- Climbing all the way up the tree (past node 2 and the root, node 17), no right child was found. Element 0 is therefore the first element in the sequence, and the predecessor is null.

Our test program (*main.cpp*) begins by constructing a tree with the keys [0, 25] (lines 8-17):



We then initialize the *current* Location to that of the front element (key 0) (line 21). In each iteration of the loop, we print the current key value (line 25) and proceed to the in-order successor (line 23), until we've passed the final element in the sequence (line 22). The resultant output is

In-order successors:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

We then repeat the process, beginning at the back element (key 25) and stepping through the in-order predecessors (lines 30-35). The resultant output is

In-order predecessors:

25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

6.7: Implementing the Iterators

Source files and folders

- *BTree/3*
- *BTree/common/BTreeIter.h*
- *BTree/common/memberFunctions_2.h*
- *BTree/insert/insert_2.h*

Chapter outline

- *Implementing BTree's iterator, reverse_iterator, const_iterator, and const_reverse_iterator*
- *Implementing BTree's find method*
- *Updating BTree's insert method with the standard return type (pair<iterator, bool>)*

A B-tree iterator has a single template parameter, *BTree* (*BTreeIter.h*, lines 10-11), which is the type of the iterator's host tree. From there we import the *pointer*, *reference*, *difference_type*, and *value_type* (lines 16-19). This is a bidirectional iterator, so its *iterator_category* is *bidirectional_iterator_tag* (line 20).

The private data members (lines 39-40) are

- *_bTree*, a pointer to the iterator's host tree
- *_location*, the referent element's *Location* in the tree

The default constructor (line 22) doesn't do any explicit initialization (lines 43-47), while the private constructor (line 37) initializes *_bTree* and *_location* according to the given arguments (lines 118-124). This constructor will be used from within the *BTree* class itself (in methods such as *begin*, *end*, etc.), which is why we're granting friend privileges to the host tree (line 14).

The comparison operators (lines 24-25) determine whether two iterators refer to the same element, by simply comparing their *Locations* (lines 49-59).

The dereference and member access operators (lines 26-27) simply call the node's *element* method, using the referent element's index value (lines 61-71).

The prefix increment operator (line 29) points to the next element via *bTreeInOrderSuccessor* (lines 73-79). The prefix decrement operator (line 30) contains 2 branches. If we're not currently at the *end*, we point to the previous element by simply calling *bTreeInOrderPredecessor* (lines 84-88). But if we are at the *end*, we must manually reset the *_location* to that of the *back* element (lines 89-93) (since *bTreeInOrderPredecessor* doesn't work on a null starting node).

The postfix increment / decrement operators (lines 31-32) provide the standard behavior (point to the next / previous element, and return an iterator to the original element) (lines 98-116).

Moving on to the *BTree* class (*BTree.h*),

- *iterator* is an alias of *BTreeIter*<*BTree*> (line 36)
- *reverse_iterator* is an alias of *dss::ReverseIter*<*iterator*> (line 37)
- *const_iterator* is an alias of *dss::ConstIter*<*BTree*> (line 32)
- *const_reverse_iterator* is an alias of *dss::ReverseIter*<*const_iterator*> (line 33)

Recall from Volume 1 that the *ReverseIter* class creates a *reverse_iterator* from any standard iterator class (*dss/ReverseIter/ReverseIter.h*). Similarly, the *ConstIter* class creates a *const_iterator* from any standard container class (*dss/ConstIter/ConstIter.h*).

BTree's *begin* method (*BTree.h*, line 75) returns an iterator to the *front* element (*memberFunctions_2.h*, lines 40-45). To implement the *const* version of *begin* (*BTree.h*, line 63), we'll use the same technique that we applied in Volume 1: we call the non-*const* version (*memberFunctions_2.h*, line 9), which returns an *iterator*, which is then implicitly converted to a *const_iterator*.

end (*BTree.h*, lines 64, 76) returns an *iterator* / *const_iterator* to the one-past-the-last element (*memberFunctions_2.h*, lines 12-17, 47-52).

rbegin (*BTree.h*, lines 65, 77) returns a *reverse_iterator* / *const_reverse_iterator* to the first element of the reverse sequence (*memberFunctions_2.h*, lines 19-24, 54-59).

rend (*BTree.h*, lines 66, 78) returns a *reverse_iterator* / *const_reverse_iterator* to the one-past-the-last element of the reverse sequence (*memberFunctions_2.h*, lines 26-31, 61-66).

Now that we've implemented the iterators, let's write *BTree*'s *find* methods (*BTree.h*, lines 69, 81),

```
const_iterator find(const key_type& key) const;
iterator find(const key_type& key);
```

which return an *iterator* / *const_iterator* to the element with the given *key* value (or *end*, if the *key* doesn't exist). To implement these functions, we'll first write a separate function (line 104)

```
Location _location(const key_type& key) const;
```

which returns the *Location* of the element with the given *key* value. The implementation of this function is very similar to that of *_findInsertionPointForNewKey*; the main difference lies in the return type and return value (*memberFunctions_2.h*, lines 75-101).

To implement *find*, we simply obtain the *Location* of the desired *key* value (via *_location*), then use that to construct an iterator to the corresponding element (lines 68-73). The *const* version of *find* is then implemented using the same technique as the *const* versions of *begin* and *end* (lines 33-38).

Lastly, we can update *BTree*'s *insert* method to return a *pair*<*iterator*, *bool*> (*BTree.h*, line 82):

- If the *newElement* was inserted, then *pair::second* is *true* and *pair::first* points to the newly

inserted element.

- If the *newElement* was not inserted (because the tree already contains an element with the same key value), then *pair::second* is *false* and *pair::first* points to the preexisting element.

The new implementation of *insert* (*insert_2.h*) is nearly identical to the original version, with only a few minor differences:

- After inserting the very first element, we return a *pair* containing an iterator to the first element and *true* (line 19).
- If the new key already exists, we return a *pair* containing an iterator to the preexisting element and *false* (lines 25-27).
- After inserting any other element, we obtain an iterator to the newly inserted element via *find*, then return a *pair* containing that iterator and *true* (line 34). The reason that we need to re-find the newly inserted element is that *_balanceOnInsert* (line 32) may have moved it to a different location.

Our test program (*main.cpp*) is similar to what we wrote in Chapter 6.5, but this time around we're using *int* (as opposed to *Traceable<int>*) keys (line 10). We've also updated *getKeyAndInsertElement* to use the *pair* returned from *BTree*'s *insert* method. When inserting a new element (line 57), we let the user know whether the element was inserted (lines 59-60), or a duplicate key was found (lines 61-62). We then demonstrate the iterators via *printContainer* and *printContainerReverse* (lines 64-69).

To test *BTree*'s *find* method, the main loop contains a new menu choice, *f* for find (line 25). This branch (lines 35-36) calls a new function, *getKeyAndFindElement* (line 13), which prompts the user for the desired key value and searches the tree (lines 76-81). If the corresponding element was found, we print it (lines 83-84); otherwise, we print a message saying that the desired key wasn't found (lines 85-86).

In the following sample run, we begin by creating the same tree from Chapter 6.5:

```
// insert 10, 0, 15, 5, 20, 30, 25, 17, 3, 2, 16, 18, 1, 19, 28, 27, 29
```

After inserting 29, the resultant output is

Forward:

```
(0,0) (1,0) (2,0) (3,0) (5,0) (10,0) (15,0) (16,0) (17,0) (18,0) (19,0) (20,0)
(25,0) (27,0) (28,0) (29,0) (30,0)
```

Reverse:

```
(30,0) (29,0) (28,0) (27,0) (25,0) (20,0) (19,0) (18,0) (17,0) (16,0) (15,0)
(10,0) (5,0) (3,0) (2,0) (1,0) (0,0)
```

Trying to insert some preexisting keys, such as [15, 3, 28], generates the output

```
Element (15,0) already exists  
Element (3,0) already exists  
Element (28,0) already exists
```

Searching for existing keys, such as 0 and 20, generates the output

```
Found element (0,0)  
Found element (20,0)
```

Searching for nonexistent keys, such as 4 and 21, generates the output

```
Key 4 not found  
Key 21 not found
```

6.8: Erasing Elements

Source files and folders

- *BTree/4*
- *BTree/common/memberFunctions_3.h*

Chapter outline

- *Implementing BTree's erase method*

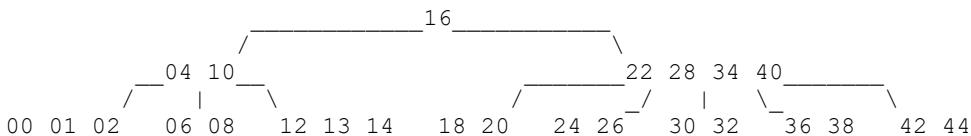
Now that we have working iterators, we can implement the function (*BTree.h*, line 83)

```
iterator erase(const_iterator element);
```

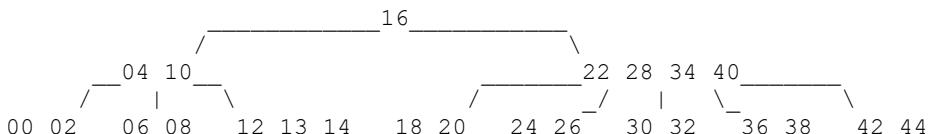
which erases the given *element* and returns an iterator to the next element in the sequence (the in-order successor). This is a bit more involved than insertion, but like any other complex procedure, we can break it up into smaller, more manageable parts. The implementation of *erase* (*memberFunctions_3.h*, lines 11-31) outlines the major steps:

- Find a *leaf element* (an element residing in a leaf node) for deletion, and remove it from the tree (lines 11-13)
- Balance the tree (line 15)
- Update the *_size*, and reset the *_root*, *_head*, and *_tail* pointers (if no more elements remain) (lines 17-26)
- Return an iterator to the deleted element's successor (if it has one), or *end* (if not) (lines 28-31)

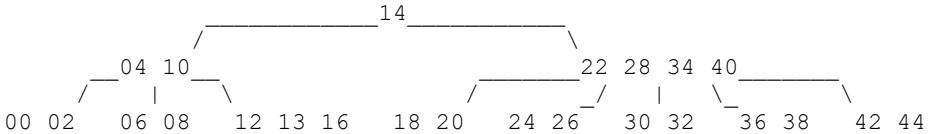
The first step, finding a leaf element, is similar to what we do when erasing an element from a binary tree. If the element that we want to erase already resides in a leaf node, we can immediately remove it from the tree; otherwise, we swap it with its in-order predecessor (thus turning it into a leaf element), after which we can easily remove it. Consider, for example, the following tree:



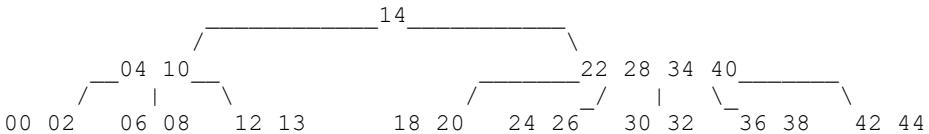
Element 1 is a leaf element, so we can easily remove it from the tree by simply deleting it from its node:



Element 16, however, is a non-leaf element. To prepare it for deletion, we swap it with its predecessor (14), thus turning it into a leaf element:



We can now easily remove element 16 from the tree by deleting it from the leaf node:



In the function (*BTree.h*, line 118)

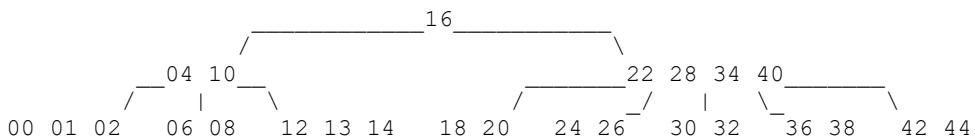
```
LeafElement _getLeafElementForDeletion(const_iterator element);
```

element points to the unwanted element (the one we plan to erase). If the unwanted element is already a leaf element, the function returns a *LeafElement* object (lines 105-111) containing:

- *node*, a pointer to the node containing the unwanted element
- *index*, the unwanted element's position within the *node*
- *hasSuccessor*, a boolean flag indicating whether the unwanted element has a successor (*true*) or not (*false*)
- *successorKey*, the key value of the unwanted element's successor (if *hasSuccessor* is *true*)

If the unwanted element is not a leaf element, *_getLeafElementForDeletion* swaps it with its predecessor (thus turning it into a leaf element), then returns a *LeafElement* object.

Suppose, for example, that we want to erase element 13 from the tree

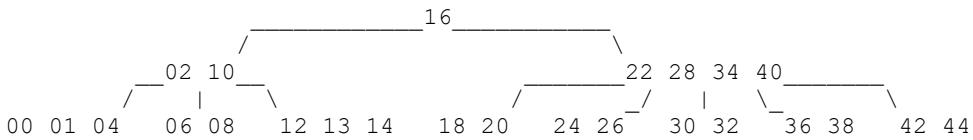


Element 13 is already a leaf element, so *_getLeafElementForDeletion* (*memberFunctions_3.h*, line 11) returns:

- a pointer to *node* 12 (the node containing element 13)
- *index* 1 (element 13's position within node 0)
- *hasSuccessor* *true* (element 13 has a successor)
- *successorKey* 14 (the key value of element 13's successor)

We can then remove element 13 from the tree by erasing the element at node 12, index 1 (line 13). After balancing the tree and updating the private members (lines 15-26), we return an iterator to element 13's successor (14) (lines 28-31). The reason that we need to re-find the successor (line 29) is that `_balanceOnErase` (line 15) (which we'll implement later) may have moved it to a different location.

If we want to erase a non-leaf element, such as 4, `_getLeafElementForDeletion` (line 11) swaps element 4 with its predecessor (2):



Element 4 thus becomes a leaf element (occupying the former location of its predecessor), and the function returns:

- a pointer to *node 0* (the node containing element 4)
- *index 2* (element 4's position within node 0)
- *hasSuccessor true* (element 4 has a successor)
- *successorKey 6* (the key value of element 4's successor)

We can then remove element 4 from the tree by erasing the element at node 0, index 2 (line 13). After balancing the tree and updating the private members (lines 15-26), we return an iterator to element 4's successor (6) (lines 28-31).

Before implementing `_getLeafElementForDeletion`, let's write a helper method (*BTree.h*, line 119),

```
void _swapElements(Location a, Location b);
```

This function swaps the the element (*KmPair*) at *Location a* with the element at *Location b* (*memberFunctions_3.h*, lines 75-80). We'll use it to perform the predecessor swap in `_getLeafElementForDeletion`.

To implement `_getLeafElementForDeletion`, we begin by constructing our return value *trash*, which is the leaf element that will eventually be removed from the tree (line 39). We then retrieve the *element's* Location *e*, and use it to find the successor (lines 41-42).

If the unwanted element resides in a leaf node (line 44), we determine whether or not it has a successor and set *hasSuccessor* and *successorKey* accordingly (lines 46-54). We then save the unwanted element's location (*node* and *index*) (lines 56-57), and return the result (line 72).

If the unwanted element doesn't reside in a leaf node (line 59), then it's guaranteed to have a successor, so we set *hasSuccessor* and *successorKey* accordingly (lines 61-62). It's also guaranteed to have a predecessor, so we swap the unwanted element with its predecessor (lines 64-66), which is guaranteed

to reside in a leaf node. We then save the unwanted element's new location (the predecessor's original location) (lines 68-69), and return the result (line 72).

Now that we can easily move the unwanted element to a leaf node and remove it from the tree, let's take a look at the balancing function (*BTree.h*, line 120),

```
void _balanceOnErase(Node* n);
```

where *n* is the leaf node from which we removed the unwanted element (*memberFunctions_3.h*, line 15). The algorithm consists of 7 cases:

```
_balanceOnErase(Node* n)
{
    if n is the root,
        return;

    if n is at least half full,
        return;

    if n's left sibling is more than half full
    {
        perform a right rotation on n's parent element;
        return;
    }
    else if n's right sibling is more than half full
    {
        perform a left rotation on n's parent element;
        return;
    }
    else if n's parent is the root and the root contains 1 element
    {
        merge the root with its two children;
        return;
    }
    else if n has a left sibling
    {
        merge n with its left sibling;
        proceed to n's parent;
    }
    else
    {
        merge n with its right sibling;
        proceed to n's parent;
    }
}
```

Before writing *_balanceOnErase*, let's examine each case and (if necessary) implement a subroutine to handle it.

Case 1: *n* is the root

Because the unwanted element is guaranteed to reside in a leaf node, if we removed an element directly from the root, then the root must be a leaf node. This means that the entire tree consists of a single node (the root), so no balancing is necessary.

```

root
/
00 05 10 15

erase(10);

root
/
00 05 15

erase(5);

root
/
00 15

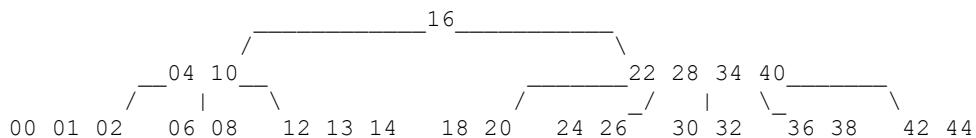
erase(0);

root
/
15

```

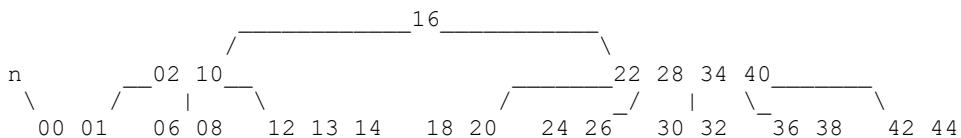
Case 2: n is at least half full

If n is at least half full, no balancing is necessary because n still contains an allowable number of elements.



```
erase(4);
```

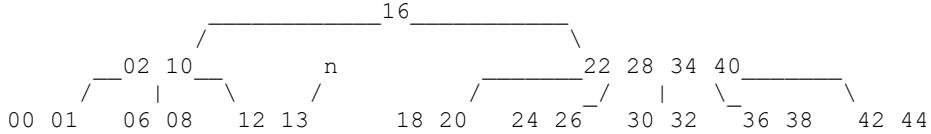
element 4 is not a leaf element, so swap it with its predecessor (2),
then remove element 4 from node 0;



n is at least half full (contains 2/4 elements),
so no balancing is necessary;

```
erase(14);
```

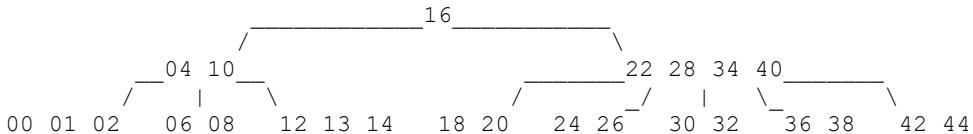
element 14 is a leaf element,
so we can immediately remove it from node 12;



n is at least half full (contains 2/4 elements),
so no balancing is necessary;

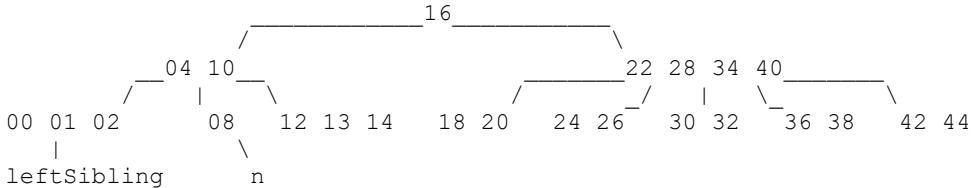
Case 3: n is less than half full, and n's left sibling is more than half full

If we arrive at Case 3, then n is 1 element short of being half full. If n's left sibling has an element to spare, we perform a right rotation on n's parent element. To perform a right rotation, we demote n's parent element and promote the spare element from n's left sibling.



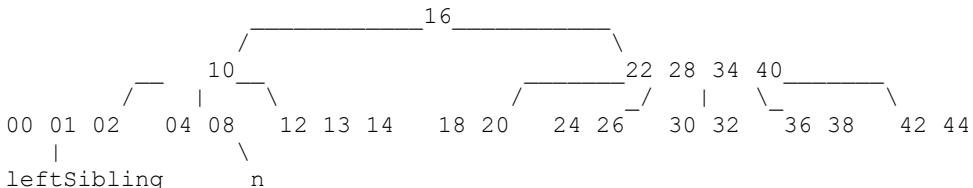
`erase(6);`

element 6 is a leaf element,
so we can immediately remove it from node 6;

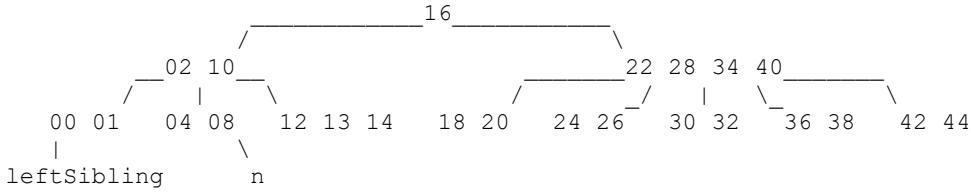


n is less than half full and n's left sibling is more than half full,
so perform a right rotation on n's parent element (4);

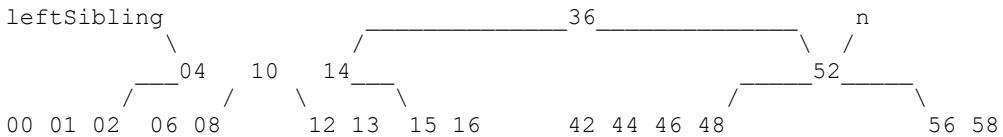
demote n's parent element by moving it to the front of n;



promote the spare element from the left sibling (move the left sibling's back element, 2, to the location formerly occupied by n's parent element);



If the left sibling is not a leaf node, then n adopts the left sibling's back child. This type of imbalance can occur after performing a merge (Case 6 / Case 7, which we'll discuss later).

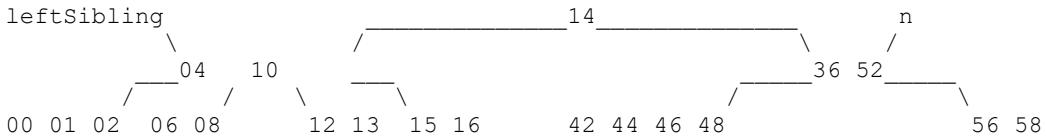


n is less than half full and n 's left sibling is more than half full, so perform a right rotation on n 's parent element (36);

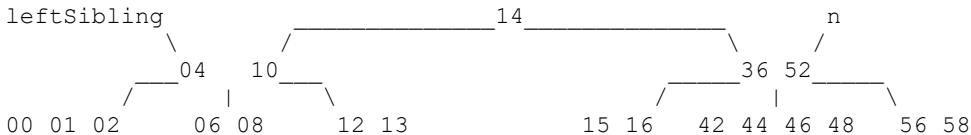
denote n 's parent element by moving it to the front of n ;



promote the spare element from the left sibling (move the left sibling's back element, 14, to the location formerly occupied by n 's parent element);



n 's left sibling is not a leaf node, so n adopts the left sibling's back child (move the left sibling's back child, node 15, to the front of n);

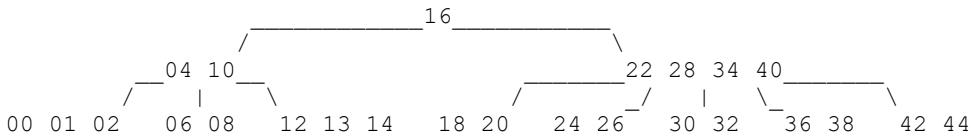


The function (*BTree.h*, line 122)

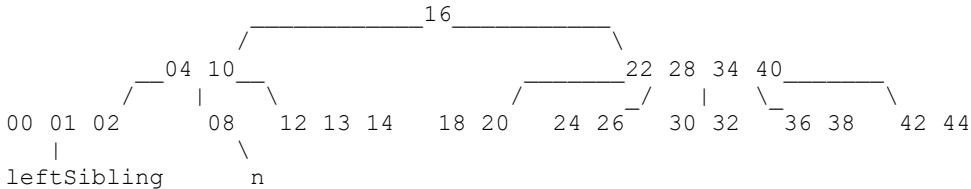
```
void _rotateParentElementRight(Node* n, Node* leftSibling);
```

performs a right rotation on n 's parent element, where n is the underflowing node with the given

leftSibling (which has a spare element) (*memberFunctions_3.h*, lines 121-140).



```
erase(6);
```

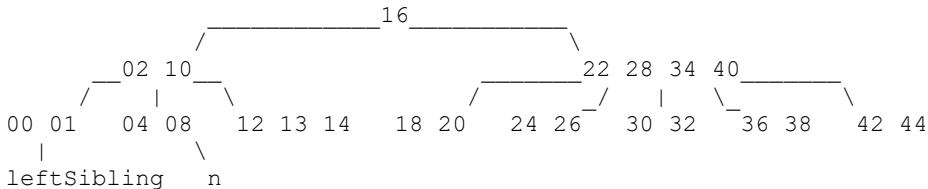


```
_rotateParentElementRight(n, leftSibling);
```

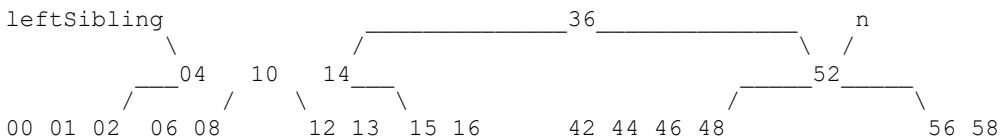
```
parentElementIndex = 0;      // the index of n's parent element (4)
parentElement = 4;
```

```
n->pushFrontElement(4);
n->parent()->overwriteElement(0, 2);
```

```
leftSibling->popBackElement();
```



```
// Imbalance at a non-leaf node (n), correctable via right rotation
```



```
_rotateParentElementRight(n, leftSibling);
```

```
parentElementIndex = 0;
parentElement = 36;
```

```
n->pushFrontElement(36);
n->parent()->overwriteElement(0, 14);
```

```
leftSibling->popBackElement();
```

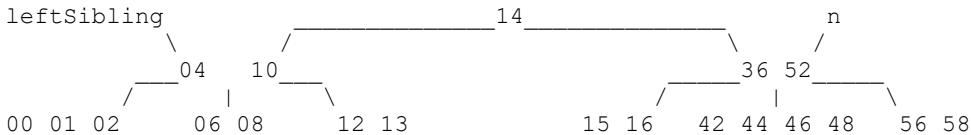
```

leftSibling           14          n
\                   /           \
04   10           12 13 15 16   42 44 46 48   36 52
/ \   / \         / \ / \ / \   / \ / \ / \
00 01 02 06 08   12 13   15 16   42 44   46 48   56 58

n->pushFrontChild(leftSibling->backChild());      // attach n to node 15
n->frontChild()->setParent(n);                      // attach node 15 to n

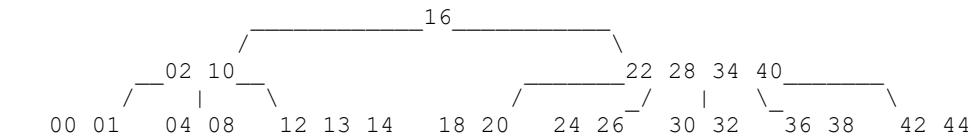
leftSibling->popBackChild();                         // detach node 15 from
                                                       // leftSibling

```



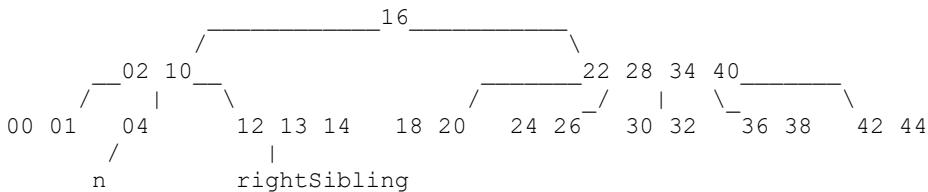
Case 4: n is less than half full, and n 's right sibling is more than half full

This is the mirror image of Case 3. If n needs an element and we can't obtain it via right rotation (because n doesn't have a left sibling, or n 's left sibling doesn't have a spare element), we look to n 's right sibling. If the right sibling has a spare element, we perform a left rotation. To perform a left rotation, we demote n 's parent element and promote the spare element from n 's right sibling.



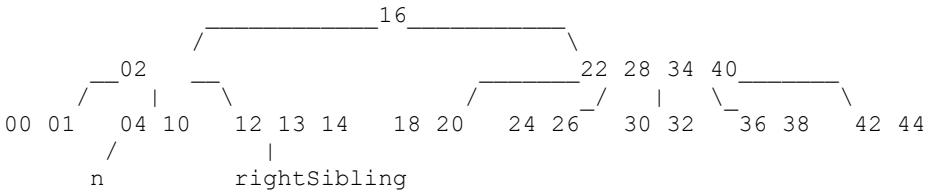
`erase(8);`

`element 8 is a leaf element,
so we can immediately remove it from node 4;`

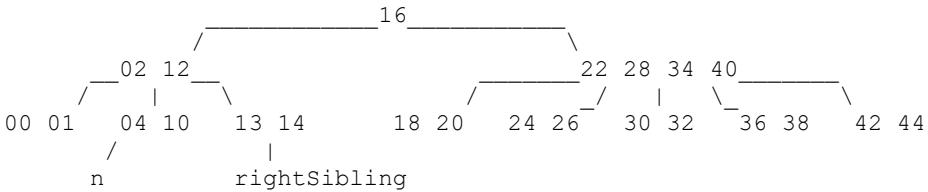


`n is less than half full and n's right sibling is more than half full,
so perform a left rotation on n's parent element (10);`

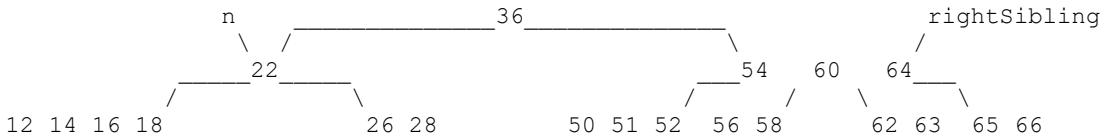
`demote n's parent element by moving it to the back of n;`



promote the spare element from the right sibling (move the right sibling's front element, 12, to the location formerly occupied by n's parent element);



If the right sibling is not a leaf node, then n adopts the right sibling's front child. This type of imbalance can occur after correcting an underflow via merging (Case 6 / Case 7, discussed later).

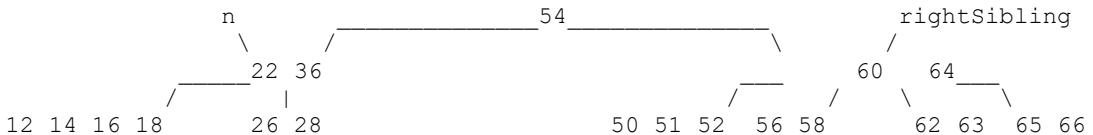


n is less than half full and n's right sibling is more than half full, so perform a left rotation on n's parent element (36);

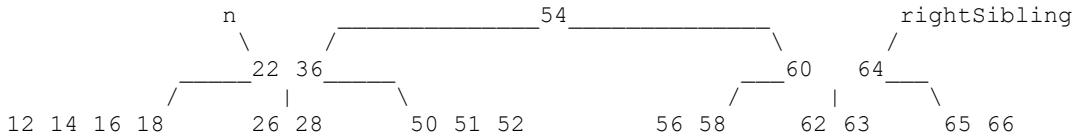
demote n's parent element by moving it to the back of n;



promote the spare element from the right sibling (move the right sibling's front element, 54, to the location formerly occupied by n's parent element);



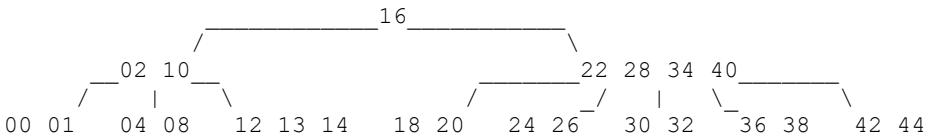
n's right sibling is not a leaf node, so n adopts the right sibling's front child (move the right sibling's front child, node 50, to the back of n);



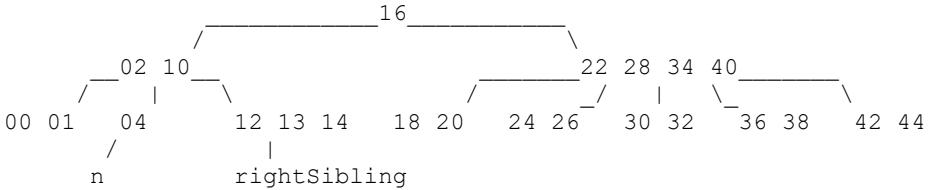
The function (*BTree.h*, line 121)

```
void _rotateParentElementLeft(Node* n, Node* rightSibling);
```

performs a left rotation on *n*'s parent element, where *n* is the underflowing node with the given *rightSibling* (which has a spare element) (*memberFunctions_3.h*, lines 142-162).



```
erase(8);
```

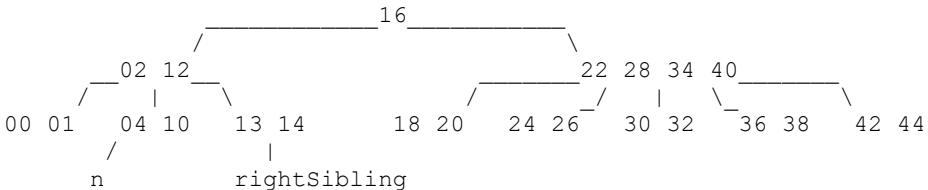


```
_rotateParentElementLeft(n, rightSibling);
```

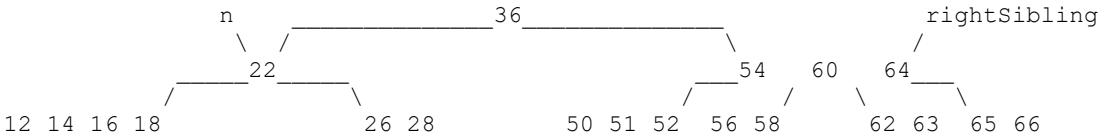
```
parentElementIndex = 1;
parentElement = 10;
```

```
n->pushBackElement(10);
n->parent()->overwriteElement(1, 12);
```

```
rightSibling->popFrontElement();
```



```
// Imbalance at a non-leaf node (n), correctable via left rotation
```

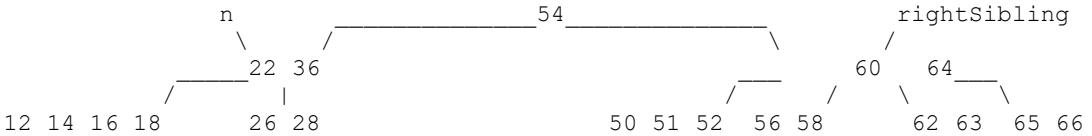


```
_rotateParentElementLeft(n, rightSibling);
```

```
parentElementIndex = 0;
parentElement = 36;
```

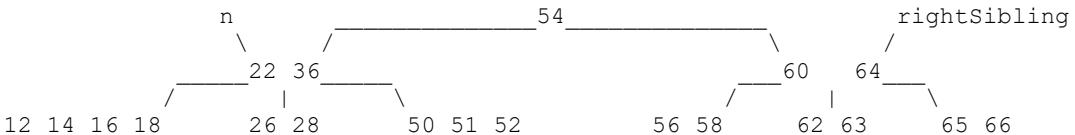
```
n->pushBackElement(36);
n->parent()->overwriteElement(0, 54);
```

```
rightSibling->popFrontElement();
```



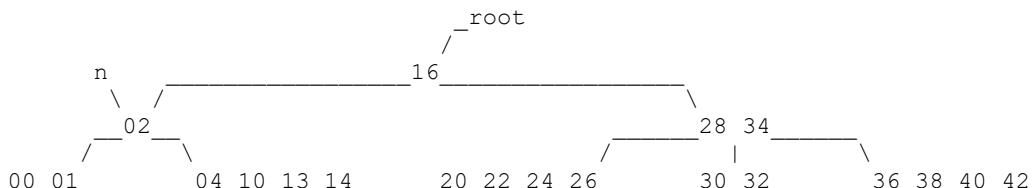
```
n->pushBackChild(rightSibling->frontChild()); // attach n to node 50
n->backChild()->setParent(n); // attach node 50 to n
```

```
rightSibling->popFrontChild(); // detach node 50 from
// rightSibling
```



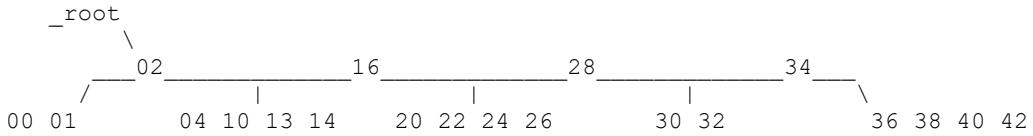
Case 5: n's parent is the root node, and the root contains exactly 1 element

If we can't correct the underflow via rotation (because neither of n's siblings have a spare element), we perform a merge operation, in which we combine nodes together. There are 3 types of merges, the first of which is Case 5. If n's parent is the root, and the root contains exactly 1 element, then the root has exactly 2 children (a left child and a right child):



To correct the underflow, we merge (combine) the root and its two children into a single node. The

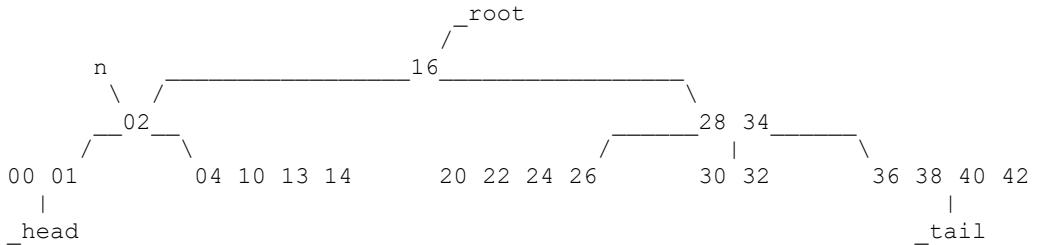
root thus becomes full and adopts all of its grandchildren:



The function (*BTree.h*, line 123)

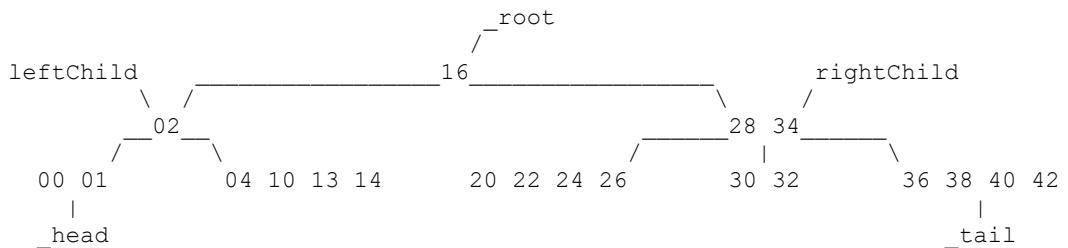
```
void _mergeRootWithItsTwoChildren();
```

merges the root node with its two children and updates the *_root*, *_head*, and *_tail* pointers accordingly. It doesn't matter whether the underflowing node is the root's left child or right child, the process remains the same (*memberFunctions_3.h*, lines 167-198):



```
_mergeRootWithItsTwoChildren();
```

```
leftChild = _root->frontChild();
rightChild = _root->backChild();
newRoot = _createNode();
```



```
// copy the leftChild's elements into the newRoot
```

```
for (Index i = 0; i != leftChild->totalElements(); ++i)
    newRoot->pushBackElement(leftChild->element(i));
```

```
newRoot
  \
  02
```

```
// attach the newRoot to the _root's left grandchildren

for (Index i = 0; i != leftChild->totalChildren(); ++i)
    newRoot->pushBackChild(leftChild->child(i));

newRoot
  \
  02 _____
 /   |   |
00 01 04 10 13 14
 |
head

// copy the _root's single element to the newRoot

newRoot->pushBackElement(_root->backElement());

newRoot
  \
  02 _____ 16
 /   |   |
00 01 04 10 13 14
 |
head

// copy the rightChild's elements into the newRoot

for (Index i = 0; i != rightChild->totalElements(); ++i)
    newRoot->pushBackElement(rightChild->element(i));

newRoot
  \
  02 _____ 16 _____ 28 _____ 34
 /   |   |   |
00 01 04 10 13 14
 |
head

// attach the newRoot to the _root's right grandchildren

for (Index i = 0; i != rightChild->totalChildren(); ++i)
    newRoot->pushBackChild(rightChild->child(i));

newRoot
  \
  02 _____ 16 _____ 28 _____ 34 _____
 /   |   |   |   |
00 01 04 10 13 14 20 22 24 26 30 32
 |
head
                                |
tail
```

```

// attach the _root's left and right grandchildren to the newRoot

for (Index i = 0; i != newRoot->totalChildren(); ++i)
    newRoot->child(i)->setParent(newRoot);

// the leftChild isn't the _head, so the _head is unchanged

if (leftChild == _head)
    _head = newRoot;

// the rightChild isn't the _tail, so the _tail is unchanged

if (rightChild == _tail)
    _tail = newRoot;

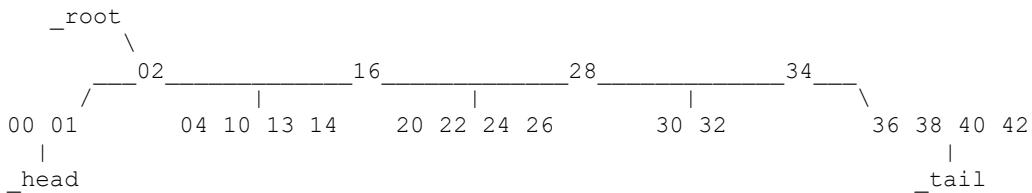
// destroy the original _root and its two children

_destroyNode(_root);
_destroyNode(leftChild);
_destroyNode(rightChild);

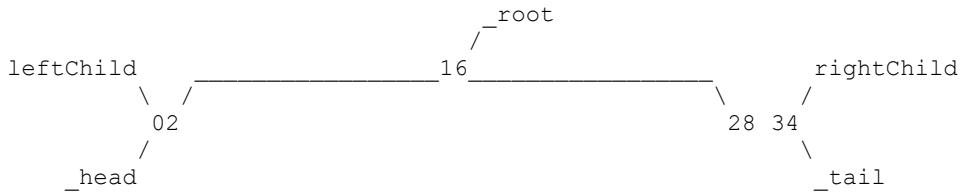
// update the _root pointer to the newRoot

_root = newRoot;

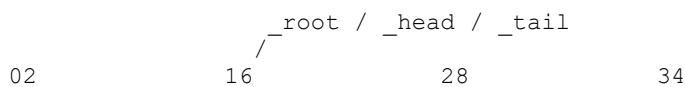
```



In this example, the `_head` and `_tail` were unchanged because they weren't the left and right children of the original `_root` (lines 188-192). If, however, the `_head` and `_tail` are the `_root`'s left and right children, as in

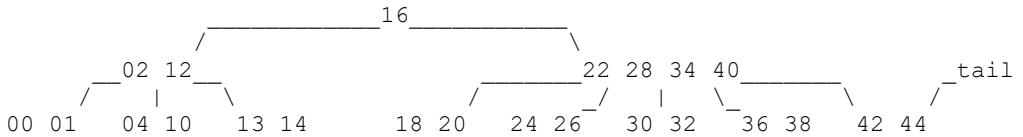


then after the merge, the `_head` and `_tail` become the new root:

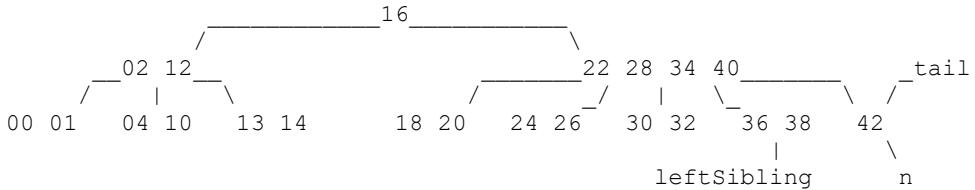


Case 6: n has a left sibling that is exactly half full

If we can't perform a rotation, and n isn't one the root's two only children, then we merge n with its left sibling. When the merge is complete, n 's left sibling becomes full, n is destroyed, and the left sibling may become the new tail:

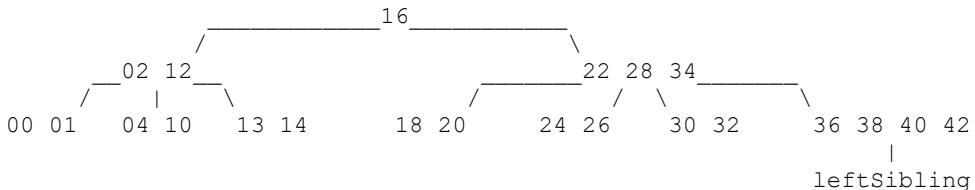


`erase(44);`

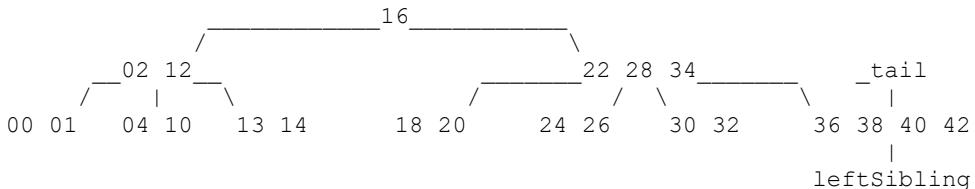


`merge n with its left sibling;`

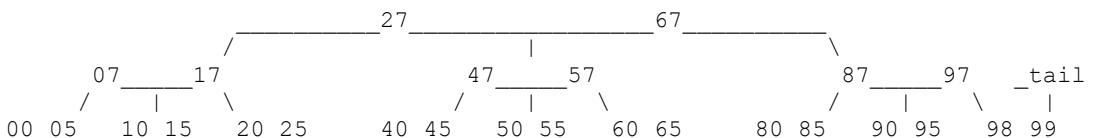
move n 's parent element (40) to the back of the left sibling;
move all of n 's elements (42) to the back of the left sibling;



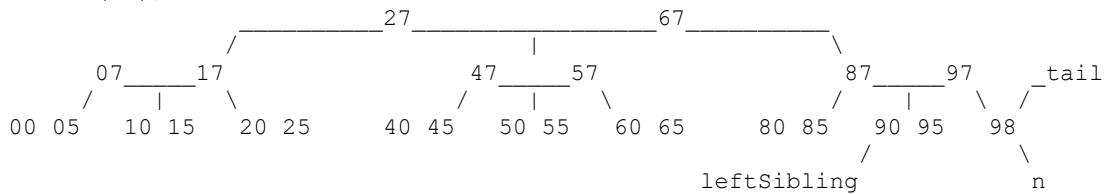
n was the tail, so update the tail to point to the left sibling;
detach n from n 's parent (node 22), and destroy n ;



If n isn't a leaf node, then n 's left sibling adopts all of n 's children. This type of imbalance can occur after merging n (Case 6 / Case 7) and proceeding to n 's parent:

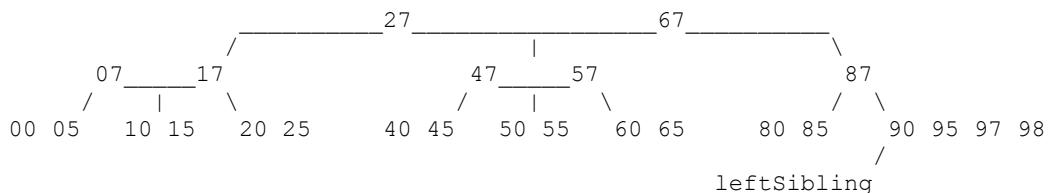


erase (99);

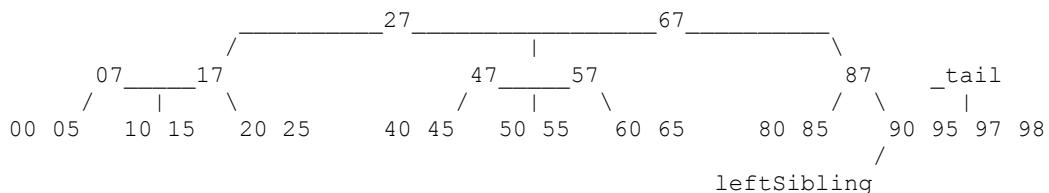


merge n with its left sibling;

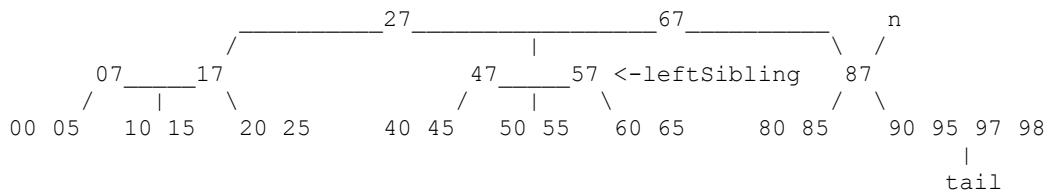
move n's parent element (97) to the back of the left sibling;
move all of n's elements (98) to the back of the left sibling;



n was the _tail, so update the _tail to point to the left sibling;
detach n from n's parent (node 97) and destroy n;

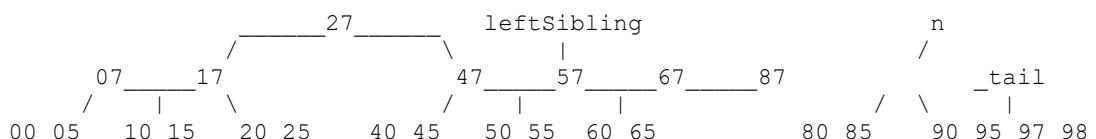


proceed to n's parent;



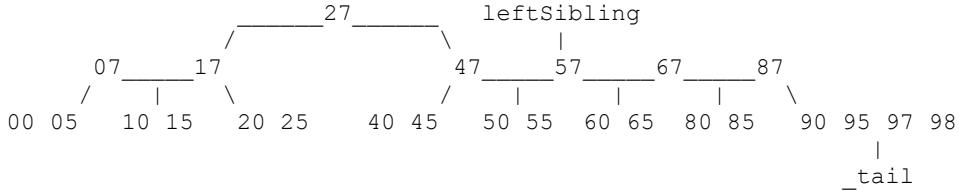
merge n with its left sibling;

move n's parent element (67) to the back of the left sibling;
move all of n's elements (87) to the back of the left sibling;



n isn't a leaf node, so the left sibling adopts all of n's children;
 n wasn't the `_tail`, so the `_tail` remains unchanged;

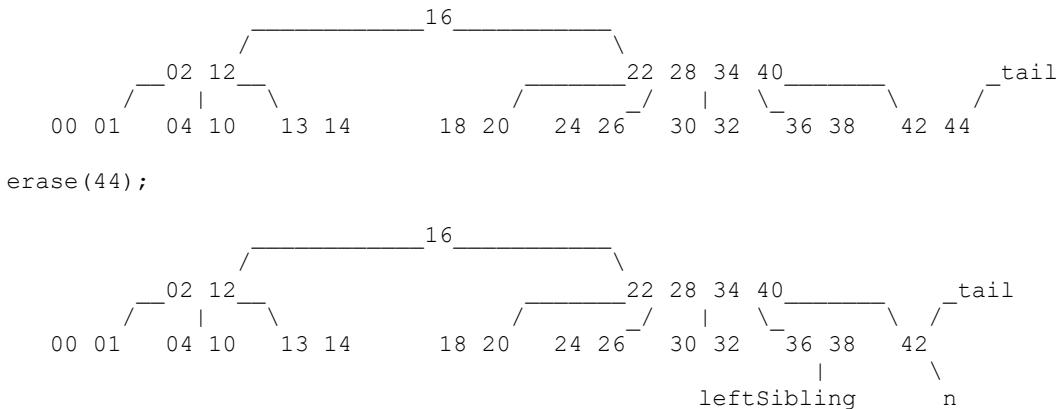
detach n from n's parent (node 27), and destroy n;



The function (*BTree.h*, line 124)

```
Node* _mergeWithLeftSibling(Node* n, Node* leftSibling);
```

merges *n* with its *leftSibling*, destroys *n*, and returns a pointer to the merged node (*memberFunctions_3.h*, lines 201-234):



```

erase(44);

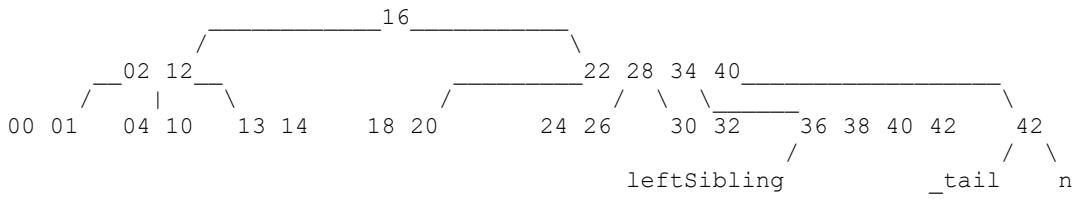
graph TD
    16[16] --- 02[02]
    16 --- 12[12]
    16 --- 22[22]
    16 --- 24[24]
    16 --- 26[26]
    16 --- 28[28]
    16 --- 30[30]
    16 --- 32[32]
    16 --- 34[34]
    16 --- 36[36]
    16 --- 38[38]
    16 --- 40[40]
    16 --- 42[42]
    16 --- 44[44]
    22[22] --- leftSibling[leftSibling]
    44[44] --- n[n]
    44 --> tail[_tail]

_mergeWithLeftSibling(n, leftSibling);

parentElementIndex = 3;
nIndex = 4; // n's index, relative to its parent (node 22)

parentElement = 40;
leftSibling->pushBackElement(40);

// copy all of n's elements to the back of the left sibling
for (Index i = 0; i != n->totalElements(); ++i)
    leftSibling->pushBackElement(n->element(i));
  
```



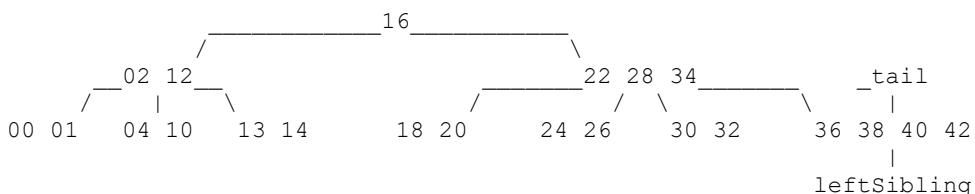
```
// n is a leaf node, so the left sibling doesn't adopt any
// children from n (lines 216-223)
```

```
// n is the _tail, so the _tail becomes the left sibling
```

```
if (n == _tail)
    _tail = leftSibling;
```

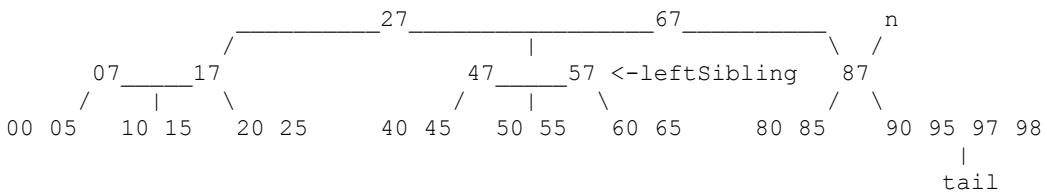
```
n->parent->eraseElement(3);      // erase 40 from node 22
n->parent->eraseChild(4);        // detach n from node 22
```

```
_destroyNode(n);
```



```
return leftSibling;           // return a pointer to the merged node
```

```
// Imbalance at non-leaf node n, correctable via merging
```



```
_mergeWithLeftSibling(n, leftSibling);
```

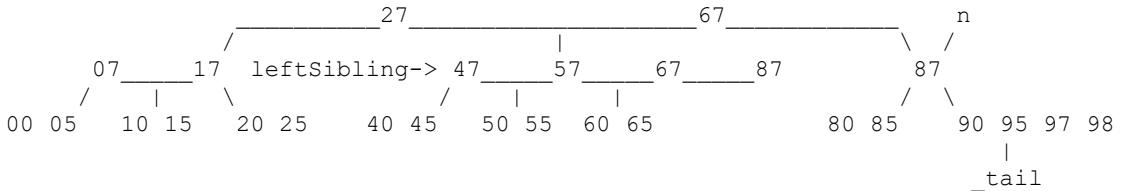
```
parentElementIndex = 1;
nIndex = 2;           // n's index, relative to its parent (node 27)
```

```
parentElement = 67;
```

```
leftSibling->pushBackElement(67);
```

```
// copy all of n's elements to the back of the left sibling
```

```
for (Index i = 0; i != n->totalElements(); ++i)
    leftSibling->pushBackElement(n->element(i));
```



```
// n is not a leaf node, so the left sibling adopts n's
// children (nodes 80 and 90)
```

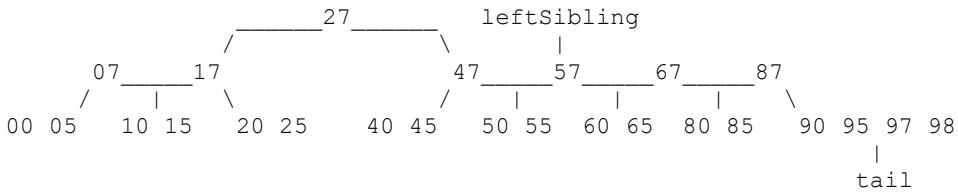
```
if (!n->isLeaf())
{
    for (Index i = 0; i != n->totalChildren(); ++i)
    {
        leftSibling->pushBackChild(n->child(i));
        leftSibling->backChild()->setParent(leftSibling);
    }
}
```

```
// n is not the _tail, so the _tail remains unchanged
```

```
if (n == _tail)
    _tail = leftSibling;
```

```
n->parent->eraseElement(1);      // erase 67 from node 27
n->parent->eraseChild(2);        // detach n from node 27
```

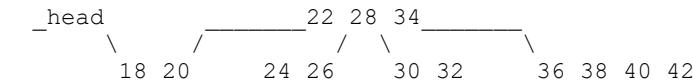
```
_destroyNode(n);
```



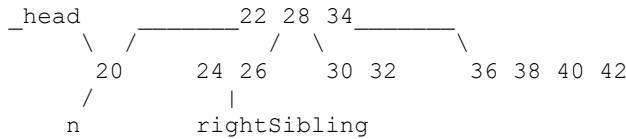
```
return leftSibling;                // return a pointer to the merged node
```

Case 7: n has a right sibling that is exactly half full

This is the mirror image of Case 6. If we can't merge n with its left sibling (because n doesn't have a left sibling), then we merge n with its right sibling. n 's right sibling becomes full, n is destroyed, and the right sibling may become the new $_head$:

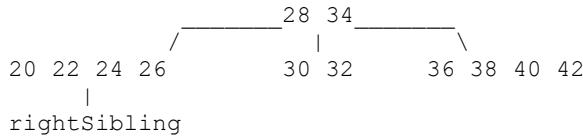


`erase(18);`

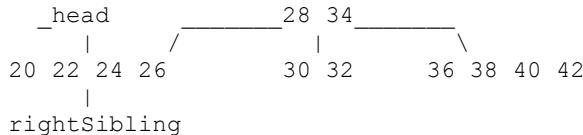


`merge n with its right sibling;`

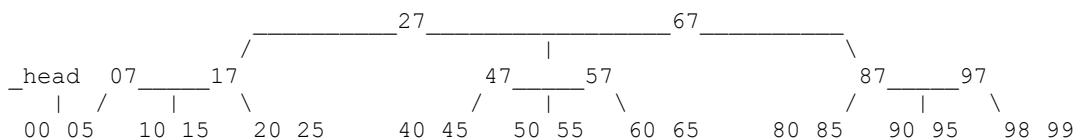
move n's parent element (22) to the front of the right sibling;
 move all of n's elements (20) to the front of the right sibling;



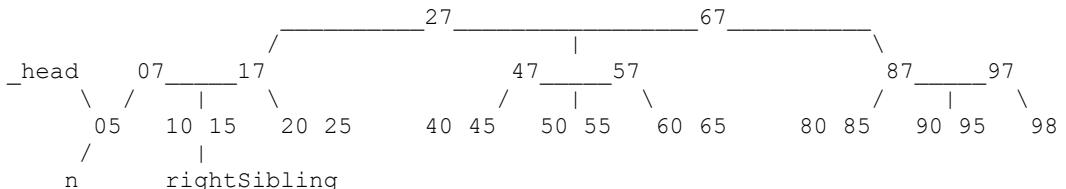
n was the `_head`, so update the `_head` to point to the right sibling;
 detach n from n's parent (node 28), and destroy n;



If n isn't a leaf node, then n's right sibling adopts all of n's children. This type of imbalance can occur after merging n (Case 6 / Case 7) and proceeding to n's parent:

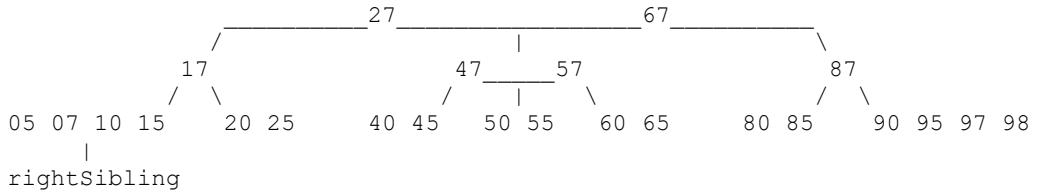


`erase(0);`

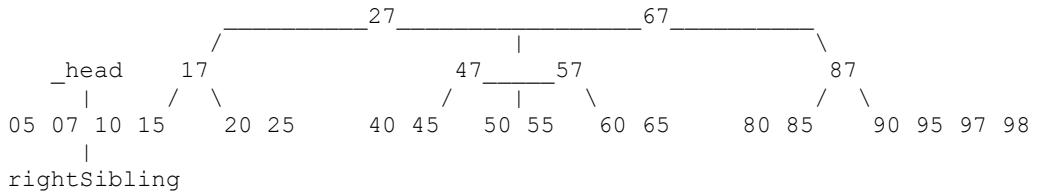


`merge n with its right sibling;`

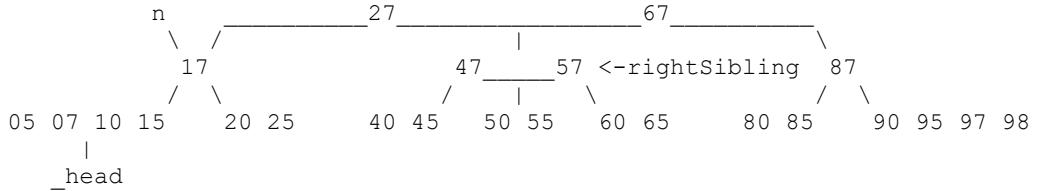
move n's parent element (7) to the front of the right sibling;
 move all of n's elements (5) to the front of the right sibling;



n was the head, so update the head to point to the right sibling;
 detach n from n's parent (node 17) and destroy n;

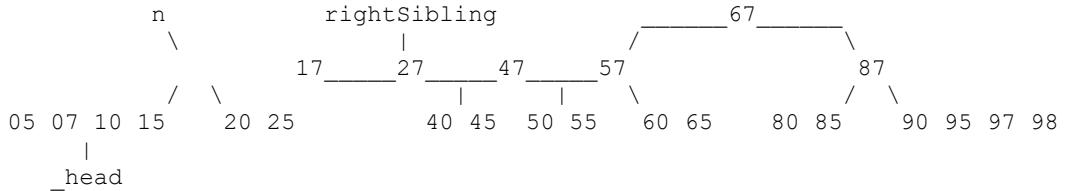


proceed to n's parent;



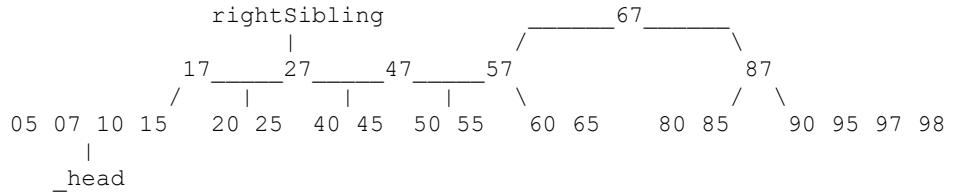
merge n with its right sibling;

move n's parent element (27) to the front of the right sibling;
 move all of n's elements (17) to the front of the right sibling;



n isn't a leaf node, so the right sibling adopts all of n's children;
 n wasn't the head, so the head remains unchanged;

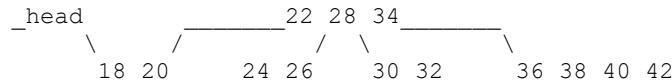
detach n from n's parent (node 67), and destroy n;



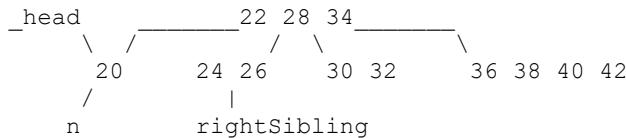
The function (*BTree.h*, line 125)

```
Node* _mergeWithRightSibling(Node* n, Node* rightSibling);
```

merges *n* with its *rightSibling*, destroys *n*, and returns a pointer to the merged node (*memberFunctions_3.h*, lines 236-274):



```
erase(18);
```



```
_mergeWithRightSibling(n, rightSibling);
```

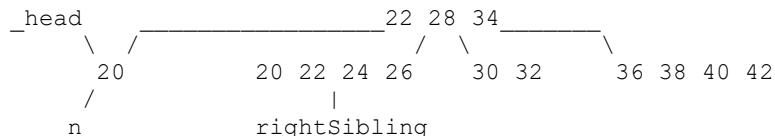
```
parentElementIndex = 0;
nIndex = 0; // n's index, relative to its parent (node 22)
```

```
parentElement = 22;
```

```
rightSibling->pushFrontElement(22);
```

```
// copy all of n's elements to the front of the right sibling
```

```
for (Index i = 0; i != n->totalElements(); ++i)
    rightSibling->pushFrontElement(n->element(i));
```



```
// n is a leaf node, so the right sibling doesn't adopt any
// children from n (lines 254-263)
```

```
// n is the _head, so the _head becomes the right sibling
```

```

if (n == _head)
    _head = rightSibling;

n->parent->eraseElement(0);      // erase 22 from node 22
n->parent->eraseChild(0);        // detach n from node 28

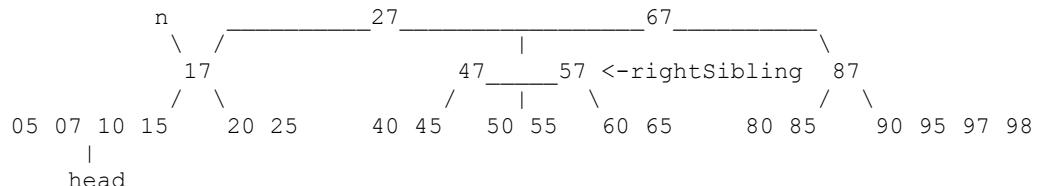
_destroyNode(n);

    _head           28 34
    |             /   \ 
  20 22 24 26     30 32     36 38 40 42
    |
rightSibling

return rightSibling;               // return a pointer to the merged node

```

// Imbalance at non-leaf node n, correctable via merging



```

_mergeWithRightSibling(n, rightSibling);

parentElementIndex = 0;
nIndex = 0;                      // n's index, relative to its parent (node 27)

parentElement = 27;

rightSibling->pushFrontElement(27);

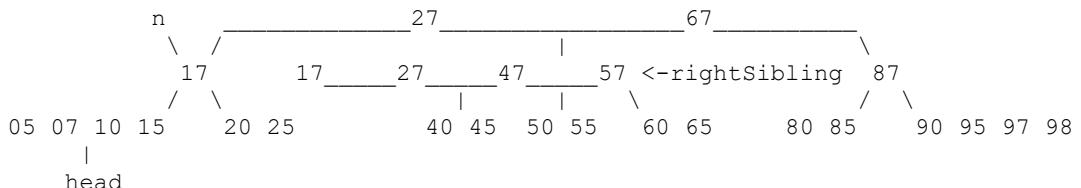
// copy all of n's elements to the front of the right sibling

```

```

for (Index i = 0; i != n->totalElements(); ++i)
    rightSibling->pushFrontElement(n->element(i));

```



```

// n is not a leaf node, so the right sibling adopts n's
// children (nodes 5 and 20)

```

```

if (!n->isLeaf())
{
    while (n->totalChildren() != 0)
    {
        rightSibling->pushFrontChild(n->backChild());
        rightSibling->frontChild()->setParent(rightSibling);

        n->popBackChild();
    }
}

// n is not the _head, so the _head remains unchanged

if (n == _head)
    _head = rightSibling;

n->parent->eraseElement(0);      // erase 27 from node 27
n->parent->eraseChild(0);        // detach n from node 67

_destroyNode(n);

                    rightSibling           67
                    |                           |
                   17   27   47   57           87
                   |   |   |   |           |
                  05  07  10  15   20  25  40  45   50  55   60  65
                   |           |
                   _head          80  85
                               |
                               90  95  97  98

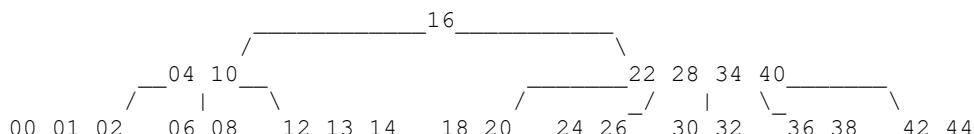
```

return rightSibling; // return a pointer to the merged node

Now that we can handle all 7 cases, we're ready to write `_balanceOnErase (memberFunctions_3.h, lines 82-119)`. The implementation closely resembles the pseudocode shown earlier:

- Case 1 (line 85)
- Case 2 (lines 87-88)
- Case 3 (lines 93-97)
- Case 4 (lines 98-102)
- Case 5 (lines 103-107)
- Case 6 (lines 108-112)
- Case 7 (lines 113-117)

Our test program (`main.cpp`) begins by constructing a `BTTree<Traceable<int>, int>` `b` (lines 14, 24-30),



In each iteration of the main loop (lines 36-50), we prompt the user to either erase an element or quit

the program. The function (line 16)

```
void getKeyAndEraseElement(_BTree& b);
```

begins by prompting the user for the key value of the element to be erased (lines 63-66), then searches for the element (line 68).

If the element was found (line 70) we erase it (line 74), obtaining an iterator to the next element in the sequence (the erased element's successor). We then print the successor (lines 78-81), along with the entire sequence and tree structure (lines 83-90).

If the element was not found (line 92), we inform the user that the desired key was not found (line 94).

The following sample run demonstrates the examples shown earlier:

```
// erase 6, _rotateParentElementRight (case 3)

Erasing element (6,0)...

Next element = (8,0)

Forward:
(0,0) (1,0) (2,0) (4,0) (8,0) (10,0) (12,0) (13,0) (14,0) (16,0) (18,0)
(20,0) (22,0) (24,0) (26,0) (28,0) (30,0) (32,0) (34,0) (36,0) (38,0) (40,0)
(42,0) (44,0)

Reverse:
(44,0) (42,0) (40,0) (38,0) (36,0) (34,0) (32,0) (30,0) (28,0) (26,0) (24,0)
(22,0) (20,0) (18,0) (16,0) (14,0) (13,0) (12,0) (10,0) (8,0) (4,0) (2,0)
(1,0) (0,0)

          16
         /   \
        02   10
       / | \
      00  01  04  08
      |   |
     12  13  14
      |   |
     18  20
      |   |
     24  26
      |   |
     30  32
      |   |
     36  38
      |   |
     42  44

key      0
parent->front 2

key      1
parent->front 2

key      2
parent->front 16
left      0
right     4

key      4
parent->front 2
```

key	8
parent->front	2
key	10
parent->front	16
left	4
right	12
key	12
parent->front	2
key	13
parent->front	2
key	14
parent->front	2
key	16
left	2
right	22
key	18
parent->front	22
key	20
parent->front	22
key	22
parent->front	16
left	18
right	24
key	24
parent->front	22
key	26
parent->front	22
key	28
parent->front	16
left	24
right	30
key	30
parent->front	22
key	32
parent->front	22
key	34
parent->front	16
left	30
right	36

```

key          36
parent->front 22

key          38
parent->front 22

key          40
parent->front 16
left         36
right        42

key          42
parent->front 22

key          44
parent->front 22

```

```
// erase 8, _rotateParentElementLeft (case 4)
```

Erasing element (8,0)...

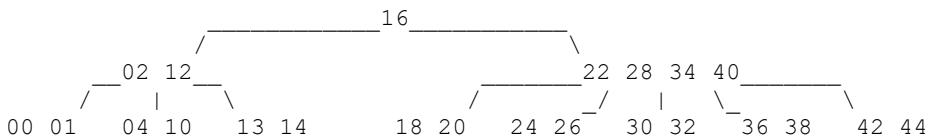
Next element = (10,0)

Forward:

```
(0,0) (1,0) (2,0) (4,0) (10,0) (12,0) (13,0) (14,0) (16,0) (18,0) (20,0)
(22,0) (24,0) (26,0) (28,0) (30,0) (32,0) (34,0) (36,0) (38,0) (40,0) (42,0)
(44,0)
```

Reverse:

```
(44,0) (42,0) (40,0) (38,0) (36,0) (34,0) (32,0) (30,0) (28,0) (26,0) (24,0)
(22,0) (20,0) (18,0) (16,0) (14,0) (13,0) (12,0) (10,0) (4,0) (2,0) (1,0)
(0,0)
```



```

key          0
parent->front 2

key          1
parent->front 2

key          2
parent->front 16
left         0
right        4

key          4
parent->front 2

```

```
key          10
parent->front 2

key          12
parent->front 16
left         4
right        13

key          13
parent->front 2

key          14
parent->front 2

key          16
left         2
right        22

key          18
parent->front 22

key          20
parent->front 22

key          22
parent->front 16
left         18
right        24

key          24
parent->front 22

key          26
parent->front 22

key          28
parent->front 16
left         24
right        30

key          30
parent->front 22

key          32
parent->front 22

key          34
parent->front 16
left         30
right        36

key          36
parent->front 22
```

```

key          38
parent->front 22

key          40
parent->front 16
left          36
right         42

key          42
parent->front 22

key          44
parent->front 22

```

```
// erase 44, _mergeWithLeftSibling (case 6)
```

```
Erasing element (44,0)...
```

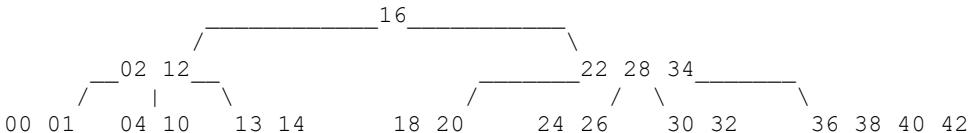
```
Next element = end
```

Forward:

```
(0,0) (1,0) (2,0) (4,0) (10,0) (12,0) (13,0) (14,0) (16,0) (18,0) (20,0)
(22,0) (24,0) (26,0) (28,0) (30,0) (32,0) (34,0) (36,0) (38,0) (40,0) (42,0)
```

Reverse:

```
(42,0) (40,0) (38,0) (36,0) (34,0) (32,0) (30,0) (28,0) (26,0) (24,0) (22,0)
(20,0) (18,0) (16,0) (14,0) (13,0) (12,0) (10,0) (4,0) (2,0) (1,0) (0,0)
```



```

key          0
parent->front 2

key          1
parent->front 2

key          2
parent->front 16
left          0
right         4

key          4
parent->front 2

key          10
parent->front 2

```

key	12
parent->front	16
left	4
right	13
key	13
parent->front	2
key	14
parent->front	2
key	16
left	2
right	22
key	18
parent->front	22
key	20
parent->front	22
key	22
parent->front	16
left	18
right	24
key	24
parent->front	22
key	26
parent->front	22
key	28
parent->front	16
left	24
right	30
key	30
parent->front	22
key	32
parent->front	22
key	34
parent->front	16
left	30
right	36
key	36
parent->front	22
key	38
parent->front	22

```

key          40
parent->front 22

key          42
parent->front 22


---


// erase 18, _mergeWithRightSibling (case 7)

Erasing element (18,0)...

Next element = (20,0)

Forward:
(0,0) (1,0) (2,0) (4,0) (10,0) (12,0) (13,0) (14,0) (16,0) (20,0) (22,0)
(24,0) (26,0) (28,0) (30,0) (32,0) (34,0) (36,0) (38,0) (40,0) (42,0)

Reverse:
(42,0) (40,0) (38,0) (36,0) (34,0) (32,0) (30,0) (28,0) (26,0) (24,0) (22,0)
(20,0) (16,0) (14,0) (13,0) (12,0) (10,0) (4,0) (2,0) (1,0) (0,0)




```

 16
 / \
 12 28
 / \ | \
 10 14 34 36
 / \ | \
 4 2 32 42
 / \ |
 1 0 30 38
 | |
 2 40
 |
 22

```



key          0
parent->front 2

key          1
parent->front 2

key          2
parent->front 16
left         0
right        4

key          4
parent->front 2

key          10
parent->front 2

key          12
parent->front 16
left         4
right        13

key          13
parent->front 2

key          14
parent->front 2

```

```

key          16
left         2
right        28

key          20
parent->front 28

key          22
parent->front 28

key          24
parent->front 28

key          26
parent->front 28

key          28
parent->front 16
left          20
right         30

key          30
parent->front 28

key          32
parent->front 28

key          34
parent->front 16
left          30
right         36

key          36
parent->front 28

key          38
parent->front 28

key          40
parent->front 28

key          42
parent->front 28

```

```

// erase 12, _mergeRootWithItsTwoChildren (case 5)

Erasing element (12,0)...

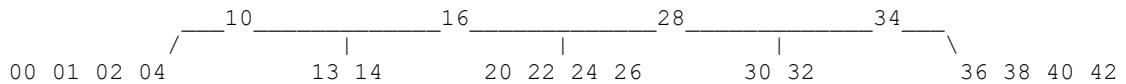
Next element = (13,0)

Forward:
(0,0) (1,0) (2,0) (4,0) (10,0) (13,0) (14,0) (16,0) (20,0) (22,0) (24,0)
(26,0) (28,0) (30,0) (32,0) (34,0) (36,0) (38,0) (40,0) (42,0)

```

Reverse:

(42,0) (40,0) (38,0) (36,0) (34,0) (32,0) (30,0) (28,0) (26,0) (24,0) (22,0)
 (20,0) (16,0) (14,0) (13,0) (10,0) (4,0) (2,0) (1,0) (0,0)



key 0
 parent->front 10

key 1
 parent->front 10

key 2
 parent->front 10

key 4
 parent->front 10

key 10
 left 0
 right 13

key 13
 parent->front 10

key 14
 parent->front 10

key 16
 left 13
 right 20

key 20
 parent->front 10

key 22
 parent->front 10

key 24
 parent->front 10

key 26
 parent->front 10

key 28
 left 20
 right 30

key 30
 parent->front 10

```
key          32
parent->front 10

key          34
left         30
right        36

key          36
parent->front 10

key          38
parent->front 10

key          40
parent->front 10

key          42
parent->front 10
```

```
// quit
```

```
~ 4
~ 2
~ 1
~ 0
~ 14
~ 13
~ 26
~ 24
~ 22
~ 20
~ 32
~ 30
~ 42
~ 40
~ 38
~ 36
~ 34
~ 28
~ 16
~ 10
```

```
All Traceables destroyed
```


6.9: Implementing Copy and Assignment

Source files and folders

- *BTree/5*
- *BTree/common/memberFunctions_4.h*

Chapter outline

- *Implementing BTree's copy constructor, assignment operator, and clear method*

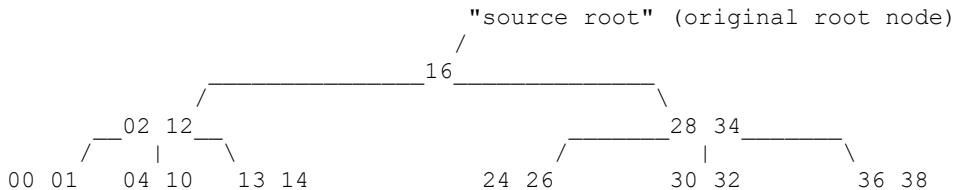
The function (*BTree.h*, line 129)

```
Node* _copyNode(Node* sourceNode, Node* newParent);
```

copies the tree rooted at the given *sourceNode* and attaches the copied tree to the *newParent*. If the *sourceNode* is not null, the function returns a pointer to the root of the new tree (the copy of the *sourceNode*); otherwise, it returns a null pointer. The procedure is

- Create a *newNode* (*memberFunctions_4.h*, line 61)
- Copy all of the *sourceNode*'s elements into the *newNode* (lines 63-64)
- Attach the *newNode* to its parent (the *newParent*) (line 66)
- For each child *i* in the *sourceNode* (line 68):
 - Recursively copy *i*, attaching the copy to its parent (the *newNode*) (line 70)
 - Insert the pointer to the copied child (returned by *_copyNode*) into the *newNode* (line 69)

Consider, for example, the tree



The following pseudocode demonstrates how *_copyNode* recursively copies the entire tree, beginning at the root. “Source root” is the original root node, “source node 2” is the original node 2, etc. “New root” is the copy of the original root, “new node 2” is the copy of the original node 2, etc.

```
_copyNode(source root, nullptr)
{
    create new root;
    copy source root's elements (16) to new root;
    new root->setParent(nullptr);

    _copyNode(source node 2, new root)
    {
        create new node 2;
        copy source node 2's elements (2, 12) to new node 2;
        new node 2->setParent(new root);

        _copyNode(soure node 0, new node 2)
        {
            create new node 0;
            copy source node 0's elements (0, 1) to new node 0;
            new node 0->setParent(new node 2);

            return new node 0;
        }
        new node 2->pushBackChild(new node 0);

        _copyNode(source node 4, new node 2)
        {
            create new node 4;
            copy source node 4's elements (4, 10) to new node 4;
            new node 4->setParent(new node 2);

            return new node 4;
        }
        new node 2->pushBackChild(new node 4);

        _copyNode(source node 13, new node 2)
        {
            create new node 13;
            copy source node 13's elements (13, 14) to new node 13;
            new node 13->setParent(new node 2);

            return new node 13;
        }
        new node 2->pushBackChild(new node 13);

        return new node 2;
    }
    new root->pushBackChild(new node 2);
}

// continued on next page
```

```

_copyNode(source node 28, new root)
{
    create new node 28;
    copy source node 28's elements (28, 34) to new node 28;
    new node 28->setParent(new root);

    _copyNode(source node 24, new node 28)
    {
        create new node 24;
        copy source node 24's elements (24, 26) to new node 24;
        new node 24->setParent(new node 28);

        return new node 24;
    }
    new node 28->pushBackChild(new node 24);

    _copyNode(source node 30, new node 28)
    {
        create new node 30;
        copy source node 30's elements (30, 32) to new node 30;
        new node 30->setParent(new node 28);

        return new node 30;
    }
    new node 28->pushBackChild(new node 30);

    _copyNode(source node 36, new node 28)
    {
        create new node 36;
        copy source node 36's elements (36, 38) to new node 36;
        new node 36->setParent(new node 28);

        return new node 36;
    }
    new node 28->pushBackChild(new node 36);

    return new node 28;
}
new root->pushBackChild(new node 28);

return new root;
}

```

To implement the copy constructor (*BTree.h*, line 59), we initialize the *_root*, *_head*, and *_tail* to the pointer returned by *_copyNode* (*memberFunctions_4.h*, lines 10-12), and set the *_size* to that of the *source* tree. We then set the *_head* to the leftmost leaf (lines 17-18), and the *_tail* to the rightmost leaf (lines 20-21).

To implement *clear* (*BTree.h*, line 86), we simply call *_destroyTree* then reset the *_root*, *_head*, *_tail*, and *_size* (*memberFunctions_4.h*, lines 46-52).

The assignment operator (*BTree.h*, line 82) uses the same technique as the *BinaryTree* and *AvlTree*

classes from Volume 1 (*memberFunctions_4.h*, lines 25-41). We *clear* the left-hand tree, create a *temp* copy of the right-hand tree (via the copy constructor), then “steal” *temp*’s elements by swapping its *_root*, *_head*, *_tail*, and *_size* with those of the left-hand tree.

Our test program (*main.cpp*) constructs a *BTree<Traceable<int>, int>* *x*, containing the keys [0, 44] in increments of 2 (lines 18-23). We then use the copy constructor to create another tree *y* from *x* (line 25).

To test the assignment operator, we create another tree *z* containing the keys [0, 4] (lines 27-30). We then overwrite the contents of *z* by assigning the contents of *y* (lines 32), after which *z* contains the keys [0, 44] (in increments of 2).

Printing the contents of *x*, *y*, and *z* (lines 34-50) shows that they are identical, generating the output

Forward:

```
(0,0) (2,0) (4,0) (6,0) (8,0) (10,0) (12,0) (14,0) (16,0) (18,0) (20,0) (22,0)
(24,0) (26,0) (28,0) (30,0) (32,0) (34,0) (36,0) (38,0) (40,0) (42,0) (44,0)

(0,0) (2,0) (4,0) (6,0) (8,0) (10,0) (12,0) (14,0) (16,0) (18,0) (20,0) (22,0)
(24,0) (26,0) (28,0) (30,0) (32,0) (34,0) (36,0) (38,0) (40,0) (42,0) (44,0)

(0,0) (2,0) (4,0) (6,0) (8,0) (10,0) (12,0) (14,0) (16,0) (18,0) (20,0) (22,0)
(24,0) (26,0) (28,0) (30,0) (32,0) (34,0) (36,0) (38,0) (40,0) (42,0) (44,0)
```

Reverse:

```
(44,0) (42,0) (40,0) (38,0) (36,0) (34,0) (32,0) (30,0) (28,0) (26,0) (24,0)
(22,0) (20,0) (18,0) (16,0) (14,0) (12,0) (10,0) (8,0) (6,0) (4,0) (2,0) (0,0)

(44,0) (42,0) (40,0) (38,0) (36,0) (34,0) (32,0) (30,0) (28,0) (26,0) (24,0)
(22,0) (20,0) (18,0) (16,0) (14,0) (12,0) (10,0) (8,0) (6,0) (4,0) (2,0) (0,0)

(44,0) (42,0) (40,0) (38,0) (36,0) (34,0) (32,0) (30,0) (28,0) (26,0) (24,0)
(22,0) (20,0) (18,0) (16,0) (14,0) (12,0) (10,0) (8,0) (6,0) (4,0) (2,0) (0,0)
```

At the end of *main*, all 3 trees are destroyed, generating the output

```
~ 2      // destroy z
~ 0
~ 8
~ 6
~ 14
~ 12
~ 10
~ 4
~ 20
~ 18
~ 26
~ 24
~ 32
```

```
~ 30
~ 38
~ 36
~ 44
~ 42
~ 40
~ 34
~ 28
~ 22
~ 16
~ 2      // destroy y
~ 0
~ 8
~ 6
~ 14
~ 12
~ 10
~ 4
~ 20
~ 18
~ 26
~ 24
~ 32
~ 30
~ 38
~ 36
~ 44
~ 42
~ 40
~ 34
~ 28
~ 22
~ 16
~ 2      // destroy x
~ 0
~ 8
~ 6
~ 14
~ 12
~ 10
~ 4
~ 20
~ 18
~ 26
~ 24
~ 32
~ 32
~ 30
~ 38
~ 36
~ 44
~ 42
~ 40
~ 34
~ 28
```

204

~ 22
~ 16

All Traceables destroyed

Part 7: Red-Black Trees

7.1: Introducing the *RedBlackTree* Class

Source files and folders

- *PrintRedBlackNode*
- *RedBlackTree/l*
- *RedBlackTree/common/memberFunctions_1.h*
- *RedBlackTree/common/memberFunctions_2.h*
- *RedBlackTree/common/RedBlackTreeNode.h*

Chapter outline

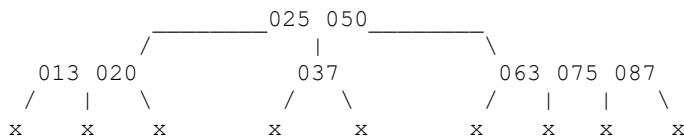
- *Properties and terminology*
- *Creating the node class*
- *Creating the RedBlackTree class*
- *Printing the structure of a red-black tree*

In Volume 1 we studied AVL trees, introducing the concept of binary tree node rotation. Now that we have an understanding of B-trees, we can combine these two concepts to implement red-black trees, another common type of balanced binary search tree.

The *red-black tree*, invented by Leonidas Guibas and Robert Sedgewick, is an order-4 B-tree represented as a binary search tree. The specification for this B-tree, however, is a bit less strict: each node is only required to be a third (as opposed to half) full. According to this definition, the tree consists of 3 types of nodes:

- *2-node*: A node that has 2 children (1 element)
- *3-node*: A node that has 3 children (2 elements)
- *4-node*: A node that has 4 children (3 elements)

This type of B-tree is therefore also known as a *2-3-4 tree*. Consider, for example, the tree



where *x* represents a null child node:

- The root, node {25, 50}, is a 3-node

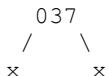
- Node {13, 20} is a 3-node
- Node {37} is a 2-node
- Node {63, 75, 87} is a 4-node

Using the standard implementation of B-tree nodes,

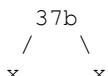
- A 2-node contains an array of 2 child pointers and an array of 1 element
- A 3-node contains an array of 3 child pointers and an array of 2 elements
- A 4-node contains an array of 4 child pointers and an array of 3 elements

To represent 2-nodes, 3-nodes, and 4-nodes using traditional binary tree nodes (each consisting of a single element and left / right child pointers), each node is given a color, red or black:

- A 2-node is represented by a single black binary tree node. The B-tree 2-node



for example, becomes

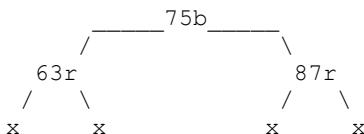


where *b* indicates the color black.

- A 4-node is represented by 3 binary tree nodes: 1 black node containing the middle element, and 2 red nodes containing the left and right elements. The B-tree 4-node



for example, becomes



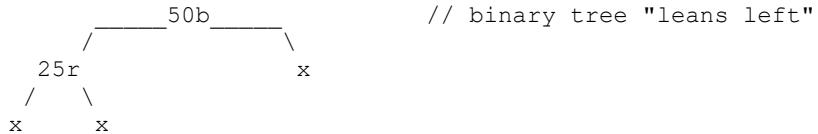
where *r* indicates the color red.

- A 3-node, represented by 2 binary tree nodes, can occur in 2 possible configurations:
 - A *left-leaning 3-node* consists of a black node containing the right element, and a red node

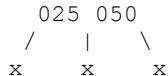
containing the left element. The B-tree 3-node



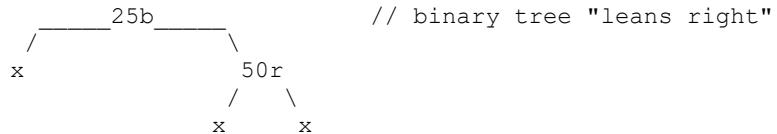
for example, becomes



- A *right-leaning 3-node* consists of a black node containing the left element, and a red node containing the right element. The B-tree 3-node



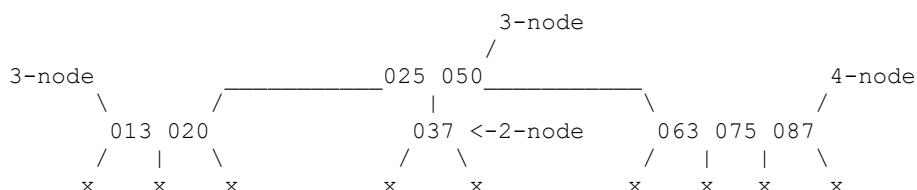
for example, becomes



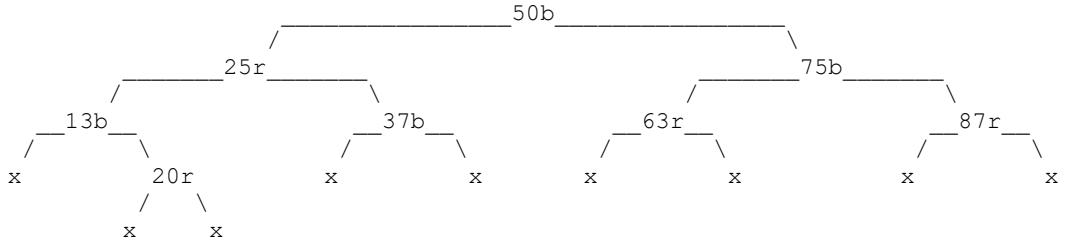
A red-black tree, which consists entirely of 2-nodes, 3-nodes, and 4-nodes, has the following properties:

- Each non-null node has a color, either red or black
- All null child nodes are considered to be black
- The root node is black
- Each red node has 2 black children (a red node cannot have a red child)
- From any given node n , all paths from n to the bottom of the tree contain the same number of black nodes (the number of black nodes in a given path is called the *black height* of that path)

Given the B-tree



for example, one possible red-black representation is



where $\{25, 50\}$ is a left-leaning 3-node and $\{13, 20\}$ is a right-leaning 3-node. We can verify that it satisfies all the requirements of a red-black tree:

- The root node (50) is black
- Each red node (20, 25, 63, 87) has 2 black children
- From any given node n , all paths from n to the bottom of the tree contain the same number of black nodes (all paths have an equal black height):
 - From node 13, all 3 paths to the bottom of the tree contain 2 black nodes:
 - Path 1 = 13 (black), null left child (black)
 - Path 2 = 13 (black), 20 (red), null left child (black)
 - Path 3 = 13 (black), 20 (red), null right child (black)
 - From node 20, all 2 paths to the bottom of the tree contain 1 black node:
 - Path 1 = 20 (red), null left child (black)
 - Path 2 = 20 (red), null right child (black)
 - From node 25, all 5 paths to the bottom of the tree contain 2 black nodes:
 - Path 1 = 25 (red), 13 (black), null left child (black)
 - Path 2 = 25 (red), 13 (black), 20 (red), null left child (black)
 - Path 3 = 25 (red), 13 (black), 20 (red), null right child (black)
 - Path 4 = 25 (red), 37 (black), null left child (black)
 - Path 5 = 25 (red), 37 (black), null right child (black)
 - From node 37, all 2 paths to the bottom of the tree contain 2 black nodes:
 - Path 1 = 37 (black), null left child (black)
 - Path 2 = 37 (black), null right child (black)
 - From node 50, all 9 paths to the bottom of the tree contain 3 black nodes:
 - Path 1 = 50 (black), 25 (red), 13 (black), null left child (black)
 - Path 2 = 50 (black), 25 (red), 13 (black), 20 (red), null left child (black)

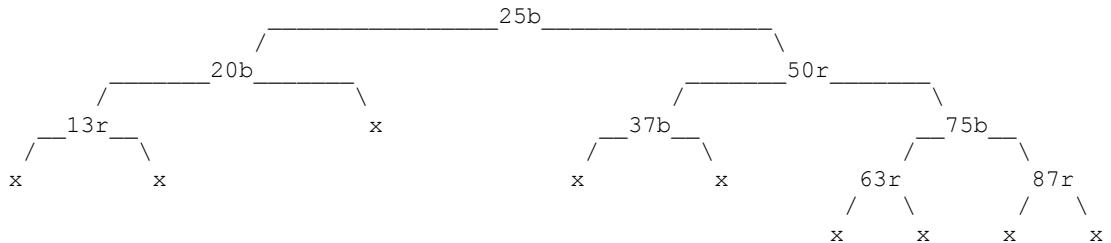
- Path 3 = 50 (black), 25 (red), 13 (black), 20 (red), null right child (black)
- Path 4 = 50 (black), 25 (red), 37 (black), null left child (black)
- Path 5 = 50 (black), 25 (red), 37 (black), null right child (black)
- Path 6 = 50 (black), 75 (black), 63 (red), null left child (black)
- Path 7 = 50 (black), 75 (black), 63 (red), null right child (black)
- Path 8 = 50 (black), 75 (black), 87 (red), null left child (black)
- Path 9 = 50 (black), 75 (black), 87 (red), null right child (black)

- From node 63, all 2 paths to the bottom of the tree contain 1 black node:
 - Path 1 = 63 (red), null left child (black)
 - Path 2 = 63 (red), null right child (black)

- From node 75, all 4 paths to the bottom of the tree contain 2 black nodes:
 - Path 1 = 75 (black), 63 (red), null left child (black)
 - Path 2 = 75 (black), 63 (red), null right child (black)
 - Path 3 = 75 (black), 87 (red), null left child (black)
 - Path 4 = 75 (black), 87 (red), null right child (black)

- From node 87, all 2 paths to the bottom of the tree contain 1 black node:
 - Path 1 = 87 (red), null left child (black)
 - Path 2 = 87 (red), null right child (black)

Alternatively, we could create another (equally valid) red-black tree by making {25, 50} a right-leaning 3-node and {13, 20} a left-leaning 3-node:



Regardless of whether the 3-nodes are depicted as left or right-leaning, however, we can always “reassemble” the underlying B-tree from its red-black representation, using the following rules:

- A black node with 2 black children is a 2-node
- A black node with 1 red child is a 3-node
- A black node with 2 red children is a 4-node

Each red element, in other words, can be thought of as belonging to the same underlying B-tree node as its black parent element. Additionally, no 2 black elements can ever belong to the same B-tree node. In the above tree, for example,

- 13 (red) belongs to the same B-tree node as 20 (black parent), forming the 3-node

13 20

- 37 (black) has 2 black children, forming the 2-node

37

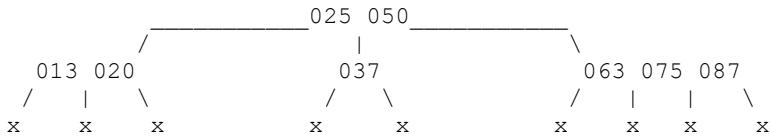
- 63 (red) and 87 (red) belong to the same B-tree node as 75 (black parent), forming the 4-node

63 75 87

- 50 (red) belongs to the same B-tree node as 25 (black parent), forming the 3-node

25 50

Beginning at the root, {25, 50} is a 3-node, so it has 3 children. Its left child must be {13, 20}, its middle child {37}, and its right child {63, 75, 87}:



Visualizing red-black trees as B-trees will be particularly helpful when we implement the insertion and erasure algorithms in the next two chapters. But for now, let's begin by creating the node class (*RedBlackTreeNode.h*). Its data members (lines 26-36) are

- *element*, the contained key-mapped pair
- *parent*, a pointer to the node's parent
- *left*, a pointer to the node's left child
- *right*, a pointer to the node's right child
- *predecessor*, a pointer to the node's in-order predecessor
- *successor*, a pointer to the node's in-order successor
- *_isRed*, a boolean flag indicating whether the node is red (*true*) or black (*false*)

The constructor (line 17) creates a red node from the given *sourceElement*, initializing all of its links to null (lines 39-51).

The member functions *isRed* and *isBlack* (lines 19-20) tell us whether the node is red or black (lines 53-63).

paintRed and *paintBlack* (lines 22-23) set the node's color to red and black, respectively (lines 65-75).

matchColor (line 24) sets the node's color to that of the given node *n* (lines 77-81).

The framework for the *RedBlackTree* class (*RedBlackTree.h*) is nearly identical to that of the *BinaryTree* class that we created in Volume 1. Starting with the iterators,

- *iterator* (line 32) is an alias of the type *dss::BinaryTreeIter<RedBlackTree>*
- *reverse_iterator* (line 33) is an alias of the type *dss::ReverseIter<iterator>*
- *const_iterator* (line 28) is an alias of the type *dss::ConstIter<RedBlackTree>*
- *const_reverse_iterator* (line 29) is an alias of the type *dss::ReverseIter<const_iterator>*

Node (line 36) is an alias of the type *RedBlackTreeNode<key_type, mapped_type>*.

The data members (lines 73-78) are:

- *_root*, a pointer to the root node
- *_head*, a pointer to the node containing the *front* element
- *_tail*, a pointer to the node containing the *back* element
- *_size*, the total number of elements
- *_predicate*, the function object used to compare key values
- *_alloc*, an *allocator* used to create and destroy nodes

The only member functions that we need to modify from the original (*BinaryTree*) versions are *_copyNode* (line 67) and *_overwriteElement* (line 71), both of which only require one additional line:

- In *_copyNode*, we need the copied node (*newNode*) to be the same color as the original node (*sourceNode*), achieved by simply calling *matchColor* (*memberFunctions_2.h*, line 16).
- In *_overwriteElement*, we need the given node *n* to retain its original color, achieved by saving *n*'s color to *temp* (line 38).

The remaining member functions (*RedBlackTree.h*, lines 38-64, 68-70) are defined in the header file *memberFunctions_1.h*. The only updates that we need to make here are *using* declarations (lines 22-23) and namespace identification (line 204), since we're now outside namespace *dss*.

We can also reuse our testing tools from Volume 1, *traverseInOrder* and *PrintBtNode*. Our new function object, *PrintRedBlackNode* (*PrintRedBlackNode.h*), contains a *PrintBtNode* (line 17), which prints the current node's key value and those of its parent and children (line 25), after which we print the color (lines 27-30). To print the entire structure of a red-black tree, we simply call *traverseInOrder*, passing a pointer to the root along with a *PrintRedBlackNode* function object.

7.2: Inserting Elements

Source files and folders

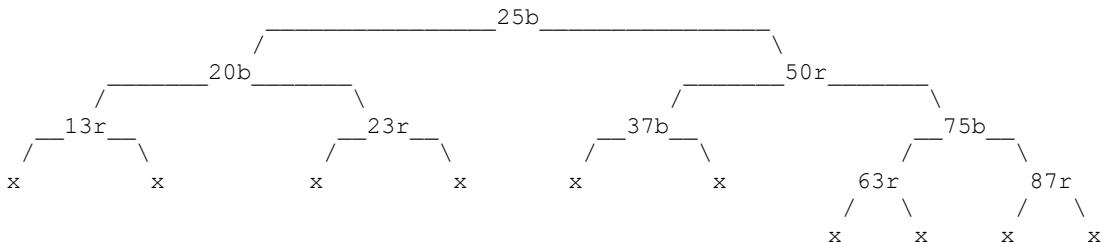
- *bt*
- *RedBlackTree/2*
- *RedBlackTree/common/memberFunctions_3.h*

Chapter outline

- *Creating additional functions for finding a node's relatives*
- *Implementing RedBlackTree::insert*

Before implementing *RedBlackTree*'s *insert* method, let's write a few stand-alone functions for accessing a node's more distant relatives (*relatives.h*). We'll declare them in namespace *ds2::bt*, similar to how we declared the Volume 1 set of *relatives* functions in namespace *dss::bt*.

The first new function, *hasGrandparent* (lines 10-11), returns *true* if the given node *n* has a non-null grandparent (lines 25-31). *hasGrandparent* will be *true* as long as *n* isn't null, the root, or a direct child of the root. In the tree



for example,

- *hasGrandparent* returns *true* for nodes 13, 23, 37, 63, 75, and 87
- *hasGrandparent* returns *false* for nodes 20, 25, and 50

hasUncle (lines 13-14) returns *true* if *n* has a non-null uncle (the *uncle* of a node *n* is the sibling of *n*'s parent):

- If *n* doesn't have a grandparent, then *n* can't possibly have an uncle (lines 38-39).
- If *n*'s parent is the left child of *n*'s grandparent, we check whether *n*'s grandparent has a right child; this right child is *n*'s uncle (the sibling of *n*'s parent) (lines 40-41).
- If *n*'s parent is the right child of *n*'s grandparent, we check whether *n*'s grandparent has a left child; this left child is *n*'s uncle (the sibling of *n*'s parent) (lines 42-43).

In the above tree,

- *hasUncle(node 13)* is *true* (node 13's uncle is node 50)
- *hasUncle(node 20)* is *false*
- *hasUncle(node 23)* is *true* (node 23's uncle is node 50)
- *hasUncle(node 25)* is *false*
- *hasUncle(node 37)* is *true* (node 37's uncle is node 20)
- *hasUncle(node 50)* is *false*
- *hasUncle(node 63)* is *true* (node 63's uncle is node 37)
- *hasUncle(node 75)* is *true* (node 75's uncle is node 20)
- *hasUncle(node 87)* is *true* (node 87's uncle is node 37)

grandparent (lines 16-17) returns a pointer to *n*'s grandparent (the parent of *n*'s parent) (lines 46-50). If *n* is the root, the function is undefined, and if *n* is a direct child of the root, the function returns a null pointer. In the above tree,

- *grandparent(node 13)* returns a pointer to node 25
- *grandparent(node 20)* returns a null pointer
- *grandparent(node 23)* returns a pointer to node 25
- *grandparent(node 25)* is undefined
- *grandparent(node 37)* returns a pointer to node 25
- *grandparent(node 50)* returns a null pointer
- *grandparent(node 63)* returns a pointer to node 50
- *grandparent(node 75)* returns a pointer to node 25
- *grandparent(node 87)* returns a pointer to node 50

uncle (lines 19-20) returns a pointer to *n*'s uncle (lines 52-61). Because it directly accesses *n*'s grandparent without performing any null checks, this function is undefined if *n* is either the root or a direct child of the root. We'll therefore need to verify that *n* has an uncle (via *hasUncle*) before calling this function. In the above tree,

- *uncle(node 13)* returns a pointer to node 50
- *uncle(node 20)* is undefined
- *uncle(node 23)* returns a pointer to node 50
- *uncle(node 25)* is undefined
- *uncle(node 37)* returns a pointer to node 20
- *uncle(node 50)* is undefined
- *uncle(node 63)* returns a pointer to node 37
- *uncle(node 75)* returns a pointer to node 20
- *uncle(node 87)* returns a pointer to node 37

sibling (lines 22-23) returns a pointer to *n*'s sibling (the other child of *n*'s parent). If *n* is the root (which doesn't have a sibling), the function returns a null pointer (lines 63-74). In the above tree,

- *sibling*(node 13) returns a pointer to node 23
- *sibling*(node 20) returns a pointer to node 50
- *sibling*(node 23) returns a pointer to node 13
- *sibling*(node 25) returns a null pointer
- *sibling*(node 37) returns a pointer to node 75
- *sibling*(node 50) returns a pointer to node 20
- *sibling*(node 63) returns a pointer to node 87
- *sibling*(node 75) returns a pointer to node 37
- *sibling*(node 87) returns a pointer to node 63

We're now ready to implement *RedBlackTree*'s *insert* method (*RedBlackTree.h*, line 64),

```
std::pair<iterator, bool> insert(const value_type& newElement);
```

- If the *newElement* was inserted, the returned *pair* will contain an iterator to the newly inserted element and *true*
- If the tree already contains an element with the same key value as the *newElement*, the returned *pair* will contain an iterator to the preexisting element and *false*

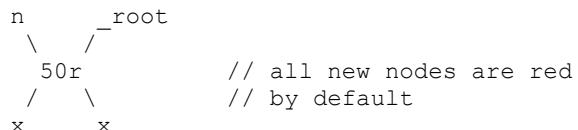
The insertion procedure (*memberFunctions_3.h*, lines 16-53, 56) is nearly identical to the original (*BinaryTree*) version. The only differences are that the *newNode* now has a color (red, set by the constructor), and after inserting the new node, we call the private member function (*RedBlackTree.h*, line 73)

```
void _balanceOnInsert(Node* n);
```

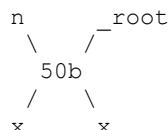
to balance the tree (*memberFunctions_3.h*, line 54). Before implementing *_balanceOnInsert*, let's take a look at the different types of violations that can occur and write functions to correct them.

Case 1: *n* is the root

If the newly inserted node *n* is the root of the tree, then we simply paint it black, thus making the root a 2-node; no further balancing is necessary. Suppose, for example, that we insert the key value 50:



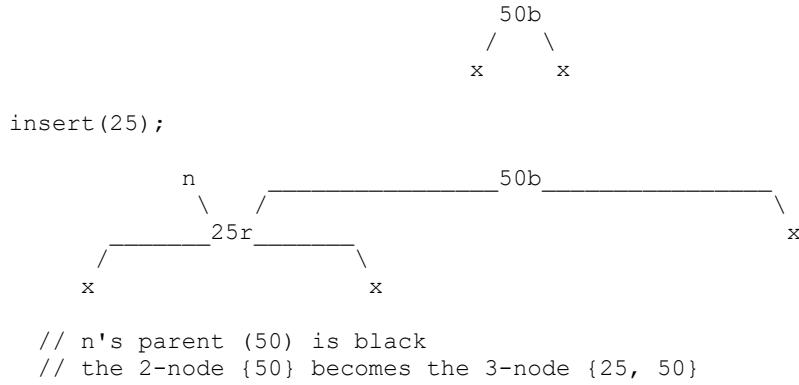
```
n->paintBlack();
```



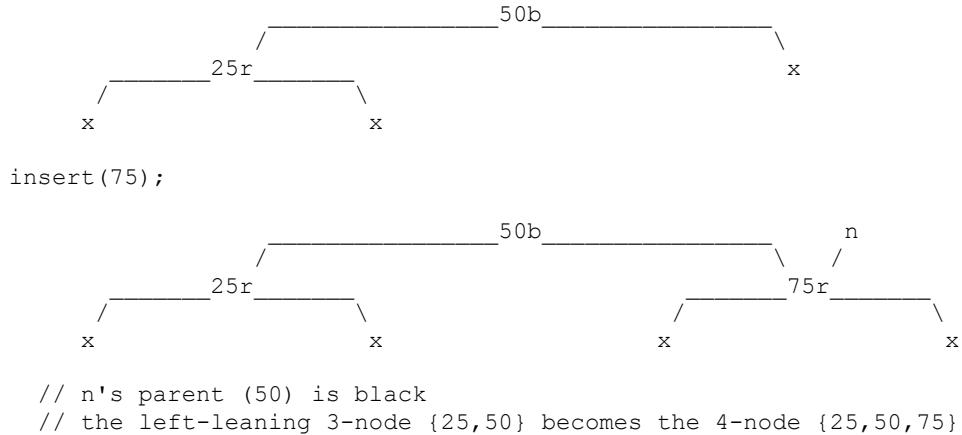
Case 2: n 's parent is black

If n 's parent is black, then there are 3 possibilities, none of which require any balancing:

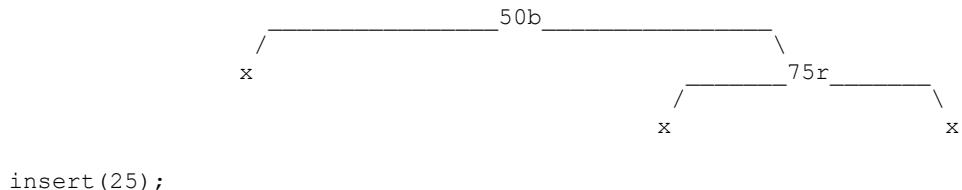
- We've inserted n into a 2-node, which becomes a 3-node:

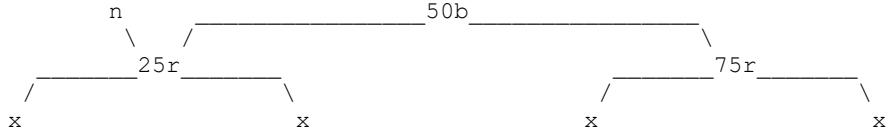


- We've inserted n at the right-hand side of a left-leaning 3-node, which becomes a 4-node:



- We've inserted n at the left-hand side of a right-leaning 3-node, which becomes a 4-node:

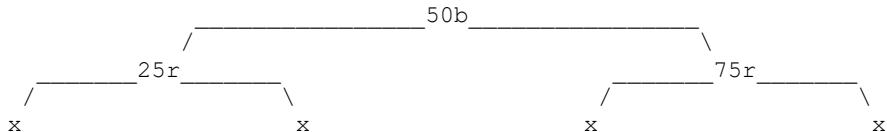




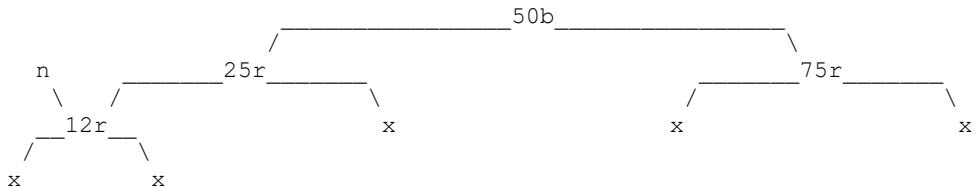
```
// n's parent (50) is black
// the right-leaning 3-node {50, 75} becomes the 4-node {25, 50, 75}
```

Case 3: n's uncle is red

If we've arrived at Case 3, then n 's parent is red. If n 's uncle is also red, then we've inserted n into a 4-node, which now overflows:



```
insert(12);
```



```
// n's uncle (75) is red, so we've inserted n into a 4-node
// the 4-node {25, 50, 75} becomes {12, 25, 50, 75} (overflows)
```

Recall that when a B-tree node overflows, we correct the violation by promoting the middle element and splitting the node in half:

```
25 50 75      // B-tree 4-node

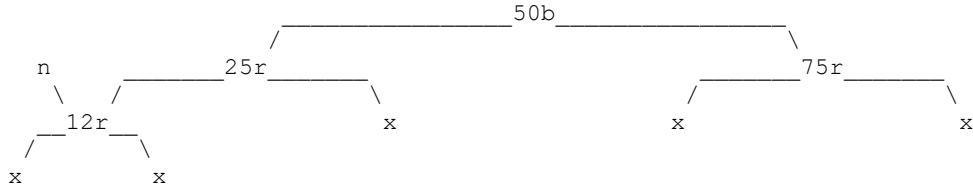
insert(12);

12 25 50 75  // overflow

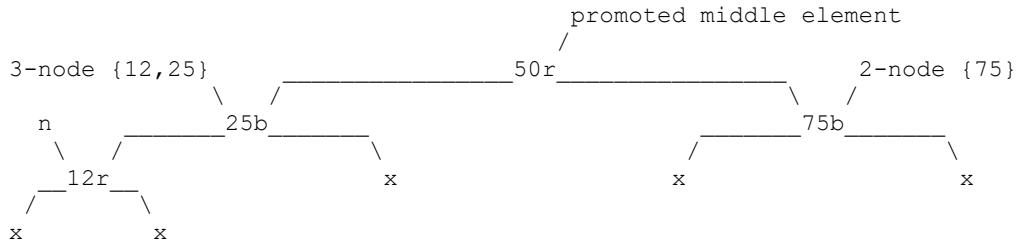
// promote the middle element (50) and split the node

            2-node
            /
3-node      50      2-node
           \   / \   /
           12 25 75
```

In the red-black equivalent of this procedure, we simply paint the middle element (n 's grandparent) red and paint the left and right elements (n 's parent and uncle) black:

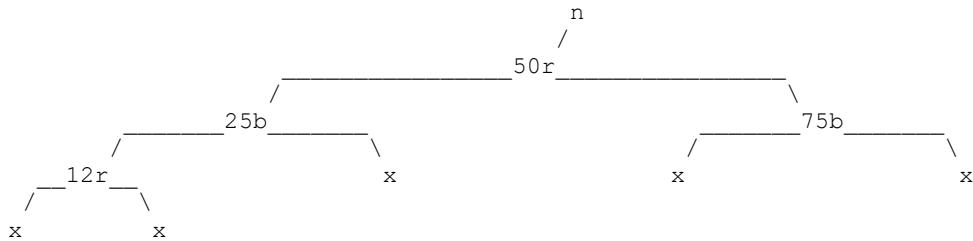


```
grandparent(n)->paintRed();
n->parent->paintBlack();
uncle(n)->paintBlack();
```

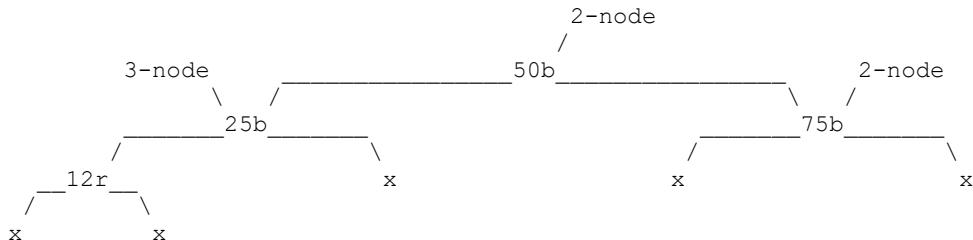


Painting the middle element red promotes it within the underlying B-tree, which could create another overflow. We therefore need to check the promoted element (n 's grandparent) for any balance violations:

```
n = grandparent(n);
```



```
// n is the root (Case 1), so paint it black, after which we're done
```

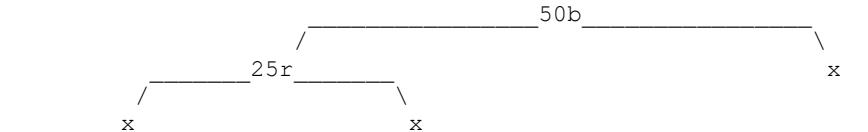


Case 4: n 's uncle is black

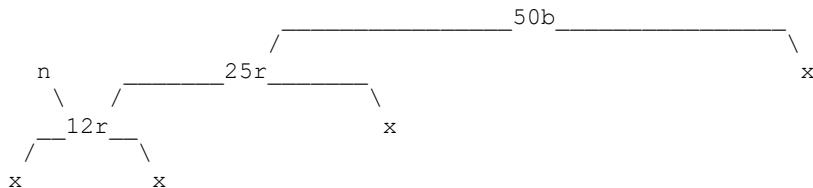
If n 's parent is red and its uncle is black, then we've inserted n into a 3-node. Although this doesn't

cause any overflow, it does result in a red node (n 's parent) having a red child (n), which violates the property that each red node must have 2 black children. There are 4 types of Case 4 violations, each of which is corrected via rotation and recoloring. The original 3-node becomes a 4-node, so no further balancing is necessary.

The first type of Case 4 violation, a *left-left violation*, occurs when n 's parent is the left child of its parent and n is the left child of its parent (n was inserted at the left-hand side of a left-leaning 3-node):



`insert(12);`

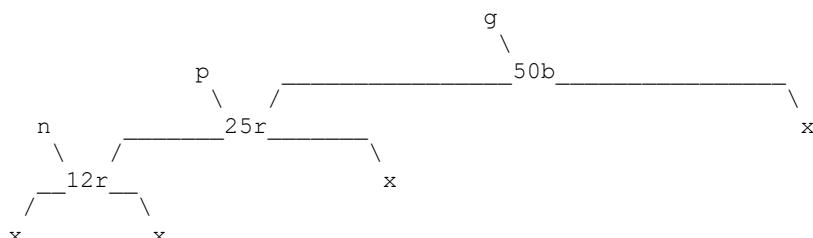


// n 's parent (25) is red and n 's uncle (null) is black,
// so we've inserted n into a 3-node (Case 4)

// n 's parent (25) is the left child of its parent (50), and
// n (12) is the left child of its parent (25)

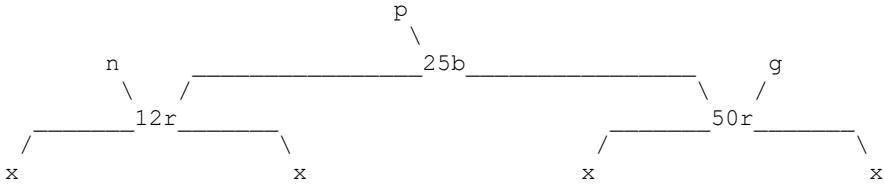
// left-left violation at n (node 12)

To correct the violation, we perform a right rotation on n 's parent, paint the parent black, and paint n 's grandparent red:



`rotateRight(p);`

`p->paintBlack();`
`g->paintRed();`



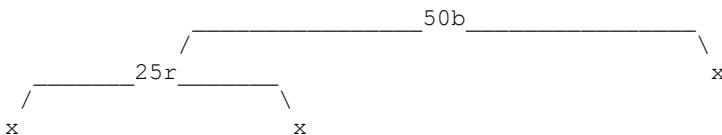
```
// the left-leaning 3-node {25,50} becomes the 4-node {12,25,50}
```

The private member function (*RedBlackTree.h*, line 74)

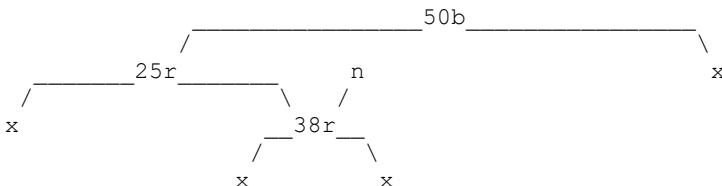
```
void _fixLeftLeft(Node* n);
```

corrects the left-left violation at the given node *n* (*memberFunctions_3.h*, lines 5, 100-113). We've reused the *rotateRight* function from Volume 1, provided by the header file *dss/bt/rotate.h* (which also includes the mirror image, *rotateLeft*).

The second type of Case 4 violation, a *left-right* violation, occurs when *n*'s parent is the left child of its parent and *n* is the right child of its parent (*n* was inserted in the middle of a left-leaning 3-node):



```
insert(38);
```

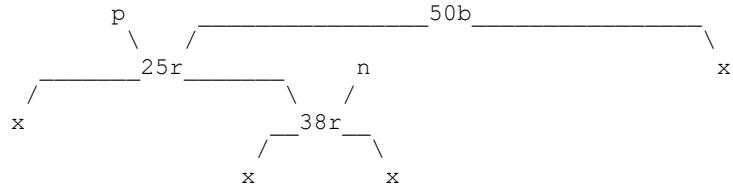


```
// n's parent (25) is red and n's uncle (null) is black,  
// so we've inserted n into a 3-node (Case 4)
```

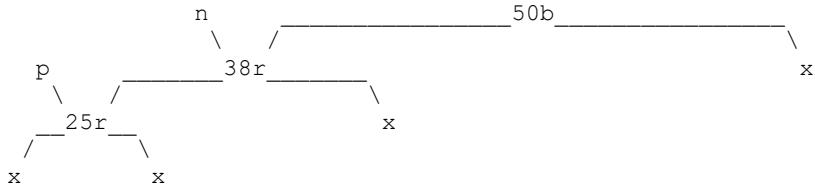
```
// n's parent (25) is the left child of its parent (50), and  
// n (38) is the right child of its parent (25)
```

```
// left-right violation at n (node 38)
```

To correct the violation, we begin by performing a left rotation on *n*:

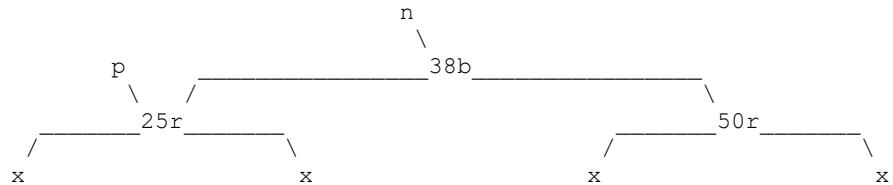


```
rotateLeft(n);
```



This transforms the original violation into a left-left violation at n 's original parent p , which we can then correct via `_fixLeftLeft`:

```
_fixLeftLeft(p);
```



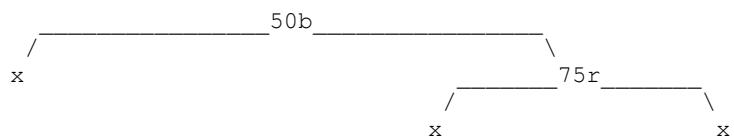
```
// the left-leaning 3-node {25,50} becomes the 4-node {25,38,50}
```

The private member function (*RedBlackTree.h*, line 75)

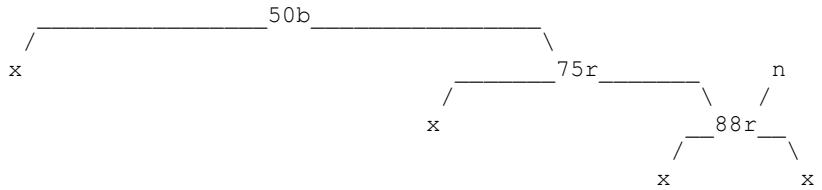
```
void _fixLeftRight(Node* n);
```

corrects the left-right violation at the given node n (*memberFunctions_3.h*, lines 115-124).

The third type of Case 4 violation, a *right-right* violation, is the mirror image of a left-left violation. This occurs when n 's parent is the right child of its parent and n is the right child of its parent (n was inserted at the right-hand side of a right-leaning 3-node):



```
insert(88);
```

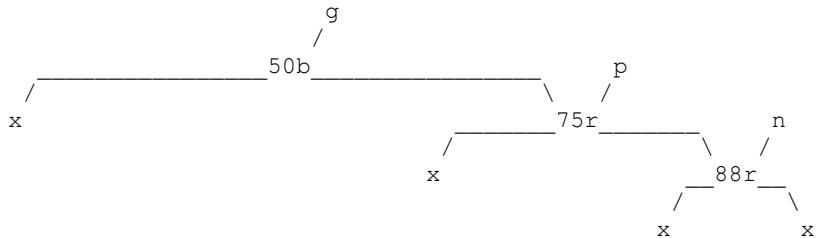


```
// n's parent (75) is red and n's uncle (null) is black,
// so we've inserted n into a 3-node (Case 4)

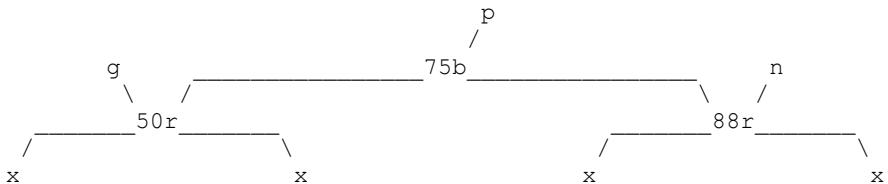
// n's parent (75) is the right child of its parent (50), and
// n (88) is the right child of its parent (75)

// right-right violation at n (node 88)
```

To correct the violation, we perform a left rotation on n 's parent, paint the parent black, and paint n 's grandparent red:



```
rotateLeft(p);
p->paintBlack();
g->paintRed();
```



```
// the right-leaning 3-node {50, 75} becomes the 4-node {50, 75, 88}
```

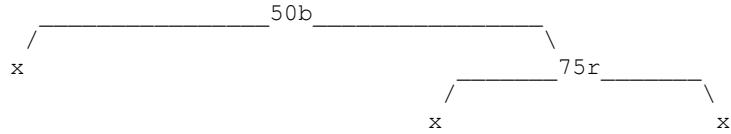
The private member function (*RedBlackTree.h*, line 76)

```
void _fixRightRight(Node* n);
```

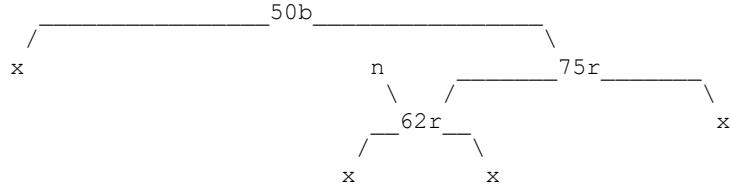
corrects the right-right violation at the given node n (*memberFunctions_3.h*, lines 126-139).

The fourth type of Case 4 violation, a *right-left violation*, is the mirror image of a left-right violation. This occurs when n 's parent is the right child of its parent and n is the left child of its parent (n was

inserted in the middle of a right-leaning 3-node):



`insert(62);`

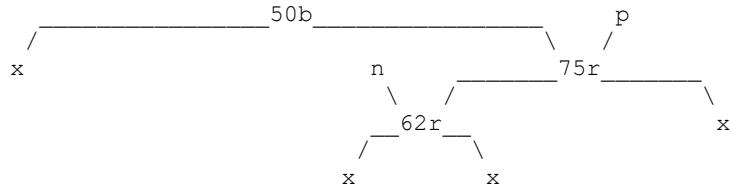


// n's parent (75) is red and n's uncle (null) is black,
// so we've inserted n into a 3-node (Case 4)

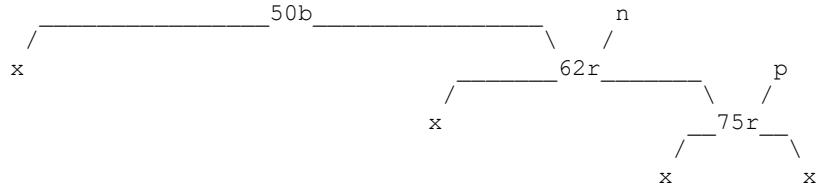
// n's parent (75) is the right child of its parent (50), and
// n (62) is the left child of its parent (75)

// right-left violation at n (node 62)

To correct the violation, we begin by performing a right rotation on *n*:

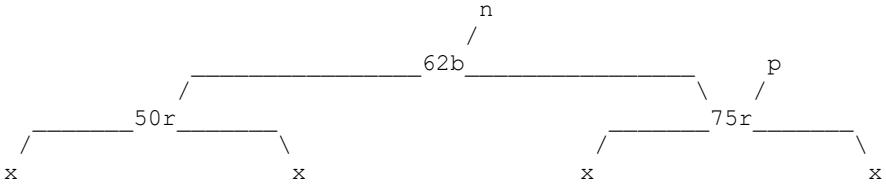


`rotateRight(n);`



This transforms the original violation into a right-right violation at *n*'s original parent *p*, which we can then correct via `_fixRightRight(p);`

`_fixRightRight(p);`



```
// the right-leaning 3-node {50,75} becomes the 4-node {50,62,75}
```

The private member function (*RedBlackTree.h*, line 77)

```
void _fixRightLeft(Node* n);
```

corrects the right-left violation at the given node *n* (*memberFunctions_3.h*, lines 141-150).

We're now ready to write *_balanceOnInsert*. Beginning at the newly inserted node *n*, each iteration of the loop proceeds through the 4 cases:

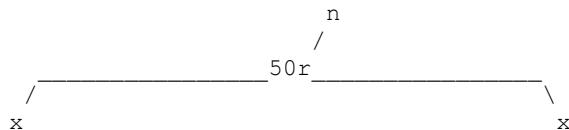
- Case 1 (*n* is the root) (lines 67-71)
- Case 2 (lines 72-75)
 - *n* was inserted into a 2-node
 - *n* was inserted at the left-hand side of a right-leaning 3-node
 - *n* was inserted at the right-hand side of a left-leaning 3-node
- Case 3 (*n* was inserted into a 4-node) (lines 76-83)
 - This is the only case in which we need to proceed up the tree and check for additional balance violations, caused by the promote-and-split procedure.
- Case 4 (lines 84-96)
 - Left-left (*n* was inserted at the left-hand side of a left-leaning 3-node)
 - Left-right (*n* was inserted in the middle of a left-leaning 3-node)
 - Right-right (*n* was inserted at the right-hand side of a right-leaning 3-node)
 - Right-left (*n* was inserted in the middle of a right-leaning 3-node)

Our test program (*main.cpp*) begins by constructing a *RedBlackTree<int, int>* *r* and a *PrintRedBlackNode* function object (lines 14-17). In each iteration of the loop, we prompt the user for a *command* (*i* to insert, *p* to print, *c* to clear, *q* to quit) (lines 21-23) and branch accordingly:

- If the user entered *i*, we prompt them for the *keyValue*, create an element (*pair*) out of it, and insert it into the tree (lines 25-32)
- If the user entered *p*, we print the structure of the tree using *traverseInOrder* (lines 33-37)
- If the user entered *c*, we clear the tree (lines 38-41)
- If the user entered *q*, we terminate the loop and exit *main* (lines 42-45)
- If the user entered some other *command*, we print the message "Invalid command" and begin the next iteration (lines 46-49)

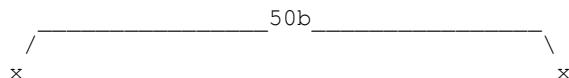
The following sample run demonstrates each of the 4 insertion cases:

```
insert 50
```



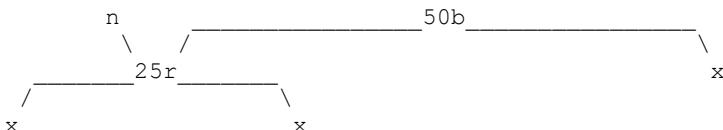
```
// case 1: n is the root
```

```
n->paintBlack();
```



```
node 50
black
```

```
insert 25
```

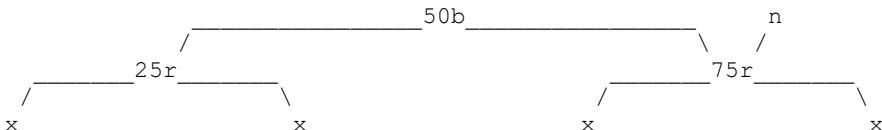


```
// case 2: n was inserted into a 2-node
```

```
// no balancing required
```

```
node 25
parent 50
red
node 50
left 25
black
```

```
insert 75
```



```
// case 2: n was inserted at the right-hand side of a
// left-leaning 3-node
```

```
// no balancing required
```

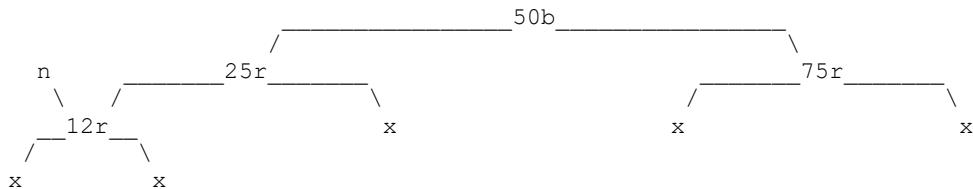
```
node 25
```

```

parent 50
red
node 50
left 25
right 75
black
node 75
parent 50
red

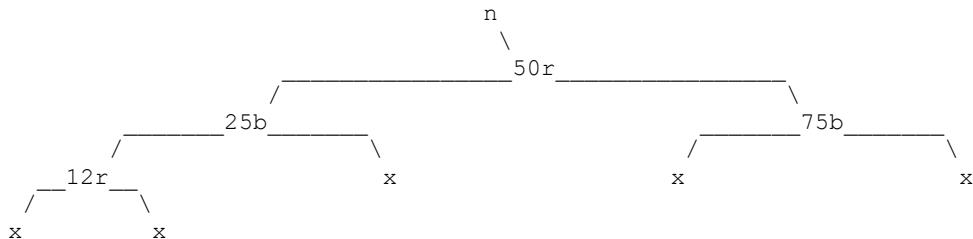
```

```
insert 12
```



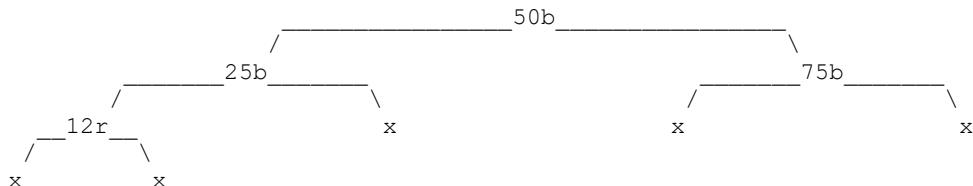
```
// case 3: n was inserted into a 4-node
```

```
// promote 50, split 4-node {25,50,75}
// proceed to n's grandparent (50)
```



```
// case 1: n is the root
```

```
n->paintBlack();
```



```

node 12
parent 25
red
node 25
parent 50
left 12

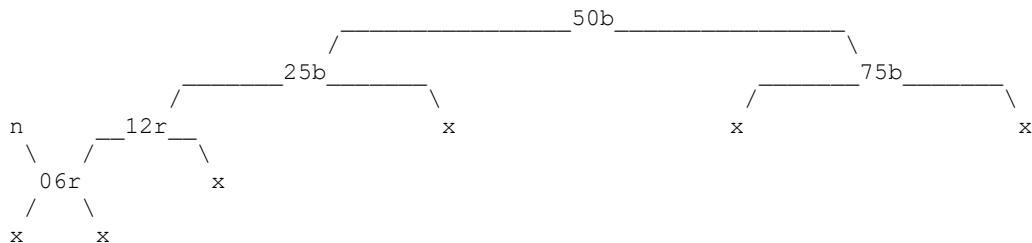
```

```

black
node 50
  left 25
  right 75
  black
node 75
  parent 50
  black

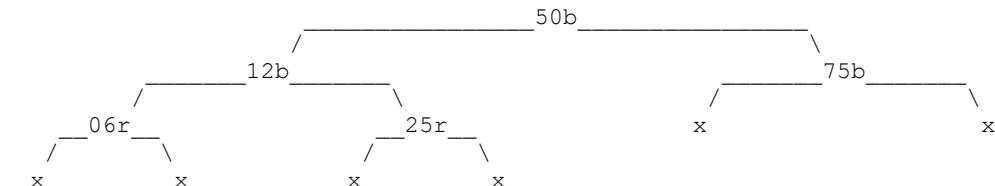
```

```
insert 6
```



```
// case 4, left-left: n was inserted at the left-hand side of a
// left-leaning 3-node
```

```
_fixLeftLeft(n);
```

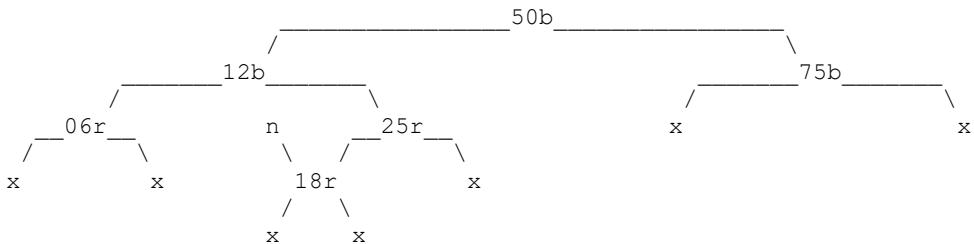


```

node 6
  parent 12
  red
node 12
  parent 50
  left 6
  right 25
  black
node 25
  parent 12
  red
node 50
  left 12
  right 75
  black
node 75
  parent 50
  black

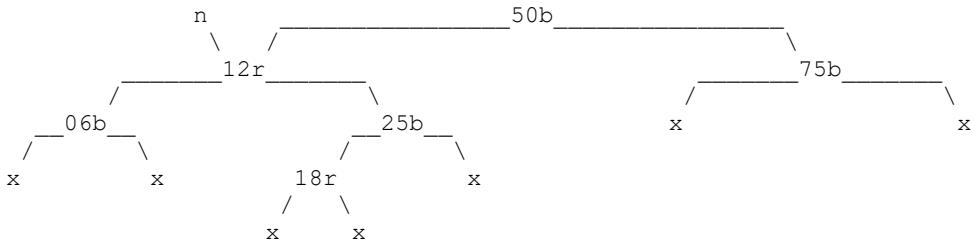
```

```
insert 18
```



```
// case 3: n was inserted into a 4-node
```

```
// promote 12, split 4-node {6,12,25}
// proceed to n's grandparent (12)
```



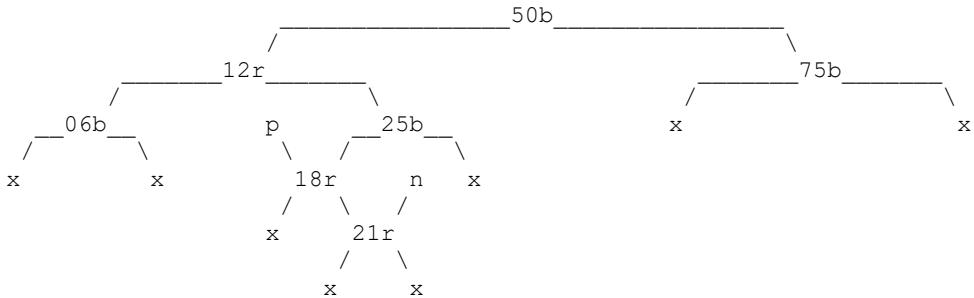
```
// case 2: n was inserted into a 2-node
```

```
// no balancing required
```

```

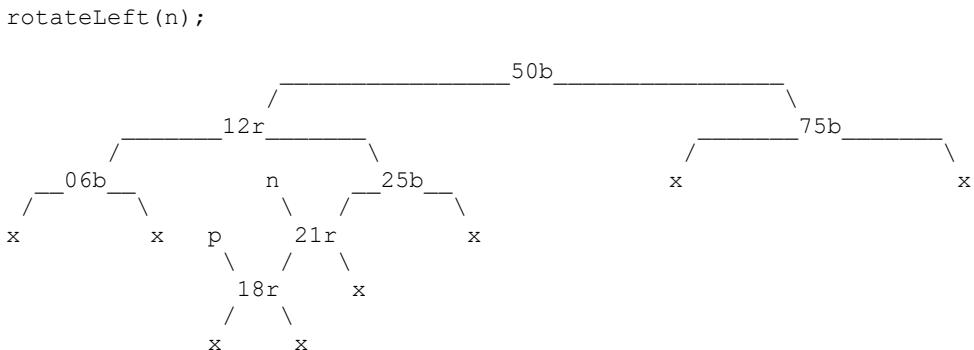
node 6
  parent 12
  black
node 12
  parent 50
  left 6
  right 25
  red
node 18
  parent 25
  red
node 25
  parent 12
  left 18
  black
node 50
  left 12
  right 75
  black
node 75
  parent 50
  black
  
```

```
insert 21
```

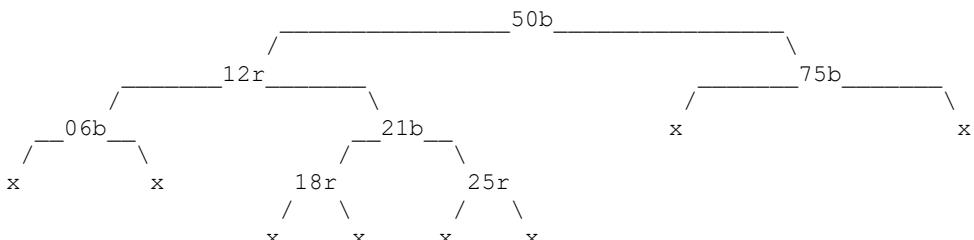


```
// case 4, left-right: n was inserted in the middle of a
// left-leaning 3-node
```

```
_fixLeftRight(n);
```



```
_fixLeftLeft(p);
```



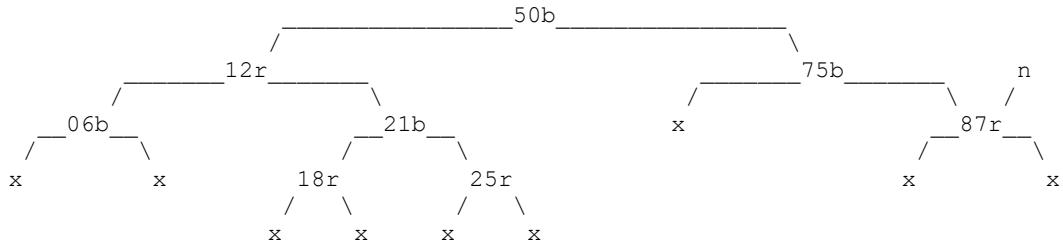
```
node 6
parent 12
black
node 12
parent 50
left 6
right 21
red
node 18
```

```

parent 21
red
node 21
parent 12
left 18
right 25
black
node 25
parent 21
red
node 50
left 12
right 75
black
node 75
parent 50
black

```

insert 87



// case 2: n was inserted into a 2-node

// no balancing required

```

node 6
parent 12
black
node 12
parent 50
left 6
right 21
red
node 18
parent 21
red
node 21
parent 12
left 18
right 25
black
node 25
parent 21
red

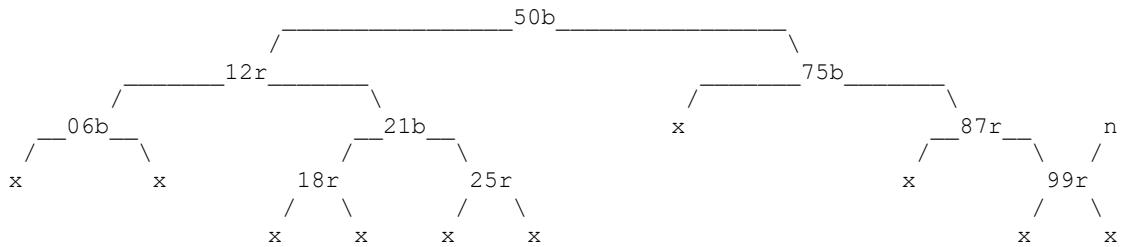
```

```

node 50
  left 12
  right 75
  black
node 75
  parent 50
  right 87
  black
node 87
  parent 75
  red

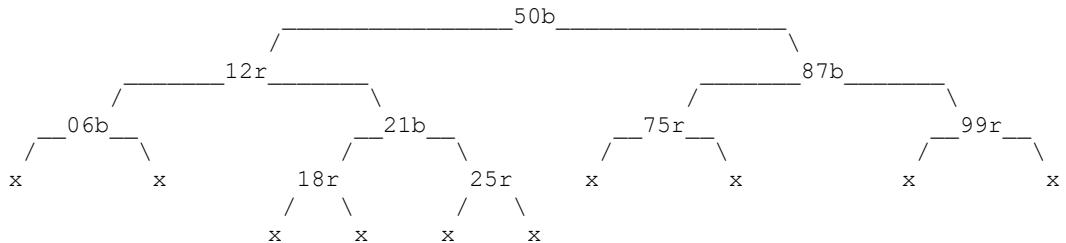
```

insert 99



// case 4, right-right: n was inserted at the right-hand side of a
// right-leaning 3-node

_fixRightRight(n);



```

node 6
  parent 12
  black
node 12
  parent 50
  left 6
  right 21
  red
node 18
  parent 21
  red
node 21
  parent 12
  left 18

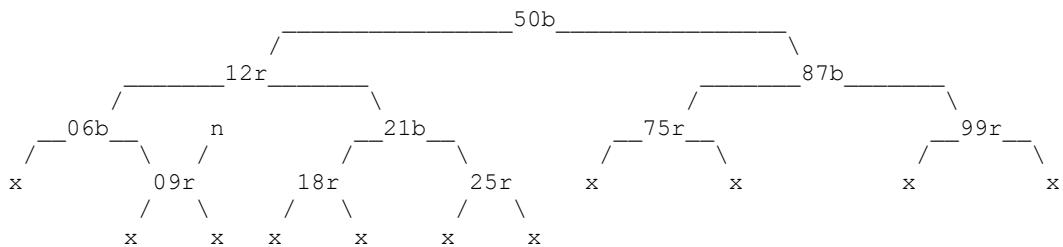
```

```

right 25
black
node 25
parent 21
red
node 50
left 12
right 87
black
node 75
parent 87
red
node 87
parent 50
left 75
right 99
black
node 99
parent 87
red

```

insert 9



// case 2: n was inserted into a 2-node

// no balancing required

```

node 6
parent 12
right 9
black
node 9
parent 6
red
node 12
parent 50
left 6
right 21
red
node 18
parent 21
red
node 21

```

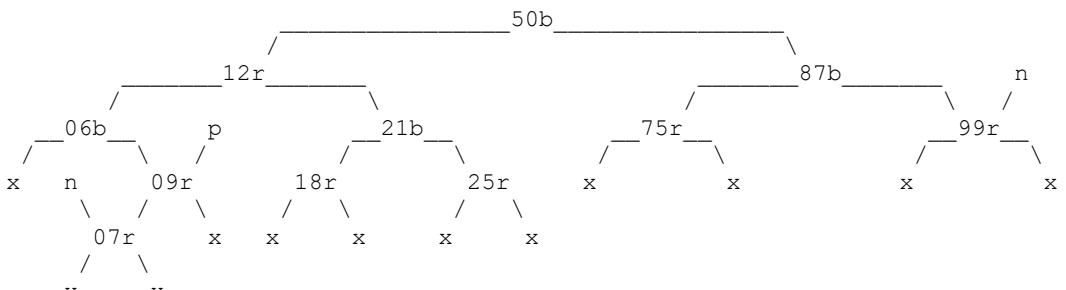
```

parent 12
left 18
right 25
black
node 25
parent 21
red
node 50
left 12
right 87
black
node 75
parent 87
red
node 87
parent 50
left 75
right 99
black
node 99
parent 87
red

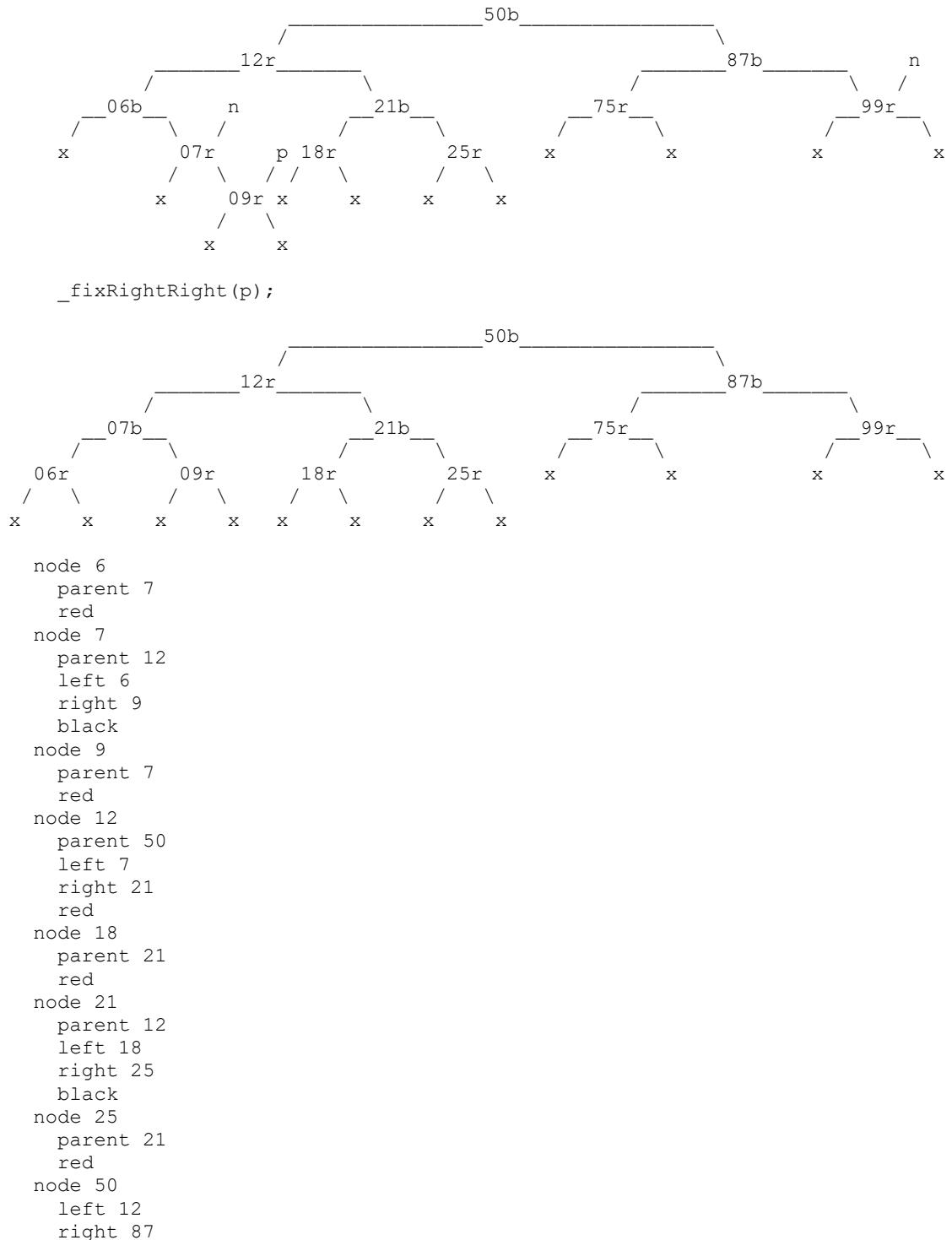
```

```
insert 7
```

```
// case 4, right-left: n was inserted in the middle of a
// right-leaning 3-node
```



```
_fixRightLeft(n);
rotateRight(n);
```

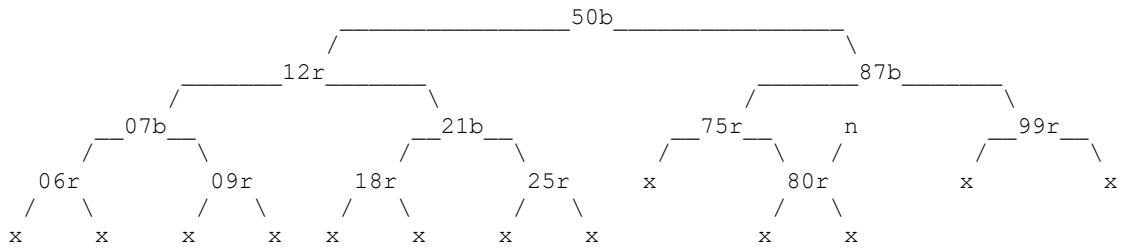


```

black
node 75
  parent 87
  red
node 87
  parent 50
  left 75
  right 99
  black
node 99
  parent 87
  red

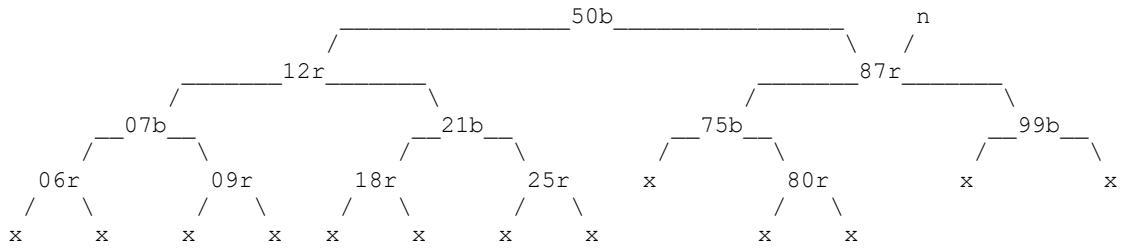
```

```
insert 80
```



```
// case 3: n was inserted into a 4-node
```

```
// promote 87, split 4-node {75, 87, 99}
// proceed to n's grandparent (87)
```



```
// case 2: n was inserted at the right-hand side of a
// left-leaning 3-node
```

```
// no balancing required
```

```

node 6
  parent 7
  red
node 7
  parent 12
  left 6
  right 9
  black

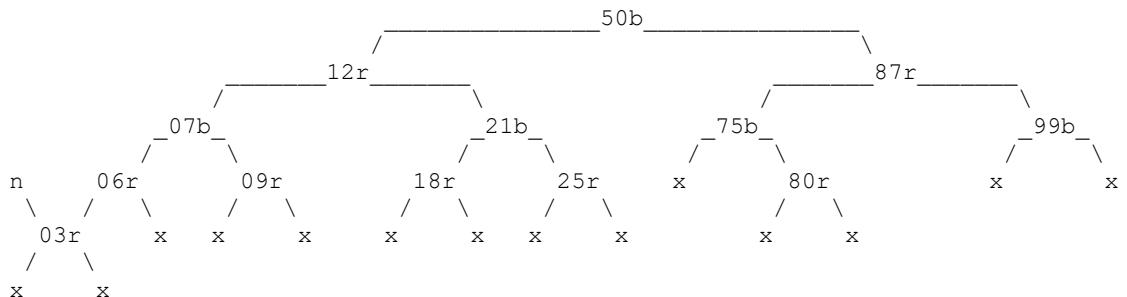
```

```

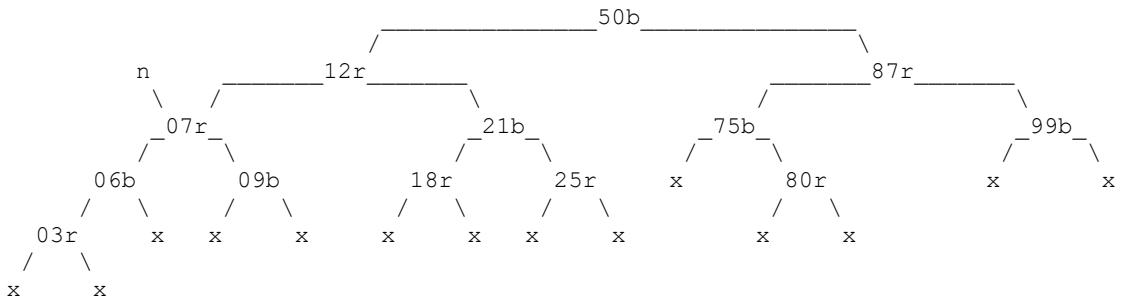
node 9
  parent 7
  red
node 12
  parent 50
  left 7
  right 21
  red
node 18
  parent 21
  red
node 21
  parent 12
  left 18
  right 25
  black
node 25
  parent 21
  red
node 50
  left 12
  right 87
  black
node 75
  parent 87
  right 80
  black
node 80
  parent 75
  red
node 87
  parent 50
  left 75
  right 99
  red
node 99
  parent 87
  black

```

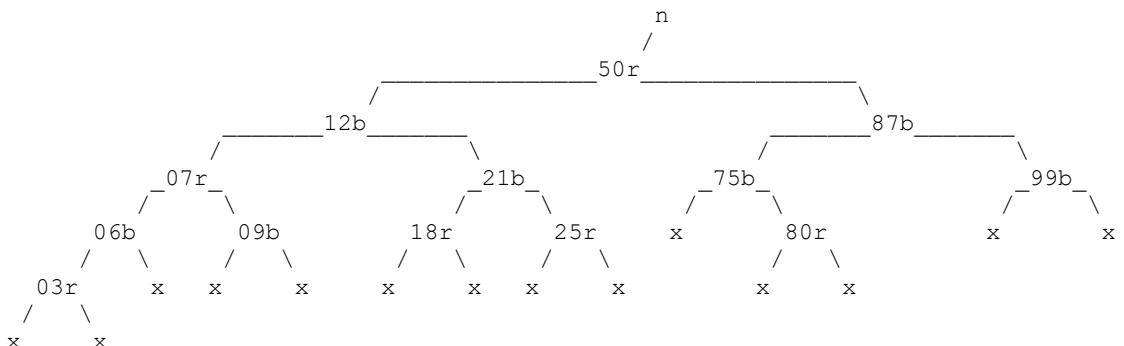
insert 3



```
// case 3: n was inserted into a 4-node
// promote 7, split 4-node {6,7,9}
// proceed to n's grandparent (7)
```

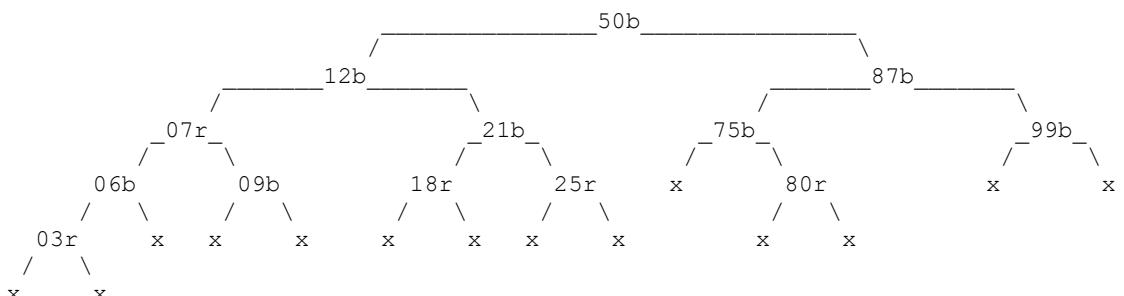


```
// case 3: n was inserted into a 4-node
// promote 50, split 4-node {12,50,87}
// proceed to n's grandparent (50)
```



```
// case 1: n is the root
```

```
n->paintBlack();
```

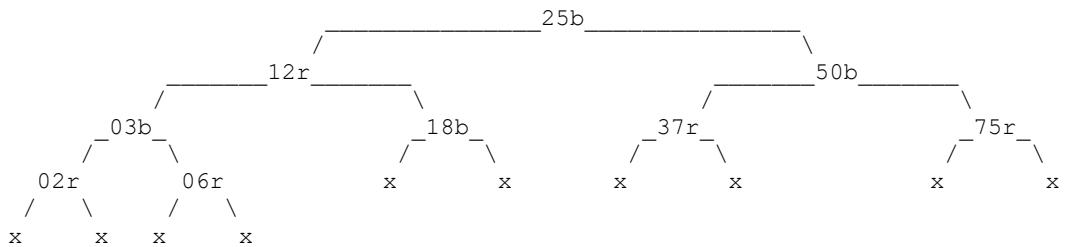


```
node 3
parent 6
```

```
    red
node 6
    parent 7
    left 3
    black
node 7
    parent 12
    left 6
    right 9
    red
node 9
    parent 7
    black
node 12
    parent 50
    left 7
    right 21
    black
node 18
    parent 21
    red
node 21
    parent 12
    left 18
    right 25
    black
node 25
    parent 21
    red
node 50
    left 12
    right 87
    black
node 75
    parent 87
    right 80
    black
node 80
    parent 75
    red
node 87
    parent 50
    left 75
    right 99
    black
node 99
    parent 87
    black
```

```
clear
```

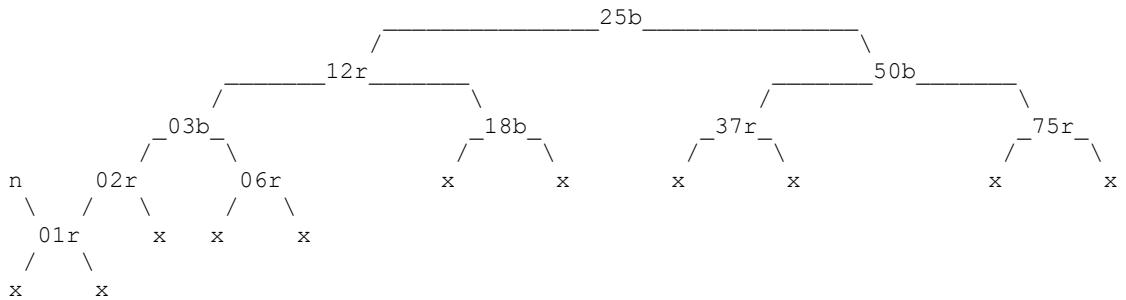
```
insert 50, 25, 12, 6, 18, 37, 75, 3, 2
```



```

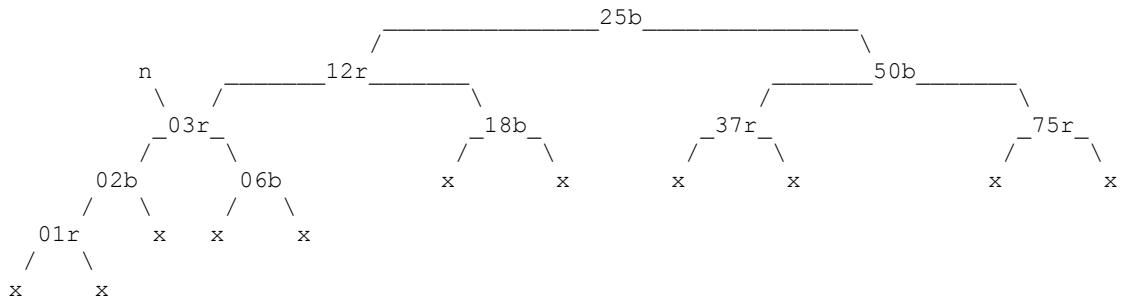
node 2
  parent 3
  red
node 3
  parent 12
  left 2
  right 6
  black
node 6
  parent 3
  red
node 12
  parent 25
  left 3
  right 18
  red
node 18
  parent 12
  black
node 25
  left 12
  right 50
  black
node 37
  parent 50
  red
node 50
  parent 25
  left 37
  right 75
  black
node 75
  parent 50
  red
  
```

insert 1



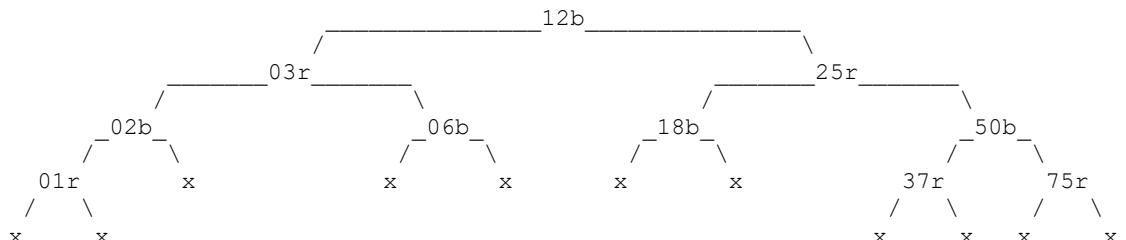
// case 3: n was inserted into a 4-node

// promote 3, split 4-node {2,3,6}
 // proceed to n's grandparent (3)



// case 4, left-left: n was inserted at the left-hand side of a
 // left-leaning 3-node

_fixLeftLeft(n);



```

node 1
parent 2
red
node 2
parent 3
left 1
black
node 3
parent 12
left 2
  
```

```
right 6
red
node 6
    parent 3
    black
node 12
    left 3
    right 25
    black
node 18
    parent 25
    black
node 25
    parent 12
    left 18
    right 50
    red
node 37
    parent 50
    red
node 50
    parent 25
    left 37
    right 75
    black
node 75
    parent 50
    red
```


7.3: Erasing Elements

Source files and folders

- *RedBlackTree/3*
- *RedBlackTree/common/memberFunctions_4.h*

Chapter outline

- *Implementing RedBlackTree::erase*

Like *insert*, *RedBlackTree*'s *erase* method (*RedBlackTree.h*, line 65) is nearly identical to the original (*BinaryTree*) version. If *trash* (the node to be erased) has 2 children, we swap *trash* with its in-order predecessor, which is guaranteed to have 1 or 0 children (*memberFunctions_4.h*, lines 15-22). After updating update the *_head* and *_tail* pointers (lines 24-28), we then balance the tree (line 30) via the private member function (*RedBlackTree.h*, line 86)

```
void _balanceOnErase(Node* trash);
```

We need to balance the tree before detaching *trash* (*memberFunctions_4.h*, lines 32-35) because once *trash* is removed, we'll no longer have access to its relatives (parent, children, etc.).

Erasure, like insertion, broadly consists of 3 cases:

- Erasure from a 4-node, which becomes a 3-node
- Erasure from a 3-node, which becomes a 2-node
- Erasure from a 2-node, which results in a B-tree underflow

The 3-node and 4-node cases are relatively simple, so we'll divide *_balanceOnErase* into two separate functions. The first function (*RedBlackTree.h*, line 87),

```
bool _balanceOn3Or4NodeErase(Node* trash);
```

determines whether *trash* is part of a 3-node or 4-node, and if so, balances the tree accordingly. A return value of *true* indicates that balancing was performed (because *trash* is part of a 3-node or 4-node). A return value of *false* indicates that no balancing was performed (because *trash* is a 2-node).

The second function (*RedBlackTree.h*, line 88),

```
void _balanceOn2NodeErase(Node* trash);
```

exclusively handles the 2-node case.

To implement *_balanceOnErase*, we first call *_balanceOn3Or4NodeErase* and save the result to *is3Or4NodeCase* (*memberFunctions_4.h*, line 108). If *trash* turns out to be a 2-node, we handle it via

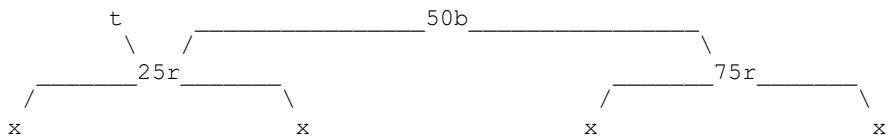
`_balanceOn2NodeErase` (lines 110-111).

We can now implement `_balanceOn3Or4NodeErase`, which consists of 3 cases. In the following diagrams, t denotes *trash*.

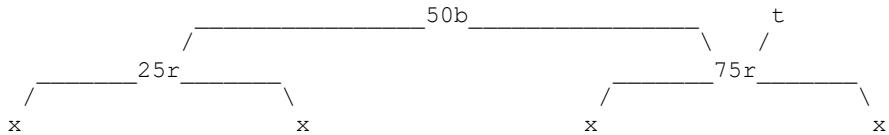
Case 1: *trash* is red (lines 119-120)

If *trash* is red, then *trash* is either

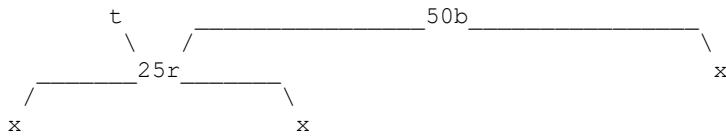
- the left element of a 4-node, as in



- the right element of a 4-node, as in



- the left element of a left-leaning 3-node, as in



- or the right element of a right-leaning 3-node, as in



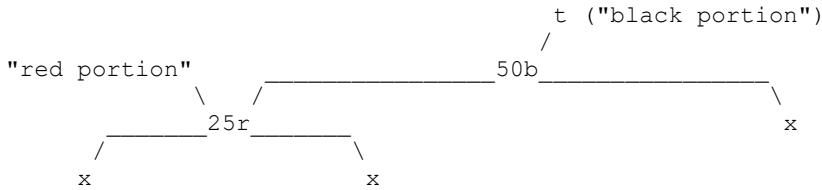
In all of these cases, erasing *trash* won't cause any balance violations:

- The original 4-node becomes a 3-node
- The original 3-node becomes a 2-node

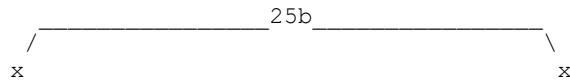
Case 2: *trash*'s left child is red (lines 122-126)

Because *trash* is guaranteed to have at most 1 child, if *trash*'s left child is red, then *trash* must be the

black portion of a left-leaning 3-node, as in

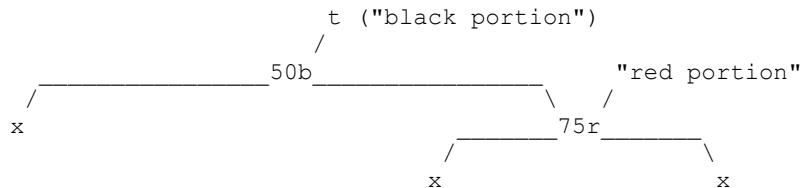


After erasing *trash*, we transform the original 3-node into a 2-node by simply painting *trash*'s left child black:

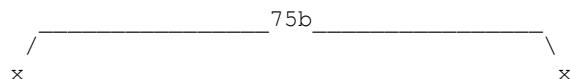


Case 3: *trash*'s right child is red (lines 128-132)

This is the mirror image of Case 2. If *trash*'s right child is red, then *trash* must be the black portion of a right-leaning 3-node, as in



After erasing *trash*, we transform the original 3-node into a 2-node by simply painting *trash*'s right child black:



Before moving on to *_balanceOn2NodeErase*, let's write some helper functions. The first of these (*RedBlackTree.h*, line 69),

```
bool _is2Node(Node* n) const;
```

returns *true* if the given node *n* is a 2-node. For *n* to be a 2-node, the following must be true (*memberFunctions_4.h*, lines 46-56):

- *n* is non-null
- *n* is black
- *n* is either a leaf node (which has two null children, considered to be black), or *n* is a non-leaf node with 2 black children

The next function (*RedBlackTree.h*, line 70),

```
bool _isLeftLeaning3Node(Node* n) const;
```

returns *true* if the given node *n* is the black portion of a left-leaning 3-node. For *n* to be the black portion of a left-leaning 3-node, the following must be true (*memberFunctions_4.h*, lines 62-64):

- *n* is non-null
- *n* is black
- *n*'s left child is red
- *n*'s right child is null (which is considered to be black), or non-null and black

The mirror image (*RedBlackTree.h*, line 71),

```
bool _isRightLeaning3Node(Node* n) const;
```

returns *true* if the given node *n* is the black portion of a right-leaning 3-node. For *n* to be the black portion of a right-leaning 3-node, the following must be true (*memberFunctions_4.h*, lines 70-72):

- *n* is non-null
- *n* is black
- *n*'s right child is red
- *n*'s left child is null (which is considered to be black), or non-null and black

To test whether a given node *n* is the black portion of a 4-node, we'll use the function (*RedBlackTree.h*, line 72)

```
bool _is4Node(Node* n) const;
```

For *n* to be the black portion of a 4-node, the following must be true (*memberFunctions_4.h*, lines 75-89):

- *n* is non-null
- *n* is black
- *n* has 2 red children

The function (*RedBlackTree.h*, line 73)

```
bool _is4NodeOrLeftLeaning3Node(Node* n) const;
```

returns *true* if *n* is either the black portion of a 4-node, or the black portion of a left-leaning 3-node. It simply passes *n* to *_is4Node* and *_isLeftLeaning3Node* (*memberFunctions_4.h*, line 95).

The mirror image (*RedBlackTree.h*, line 74),

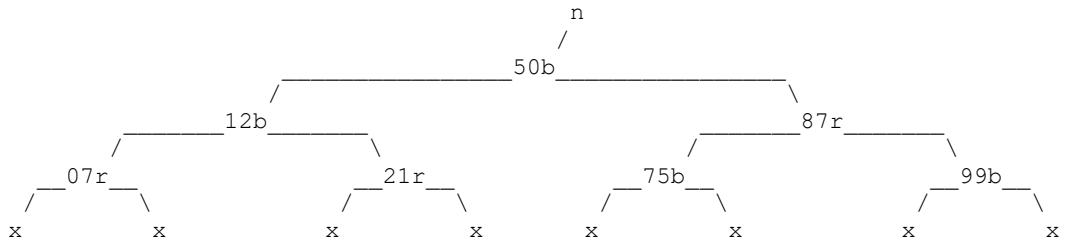
```
bool _is4NodeOrRightLeaning3Node(Node* n) const;
```

returns *true* if *n* is either the black portion of a 4-node, or the black portion of a right-leaning 3-node. It simply passes *n* to *_is4Node* and *_isRightLeaning3Node* (*memberFunctions_4.h*, line 102).

The function (*RedBlackTree.h*, line 89)

```
void _convertToLeftLeaning3Node(Node* blackPortion);
```

converts the right-leaning 3-node with the given *blackPortion* to a left-leaning 3-node. Given the tree

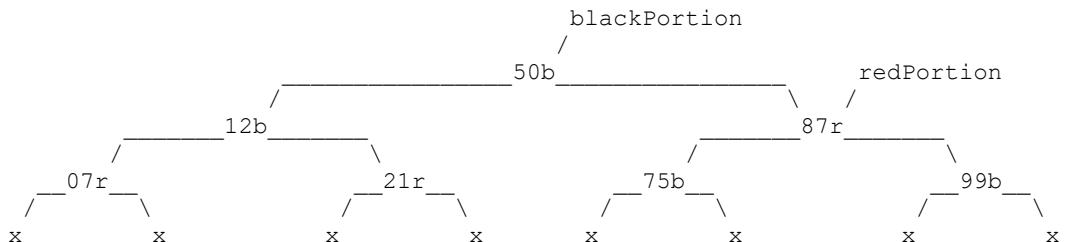


for example, *n* (node 50) is the black portion of the right-leaning 3-node {50, 87}. The function call

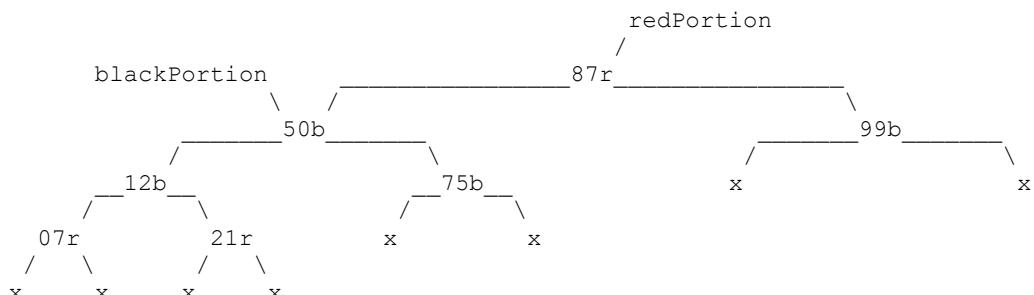
```
_convertToLeftLeaning3Node(n);
```

converts this 3-node to a left-leaning 3-node by performing a left rotation on the red portion (node 87), then recoloring the red and black portions (*memberFunctions_4.h*, lines 191-196):

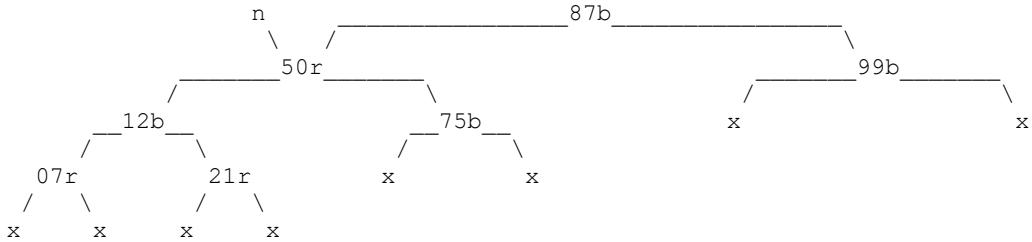
```
redPortion = blackPortion->right;
```



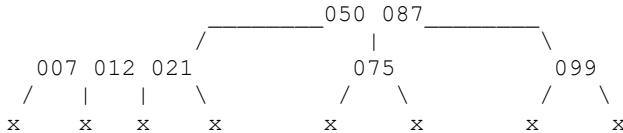
```
rotateLeft(redPortion, &_root);
```



```
redPortion->paintBlack();
blackPortion->paintRed();
```



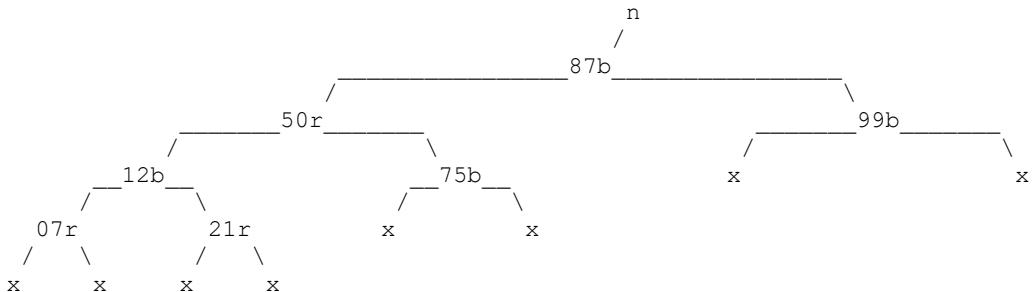
The original right-leaning 3-node $\{50, 87\}$ (black portion 50 / red portion 87) is now a left-leaning 3-node (black portion 87 / red portion 50). As we discussed in Chapter 7.1, both representations are equally valid; performing this conversion doesn't change the underlying B-tree:



The mirror image of this function (*RedBlackTree.h*, line 90),

```
void _convertToRightLeaning3Node(Node* blackPortion);
```

converts a left-leaning 3-node with the given *blackPortion* to a right-leaning 3-node. Given the tree

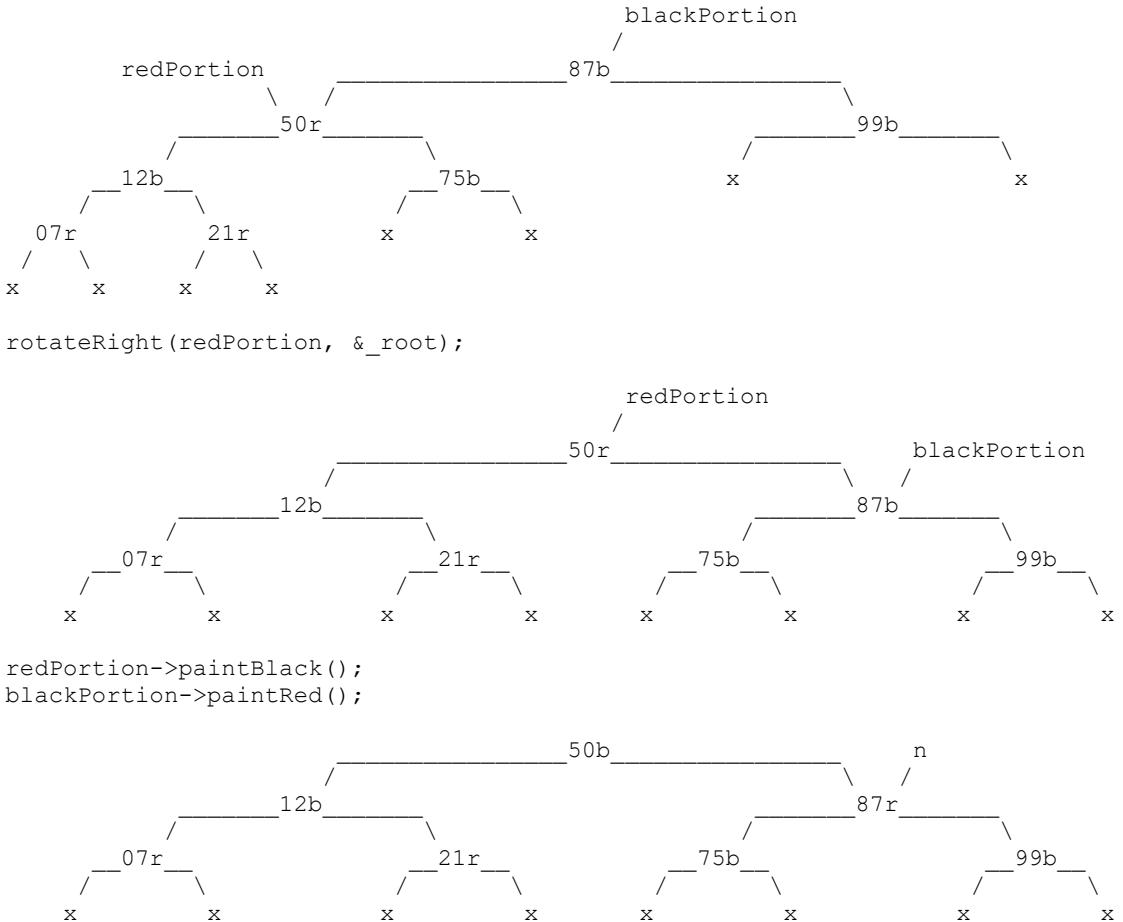


for example, *n* (node 87) is the black portion of the left-leaning 3-node $\{50, 87\}$. The function call

```
_convertToRightLeaning3Node(n);
```

converts this 3-node to a right-leaning 3-node by performing a right rotation on the red portion (node 50), then recoloring the red and black portions (*memberFunctions_4.h*, lines 203-208):

```
redPortion = blackPortion->left;
```



The original left-leaning 3-node {50, 87} (black portion 87 / red portion 50) is now a right-leaning 3-node (black portion 50 / red portion 87).

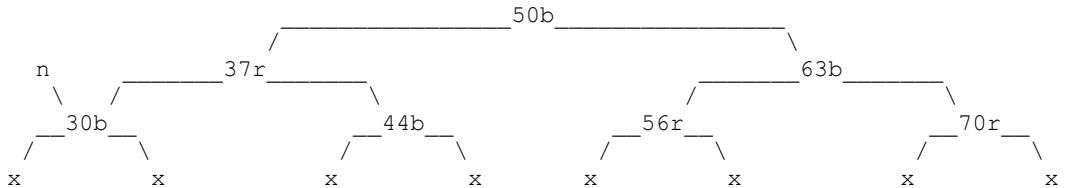
Now that we've written most of our helper functions, let's examine the types of balance violations that can occur upon the removal of a 2-node n . There are 6 cases altogether, 5 of which (Cases 2-6) are symmetrical. We'll begin with Cases 1, 4, and 6, which are easier to understand because they're completely self-contained (correcting them doesn't require any intermediate steps, and no additional balancing is required following the correction).

Case 1: n is the root (*memberFunctions_4.h*, lines 147-148)

If n is the root, it means that the tree contains only 1 element (node), so we can simply erase n without causing any balance violations.

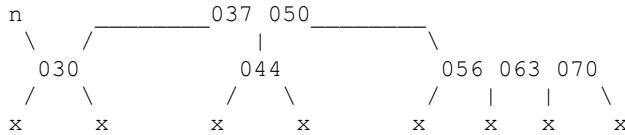
Case 4: n 's parent is red, and n 's sibling is a 2-node (lines 162-167)

If n 's parent is red, then the underlying B-tree parent node must either be a 3-node or 4-node, as in

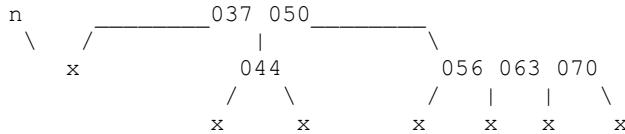


```
// n's parent (37) is red, so the underlying B-tree parent node must
// either be a 3-node or 4-node
```

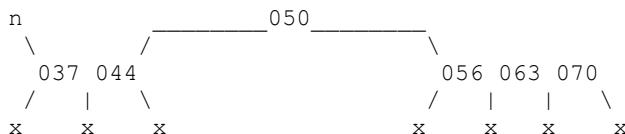
```
// the underlying B-tree parent node, {37,50} is a 3-node
```



Removing n causes the B-tree node $\{30\}$ to underflow:

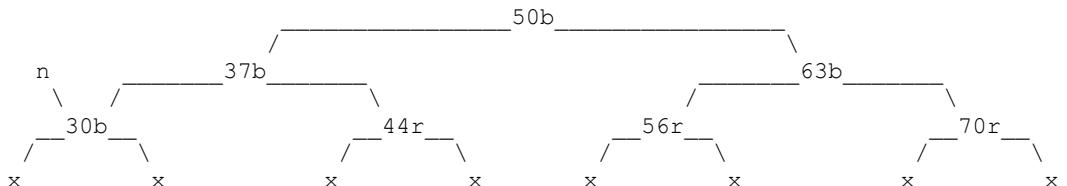


n 's sibling, node $\{44\}$, is a 2-node, so we correct the violation by demoting n 's parent element, 37, to node $\{44\}$. n 's parent node, $\{37, 50\}$, becomes a 2-node, after which no additional balancing is required:

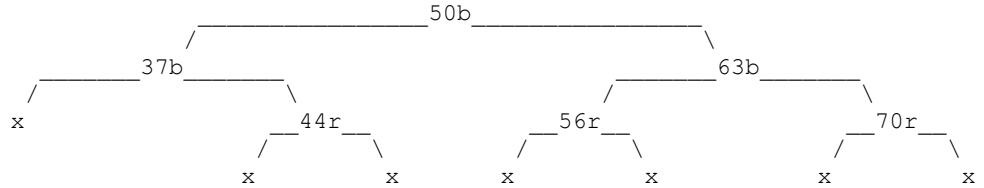


In the red-black equivalent of this operation, we simply paint n 's parent black and n 's sibling red,

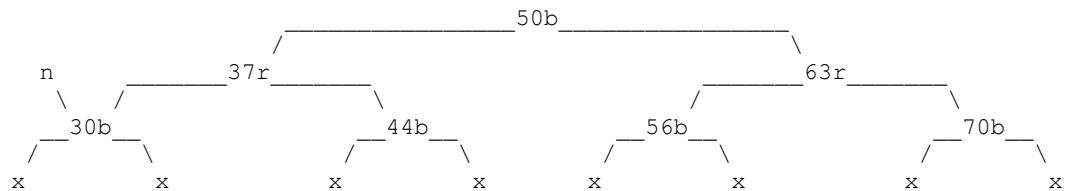
```
n->parent->paintBlack();
sibling(n)->paintRed();
```



after which we remove n :

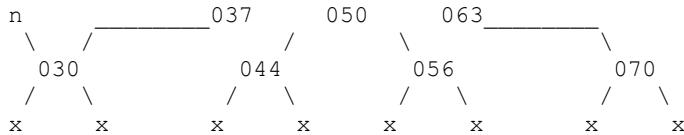


As mentioned above, if n 's parent is red, then the underlying B-tree parent node could also be a 4-node, as in

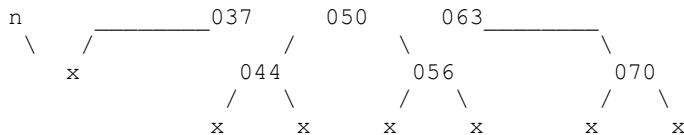


// n 's parent (37) is red, so the underlying B-tree parent node must
// either be a 3-node or 4-node

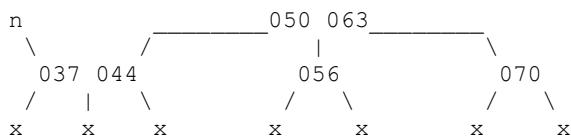
// the underlying B-tree parent node, {37, 50, 63} is a 4-node



After removing n ,

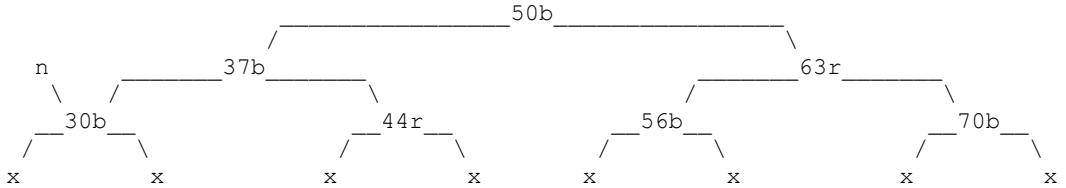


we correct the underflow in the exact same manner (demote n 's parent element, 37, to node {44}). n 's parent node, {37, 50, 63}, becomes a 3-node:

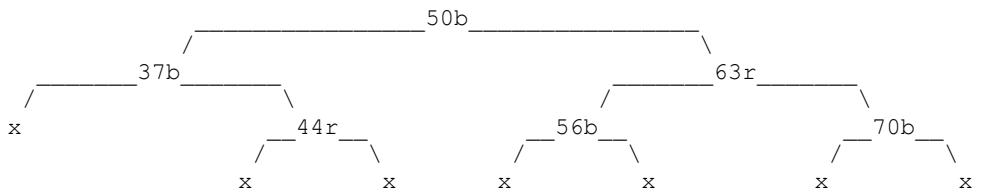


The red-black version of this operation is identical to the previous example: we paint n 's parent black and n 's sibling red,

```
n->parent->paintBlack();
sibling(n)->paintRed();
```



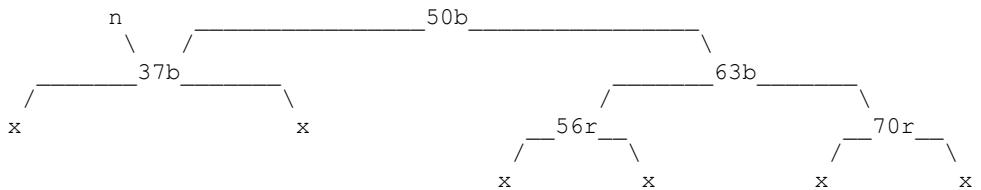
after which we remove *n*:



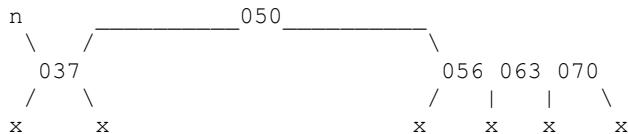
In both of these examples, *n* was the left child and *n*'s 2-node sibling was the right child; in the symmetrical case, *n* is the right child and *n*'s 2-node sibling is the left child. But since our code only references *n->parent* and *sibling(n)*, both cases are handled.

Case 6: *n* is a left child, and *n*'s sibling is either a 4-node or right-leaning 3-node (lines 174-178)
n is a right child, and *n*'s sibling is either a 4-node or left-leaning 3-node (lines 179-183)

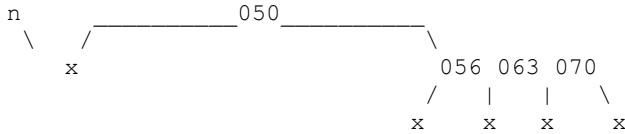
If *n* is a left child and *n*'s sibling is either a 4-node or right-leaning 3-node, we correct the underflow by performing the red-black equivalent of a B-tree left rotation. Consider, for example, the tree



```
// n (node 37) is a left child, and n's sibling (node 63) is a 4-node
```



Removing *n* causes the B-tree node {37} to underflow:

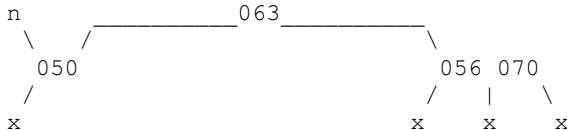


n's sibling, node {56, 63, 70}, is a 4-node, so we correct the violation by performing a B-tree left rotation:

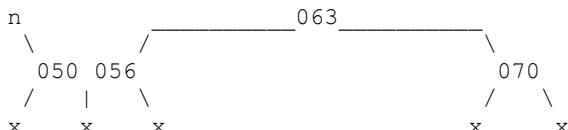
```
// denote n's parent element (50) to n
```



```
// promote the middle element from n's sibling (63) to n's parent
```



```
// move the left element from n's sibling (56) to n
```

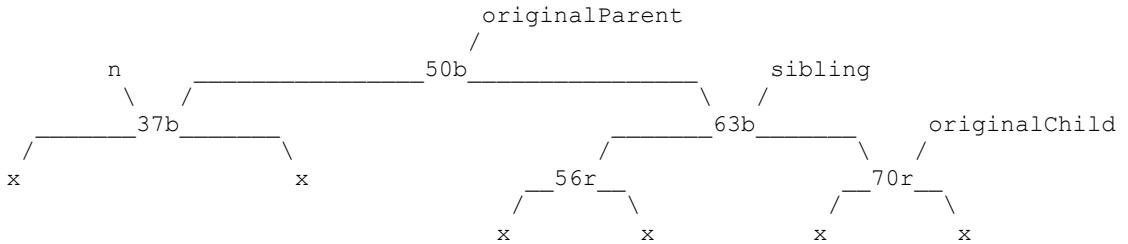


To perform the red-black equivalent of a B-tree left rotation, we'll write another helper function (*RedBlackTree.h*, line 91),

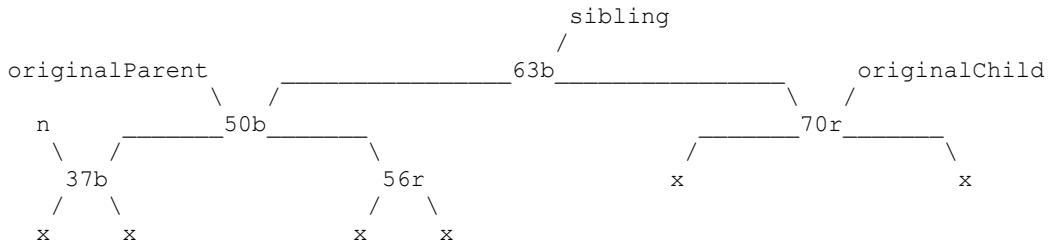
```
void _performBTreeLeftRotation(Node* sibling);
```

where *sibling* refers to the black portion of *n*'s 4-node (or right-leaning 3-node) sibling (*memberFunctions_4.h*, lines 215-222). We begin by left rotating the sibling:

```
originalParent = sibling->parent;
originalChild = sibling->right;
```



```
rotateLeft(sibling, &_root);
```



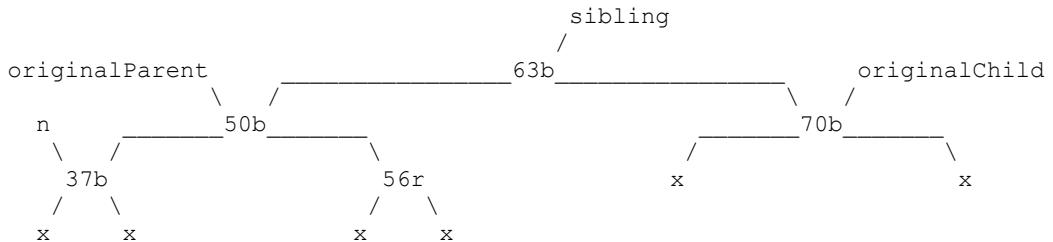
We then recolor the sibling, original parent, and original child,

```

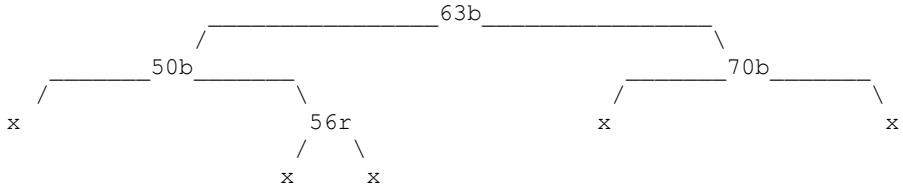
sibling->matchColor(*originalParent);           // the sibling takes the
                                                // originalParent's place (the
                                                // originalParent could've been
                                                // red or black)

originalParent->paintBlack();                  // the originalParent takes
                                                // n's place (n was black)

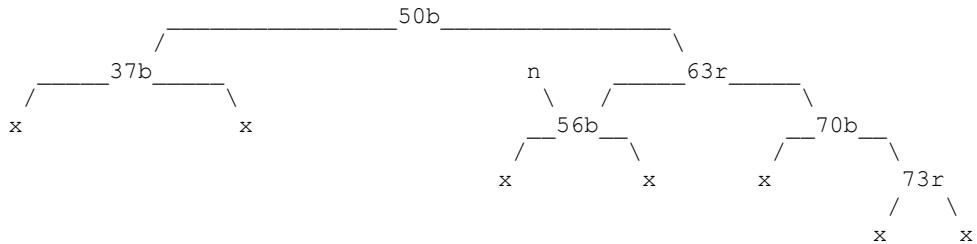
originalChild->paintBlack();                  // the originalChild takes the
                                                // sibling's place (the sibling
                                                // was black)
  
```



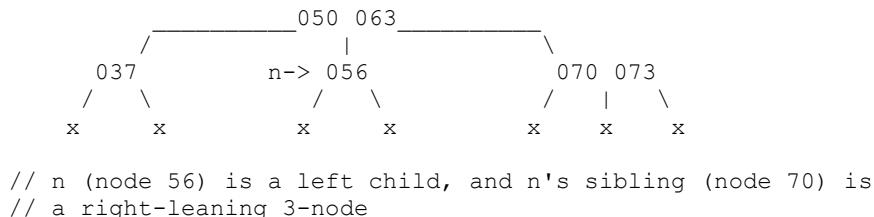
after which we can remove *n*:



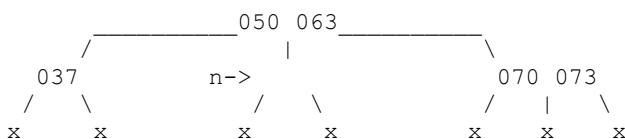
If n 's sibling is a right-leaning 3-node, as in



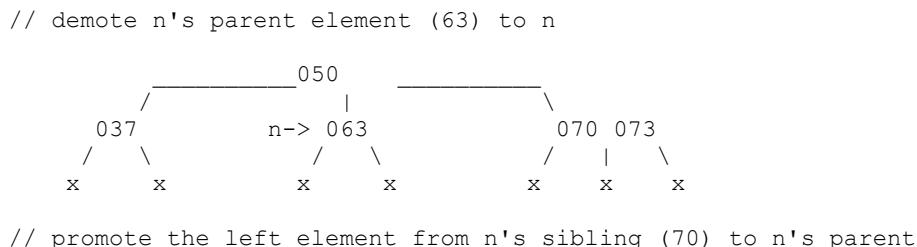
the procedure is the same:

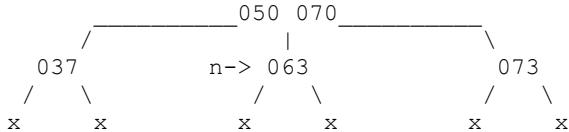


After removing n ,



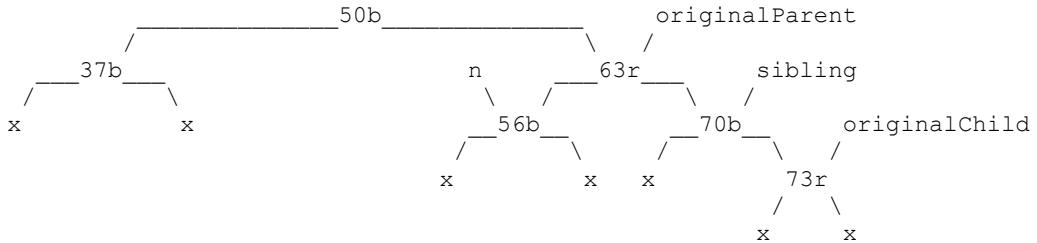
we correct the underflow by performing a B-tree left rotation:



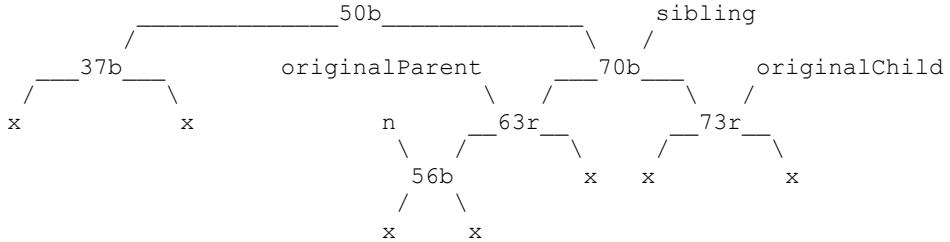


To perform the red-black equivalent of this operation, we left rotate n 's sibling,

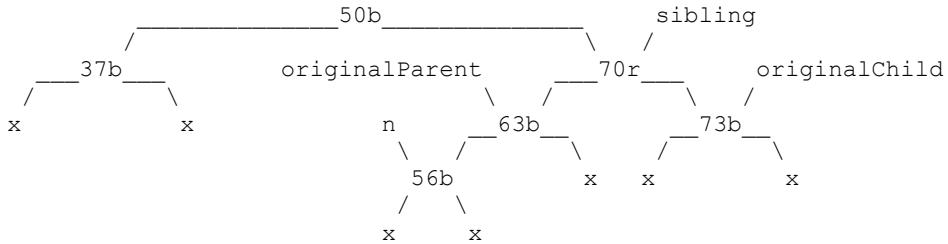
```
originalParent = sibling->parent;  
originalChild = sibling->right;
```



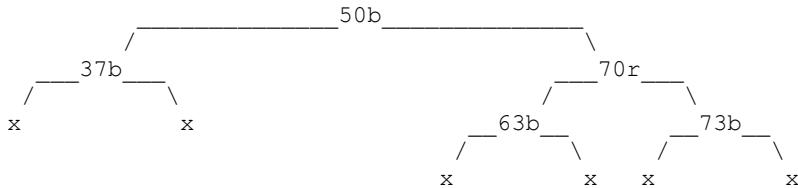
```
rotateLeft(sibling, &_root);
```



recolor,



then remove n :



The private member function (*RedBlackTree.h*, line 92)

```
void _performBTreeRightRotation(Node* sibling);
```

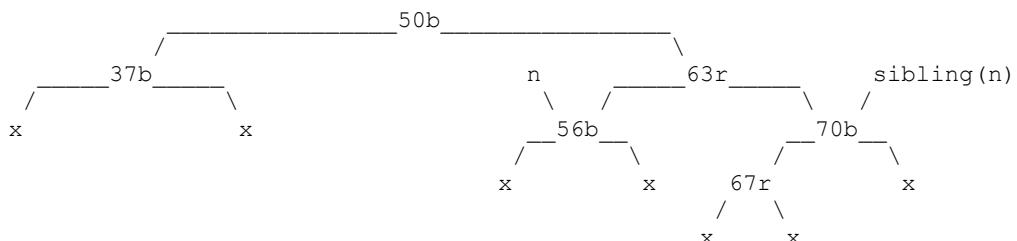
handles the mirror image of Case 6 (n is a right child, and n 's sibling is either a 4-node or left-leaning 3-node) (*memberFunctions_4.h*, lines 179-183, 229-236).

Now that we can easily correct Cases 1, 4, and 6, let's examine Cases 2, 3, and 5.

Case 5: n is a left child, and n 's sibling is a left-leaning 3-node (lines 169-170, 174-178)

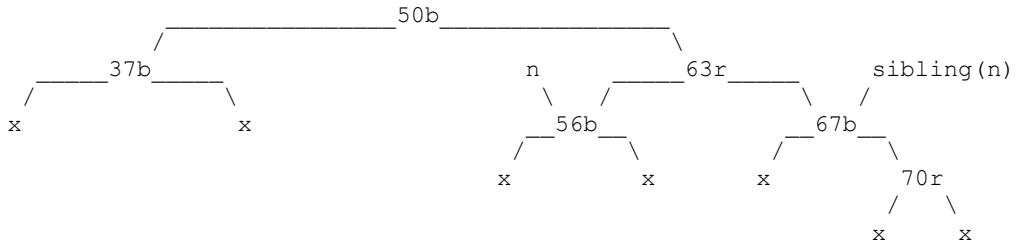
n is a right child, and n 's sibling is a right-leaning 3-node (lines 171-172, 179-183)

Case 5 is a minor variant of Case 6; the only difference lies in the representation of n 's 3-node sibling. To balance Case 5, we transform it into Case 6 by converting n 's sibling from left to right-leaning (or vice versa), which we can then repair by performing the appropriate B-tree rotation. Consider, for example, the tree



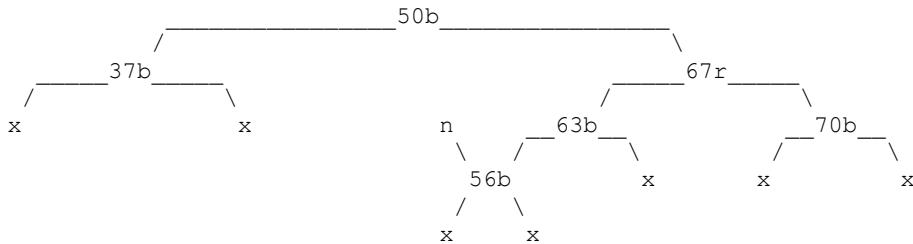
```
// Case 5: n (node 56) is a left child and n's sibling (node 70) is a
// left-leaning 3-node
```

```
_convertToRightLeaning3Node(sibling(n));
```

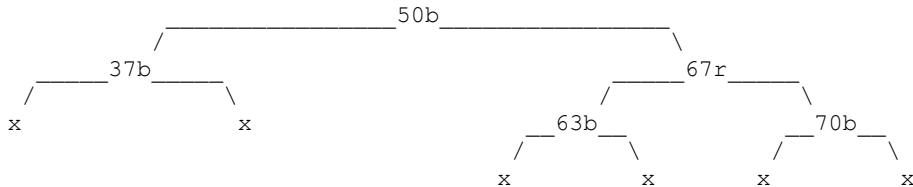


// Case 6: n (node 56) is a left child, and n's sibling (node 67) is either a 4-node or right-leaning 3-node

_performBTreeLeftRotation(sibling(n));



// remove n



The mirror image of Case 5 occurs when n is a right child and n 's sibling is a right-leaning 3-node. To correct the violation, we convert n 's sibling to a left-leaning 3-node and perform a B-tree right rotation.

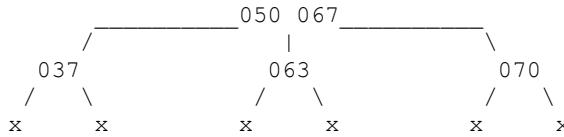
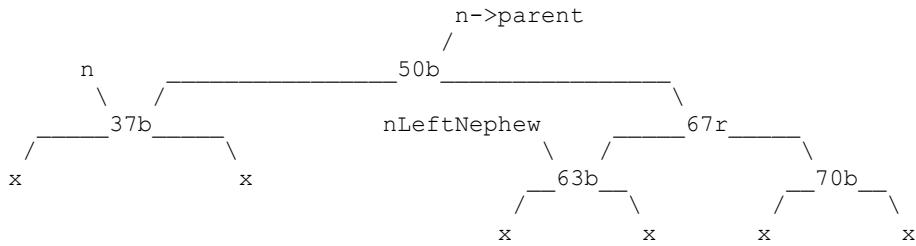
Case 2: n is a left child, and n 's parent is a right-leaning 3-node (lines 150-151)

n is a right child, and n 's parent is a left-leaning 3-node (lines 152-153)

To balance Case 2, we convert n 's 3-node parent from right to left-leaning, which transforms it into Case 4, 5, or 6. n 's left nephew, which becomes n 's sibling after the conversion, determines the type of transformation that will occur:

- If n 's left nephew is a 2-node, Case 2 will become Case 4
- If n 's left nephew is a left-leaning 3-node, Case 2 will become Case 5
- If n 's left nephew is either a 4-node or right-leaning 3-node, Case 2 will become Case 6

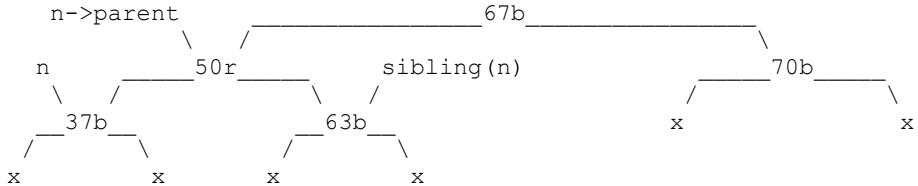
Let's take a look at each of these scenarios:



// Case 2: n (node 37) is a left child, and n's parent (node 50) is a
// right-leaning 3-node

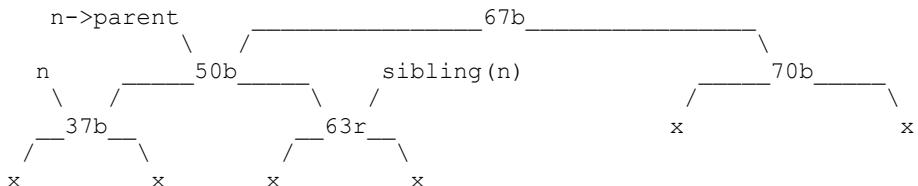
// n's left nephew (node 63) is a 2-node, so Case 2 will become Case 4
// after converting n's parent

`_convertToLeftLeaning3Node(n->parent);`

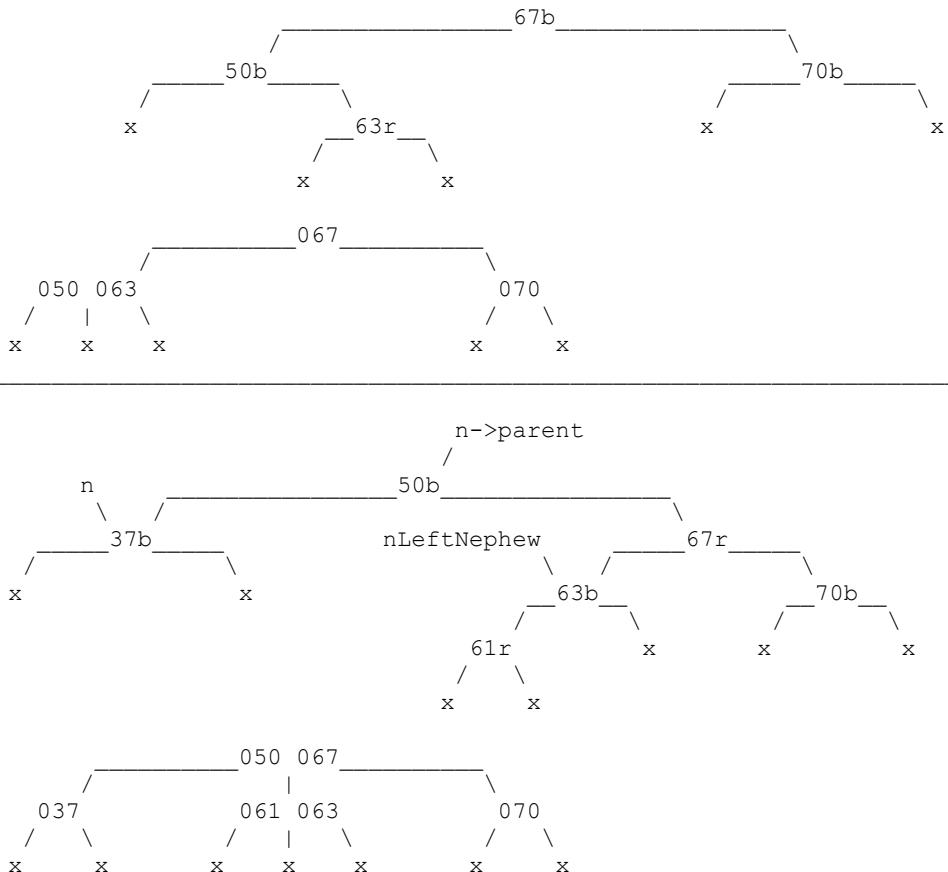


// Case 4: n's parent (node 50) is red, and n's sibling (node 63)
// is a 2-node

`n->parent->paintBlack();
sibling(n)->paintRed();`



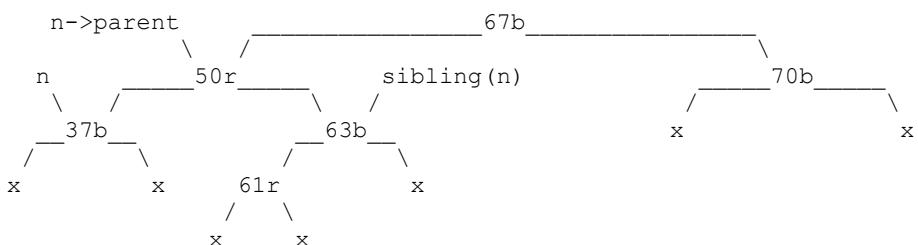
// remove n



// Case 2: n (node 37) is a left child, and n's parent (node 50) is a
// right-leaning 3-node

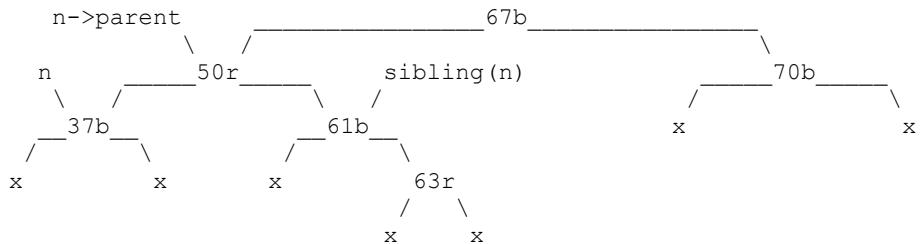
// n's left nephew (node 63) is a left-leaning 3-node, so Case 2 will
// become Case 5 after converting n's parent

_convertToLeftLeaning3Node(n->parent);



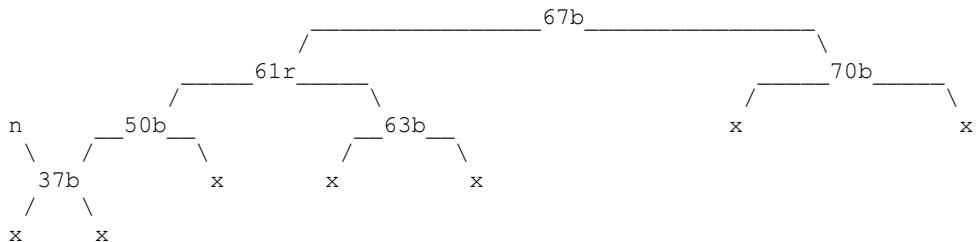
// Case 5: n (node 37) is a left child, and n's sibling (node 63)
// is a left-leaning 3-node

```
_convertToRightLeaning3Node(sibling(n));
```

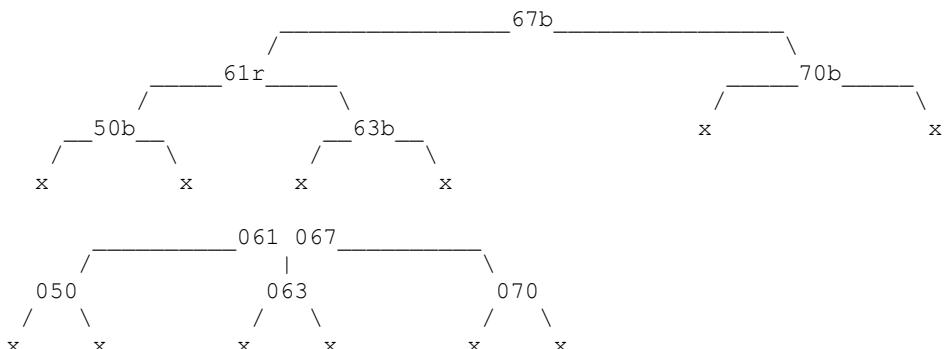


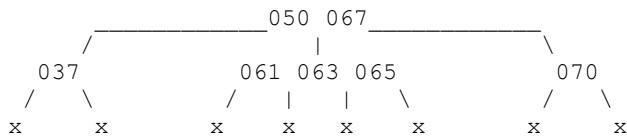
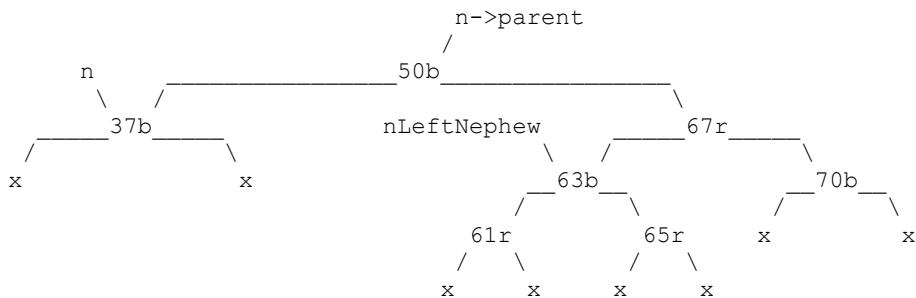
// Case 6: n (node 37) is a left child, and n's sibling (node 61) is
// either a 4-node or right-leaning 3-node

```
_performBTreeLeftRotation(sibling(n));
```



// remove n

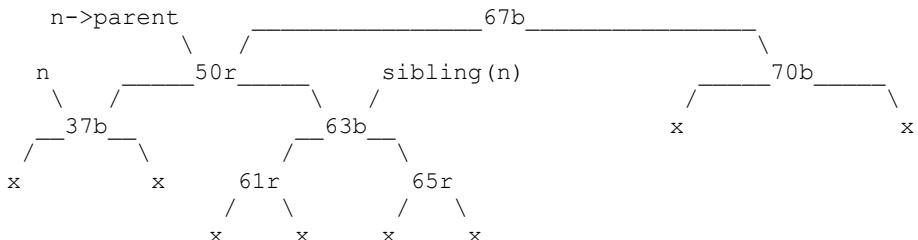




// Case 2: n (node 37) is a left child, and n's parent (node 50) is a right-leaning 3-node

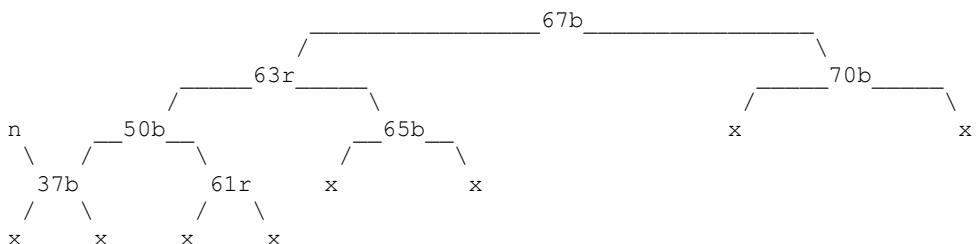
// n's left nephew (node 63) is either a 4-node or right-leaning 3-node, so Case 2 will become Case 6 after converting n's parent

convertToLeftLeaning3Node(n->parent);

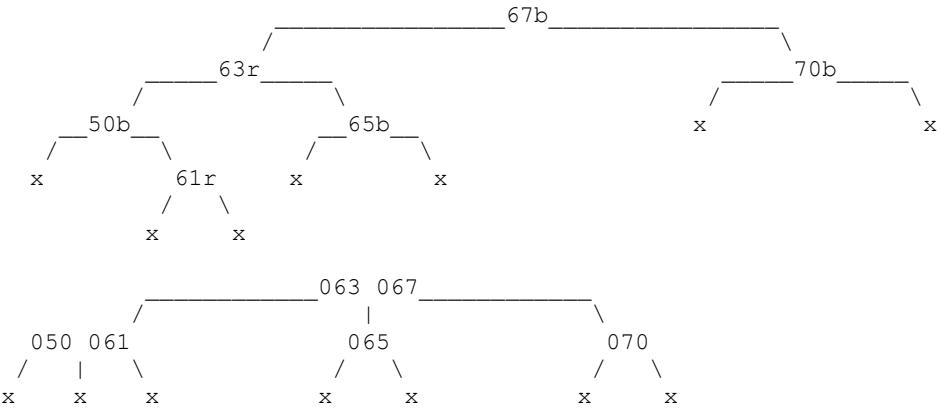


// Case 6: n (node 37) is a left child, and n's sibling (node 63) is either a 4-node or a right-leaning 3-node

performBTreeLeftRotation(sibling(n));



// remove n

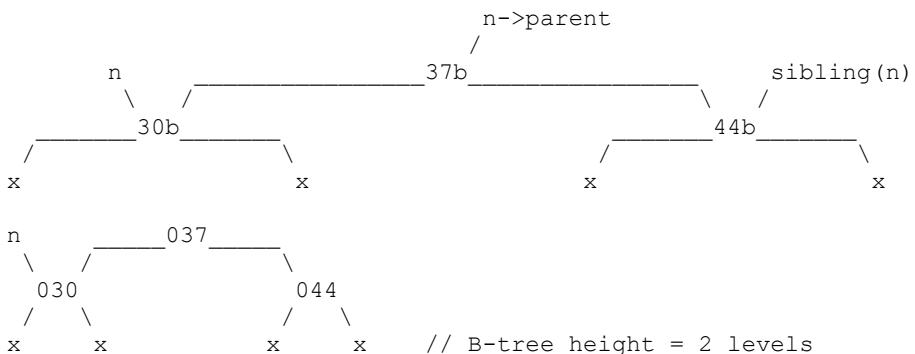


The mirror image of Case 2 occurs when n is a right child, and n 's parent is a left-leaning 3-node. To correct the violation, we convert n 's parent from left to right-leaning. n 's right nephew, which becomes n 's sibling after the conversion, determines the type of transformation that will occur:

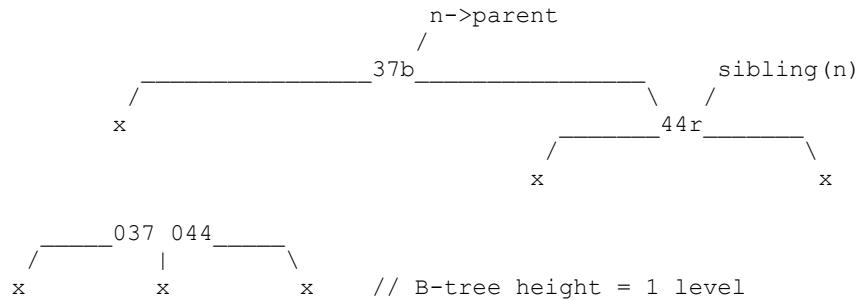
- If n 's right nephew is a 2-node, Case 2 will become Case 4
- If n 's right nephew is a right-leaning 3-node, Case 2 will become Case 5
- If n 's right nephew is either a 4-node or left-leaning 3-node, Case 2 will become Case 6

Case 3: n 's parent is black, and n 's sibling is a 2-node (lines 155-160)

If n 's parent is black and n 's sibling is a 2-node, then n 's parent must also be a 2-node, as in



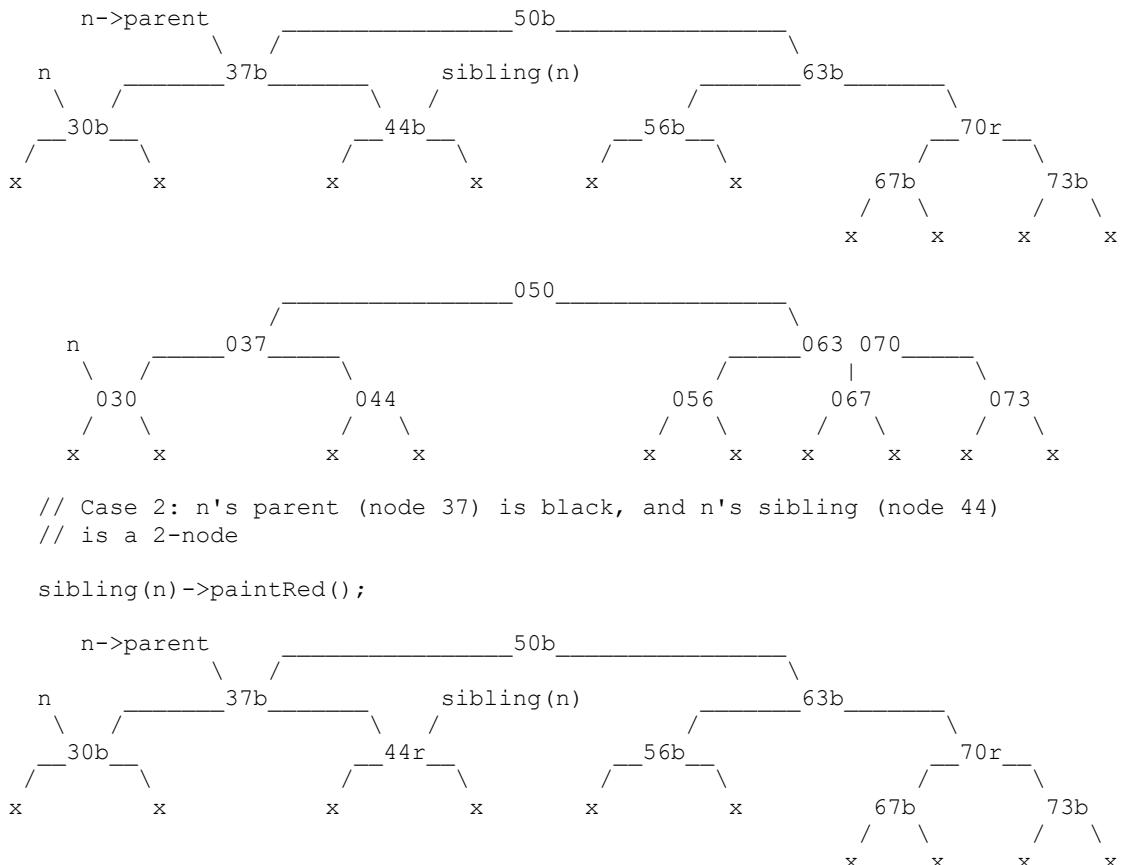
Removing n causes an underflow, which we correct by merging n 's parent and sibling elements into a single node. To perform the red-black equivalent of this operation, we simply paint n 's sibling red and remove n :



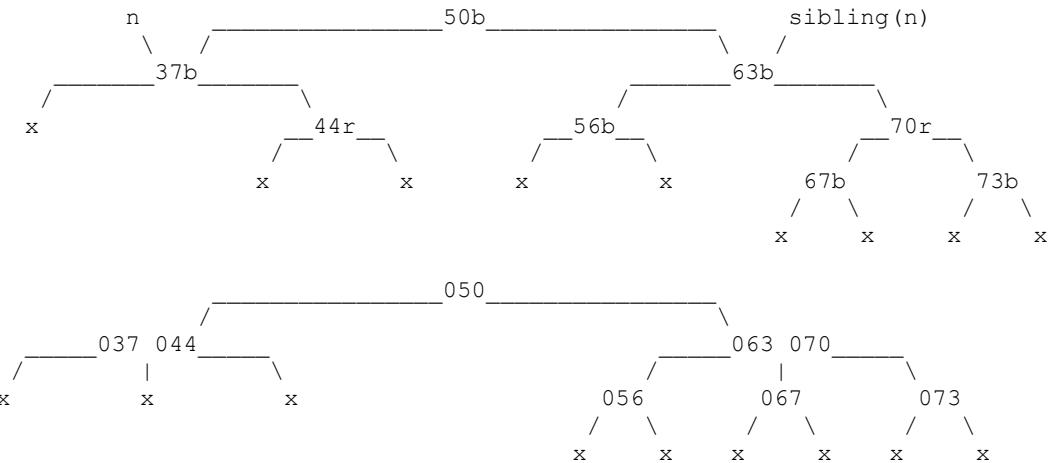
Note that this type of merge reduces the height of the original B-tree by 1 level.

If n 's parent is the root (as in the above example), then no additional balancing is required.

If, however, n 's parent is not the root, then the height reduction caused by the merge will create a new imbalance (Case 2, 3, 4, 5, or 6) farther up the tree. We must therefore proceed to n 's parent and correct the applicable violation. Consider, for example, the tree



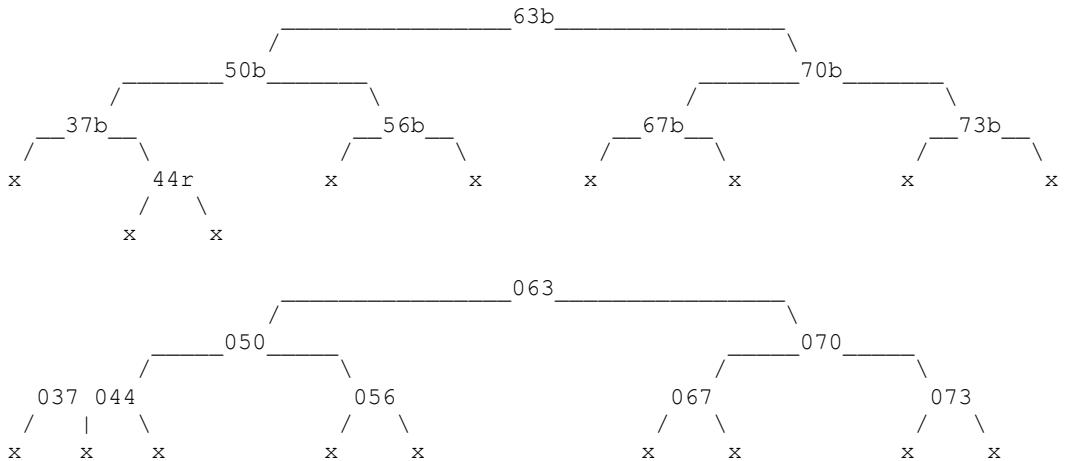
```
// remove n, proceed to n's parent (node 37)
```



```
// the subtree rooted at node {37,44} is now 1 level shorter than
// the subtree rooted at its sibling, node {63,70}
```

```
// Case 6: n (node 37) is a left child, and n's sibling (node 63)
// is either a 4-node or right-leaning 3-node
```

```
_performBTreeLeftRotation(sibling(n));
```



```
// the left and right subtrees, rooted at nodes {50} and {70}, are now
// the same height
```

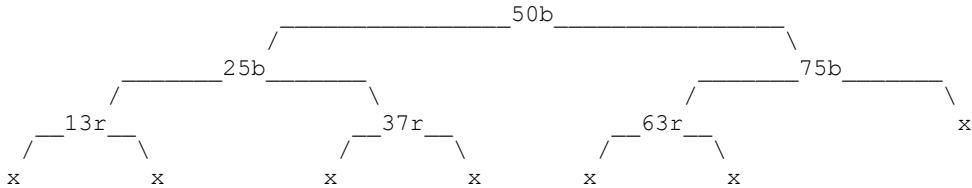
Putting it all together, our test program (*main.cpp*) begins by constructing a tree *r* and *printNode* function object (lines 20-23). In each iteration of the loop,

- We prompt the user to enter a single-character command (lines 27, 65-72)

- If the user entered *i* (for insert), we prompt them for the desired key value and insert it into the tree (lines 29-32, 74-81)
- If the user entered *e* (for erase), we prompt them for the desired key value, and if found, remove it from the tree (lines 33-40)
- If the user entered *p* (for print), we print the structure of the entire tree (lines 41-45)
- If the user entered *c* (for clear), we clear the tree (lines 46-49)
- If the user entered *q* (for quit), we terminate the loop and exit *main* (lines 50-53)
- If the user entered an unrecognized command, we print the message “Invalid command” and begin the next iteration (lines 54-57)

The following sample run demonstrates various cases of *_balanceOn3Or4NodeErase* and *_balanceOn2NodeErase*:

```
// insert 50, 25, 75, 13, 37, 63
```

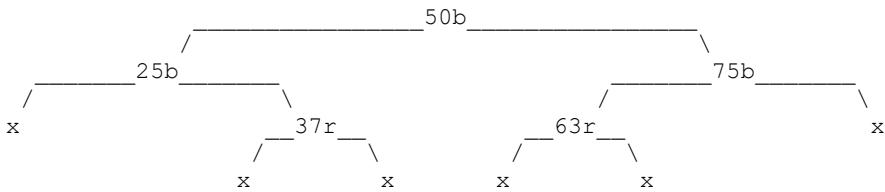


```

node 13
  parent 25
  red
node 25
  parent 50
  left 13
  right 37
  black
node 37
  parent 25
  red
node 50
  left 25
  right 75
  black
node 63
  parent 75
  red
node 75
  parent 50
  left 63
  black
  
```

```
// erase 13 (the red portion of a 3-node or 4-node)
```

```
// _balanceOn3Or4NodeErase, case 1
```



```

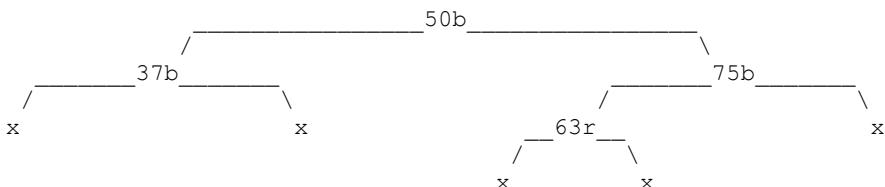
node 25
  parent 50
  right 37
  black
node 37
  parent 25
  red
node 50
  left 25
  right 75
  black
node 63
  parent 75
  red
node 75
  parent 50
  left 63
  black

```

```

// erase 25 (the black portion of a right-leaning 3-node)
// _balanceOn3Or4NodeErase, case 3

```

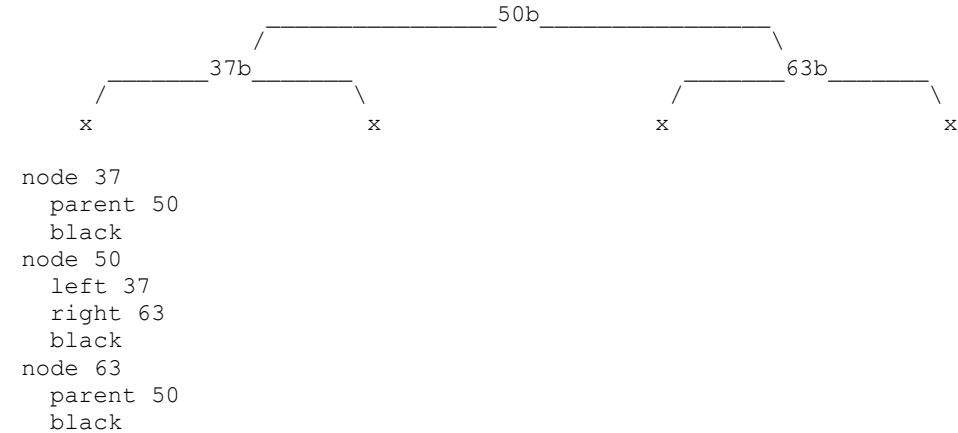


```

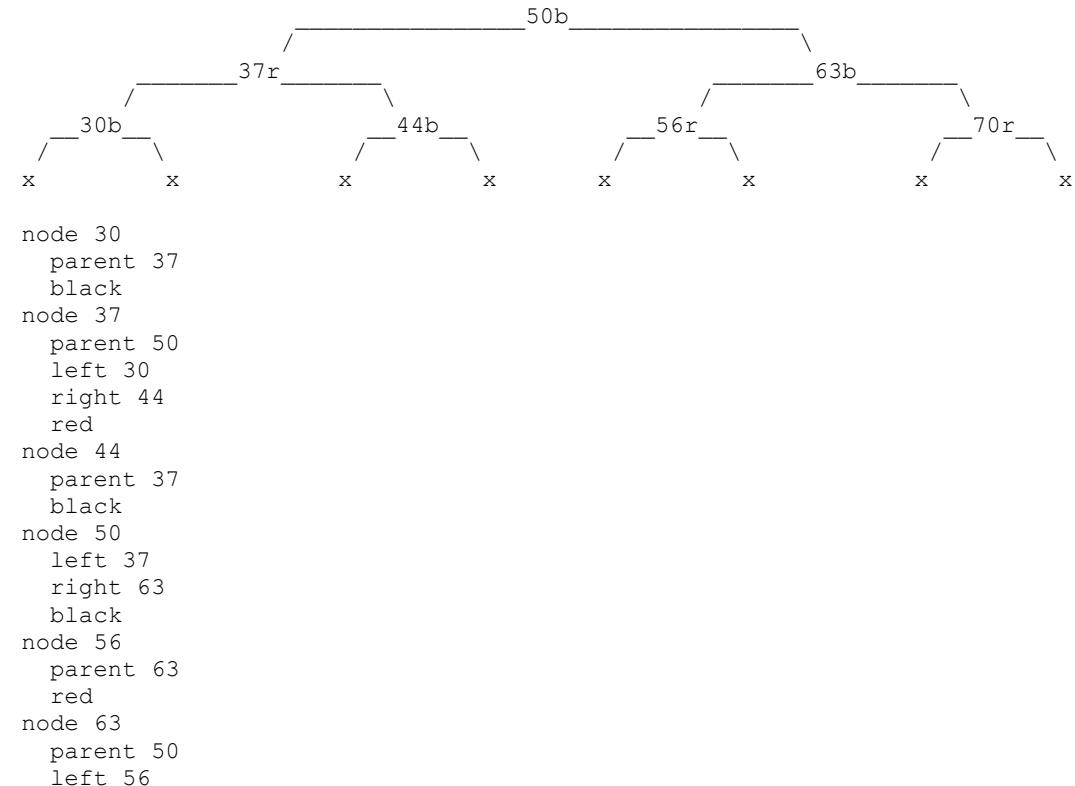
node 37
  parent 50
  black
node 50
  left 37
  right 75
  black
node 63
  parent 75
  red
node 75
  parent 50
  left 63
  black

```

```
// erase 75 (the black portion of a left-leaning 3-node)
// _balanceOn3Or4NodeErase, case 2
```



```
// insert 30, 44, 56, 70, 27
// erase 27
```



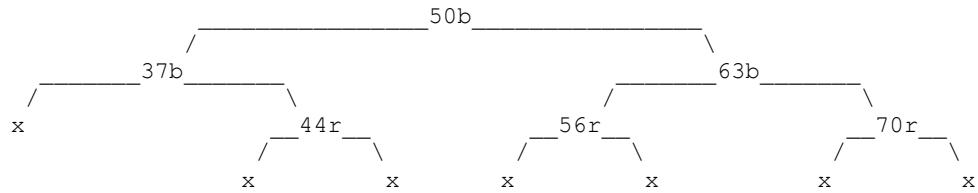
```

right 70
black
node 70
parent 63
red

// erase 30 (a 2-node whose parent is the red portion of a 3-node or 4-node,
// and whose sibling is a 2-node)

// _balanceOn2NodeErase, case 4

```

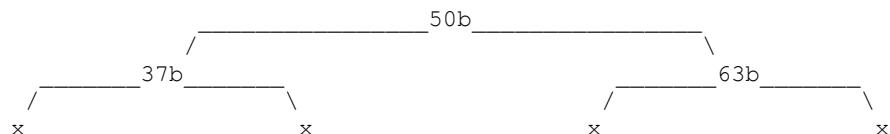


```

node 37
parent 50
right 44
black
node 44
parent 37
red
node 50
left 37
right 63
black
node 56
parent 63
red
node 63
parent 50
left 56
right 70
black
node 70
parent 63
red

```

```
// erase 44, 56, 70
```



```

node 37
parent 50
black

```

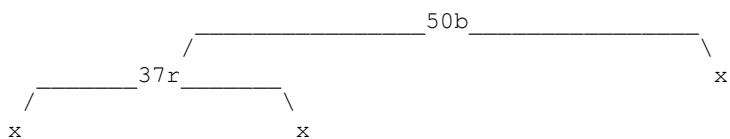
```

node 50
  left 37
  right 63
  black
node 63
  parent 50
  black

// erase 63 (a 2-node whose parent is a 2-node, and whose sibling is
// a 2-node)

// balanceOn2NodeErase, case 3 -> case 1

```



```

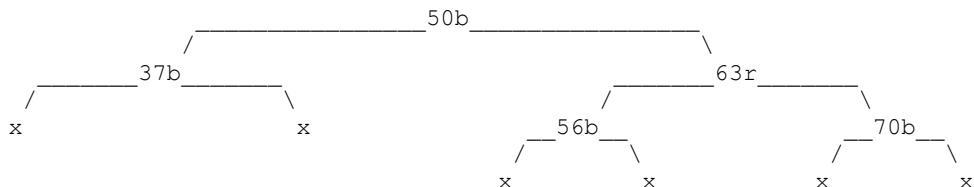
node 37
  parent 50
  red
node 50
  left 37
  black

```

```

// insert 63, 30, 44, 56, 70, 53
// erase 30, 44, 53

```



```

node 37
  parent 50
  black
node 50
  left 37
  right 63
  black
node 56
  parent 63
  black
node 63
  parent 50
  left 56
  right 70
  red
node 70

```

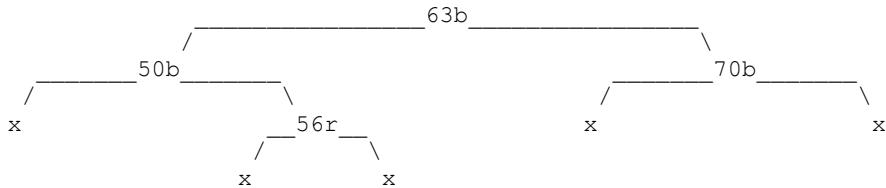
```

parent 63
black

// erase 37 (a 2-node that is a left child, and whose parent is the black
// portion of a right-leaning 3-node)

// _balanceOn2NodeErase, case 2 -> case 4

```



```

node 50
parent 63
right 56
black
node 56
parent 50
red
node 63
left 50
right 70
black
node 70
parent 63
black

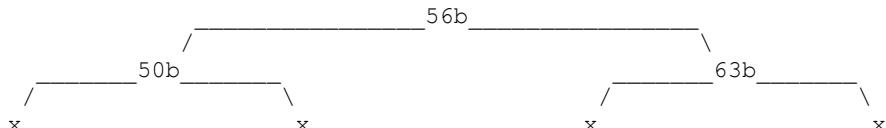
```

```

// erase 70 (a 2-node that is a right child, and whose sibling is a
// right-leaning 3-node)

```

```
// _balanceOn2NodeErase, case 5 -> case 6
```



```

node 50
parent 56
black
node 56
left 50
right 63
black
node 63
parent 56
black

```


Part 8: Skip Lists

8.1: Introducing the *SkipList* Class

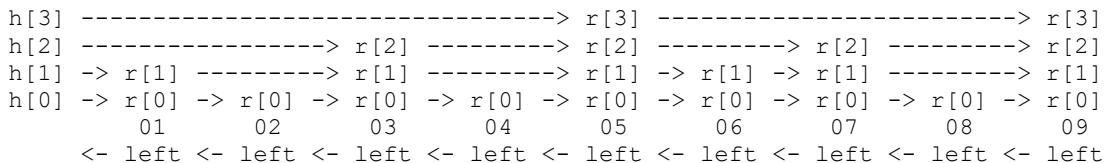
Source files and folders

- *printSkipList*
- *SkipList/l*
- *SkipList/common/memberFunctions_1.h*
- *SkipList/common/SkipListNode.h*
- *SkipList/insert/insert_1.h*

Chapter outline

- *Overview and terminology*
- *Implementing nodes, basic accessor methods, insert, clear, and test tools*

Skip lists were invented by William Pugh as a simpler alternative to balanced trees, offering comparable (logarithmic-time) performance. A skip list, like a balanced tree, is an associative data structure in which the elements are sorted using a predicate. Internally, however, a skip list more closely resembles a linked list. Each node contains a single element (key-mapped pair), a pointer to its left neighbor, and a vector of pointers to its right neighbors:



The above list contains the key values $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. As usual, we'll refer to each specific node by its key value: "node 5" is the node containing the key value 5, "node 6" is the node containing the key value 6, etc.

Each node's *left* link points to its left neighbor, which lets us traverse the list backwards: node 9's left link points to node 8, node 8's left link points to node 7, etc.

Each node also contains 1 or more *right* links, stored in a vector (*r* in the above diagram). The *height* of a node is the number of right links it contains (the size of its vector). Each right link is located on a different *level* (vector index), where level 0 is the *bottom level* and the *top level* is (*height* – 1). Node 5 has a height of 4 because it has right links at levels 0, 1, 2, and 3 (*r*[0], *r*[1], *r*[2], and *r*[3]).

Each right link at level *i* points to the next node at level *i*. In node 5, for example,

- $r[0]$ points to the next node at level 0 (node 6)
- $r[1]$ points to the next node at level 1 (node 6)
- $r[2]$ points to the next node at level 2 (node 7)
- $r[3]$ points to the next node at level 3 (node 9)

Similarly, in node 3,

- $r[0]$ points to the next node at level 0 (node 4)
- $r[1]$ points to the next node at level 1 (node 5)
- $r[2]$ points to the next node at level 2 (node 5)

The *head* (h in the above diagram) is a vector of pointers to the first node at each level:

- $h[0]$ points to the first node at level 0 (node 1)
- $h[1]$ points to the first node at level 1 (node 1)
- $h[2]$ points to the first node at level 2 (node 3)
- $h[3]$ points to the first node at level 3 (node 5)

The *tail* (node 9 in the above diagram) is the rightmost node in the list (the node containing the *back* element). The tail's right links are always null pointers.

To search the list, we begin at the top level and work our way down. The gaps at each level allow us to skip over consecutive elements (hence the name “skip list”), thereby reducing the number of nodes visited.

```

h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0]
      01      02      03      04      05      06      07      08      09
      <- left <- left
  
```

To find the key value 7, for example:

- We begin at level 3. The head link $h[3]$ takes us to node 5. Node 5's key value is less than 7, so we take the right link $r[3]$ to node 9.
- Node 9's key value is not less than 7, so we go down to level 2 of the previous node (5), and take the right link $r[2]$ to node 7.
- Node 7's key value is 7, so the search is complete. We visited 3 nodes (5, 9, and 7); in a traditional linked list we would've visited 7 nodes.

To find the key value 3:

- We begin at level 3. The head link $h[3]$ takes us to node 5. Node 5's key value is not less than 3, so we go down to level 2 of the previous node (the head), and take the right link $h[2]$ to

node 3 (As described above, the head isn't technically a node, but we're calling it a node here for the sake of simplicity).

- Node 3's key value is 3, so the search is complete. We visited 2 nodes (5 and 3); in a traditional linked list we would've visited 3 nodes.

A skip list isn't always faster than a traditional linked list – in fact, it can even be slower in some cases – but its average time complexity is still logarithmic. Finding the key value 2, for example, requires twice as many visits as a traditional linked list:

- We begin at level 3. The head link $h[3]$ takes us to node 5. Node 5's key value is not less than 2, so we go down to level 2 of the previous node (the head), and take the right link $h[2]$ to node 3.
- Node 3's key value is not less than 2, so we go down to level 1 of the previous node (the head), and take the right link $h[1]$ to node 1.
- Node 1's key value is less than 2, so we take the right link $r[1]$ to node 3.
- Node 3's key value is not less than 2, so we go down to level 0 of the previous node (1), and take the right link $r[0]$ to node 2.
- Node 2's key value is 2, so the search is complete. We visited 4 nodes (5, 3, 1, and 2); in a traditional linked list we would've visited 2 nodes.

A skip list is a type of *probabilistic* data structure because the height of each node is chosen somewhat randomly. The only requirement is that the node heights follow a particular distribution:

Top Level	Capacity = $2^{(\text{Top Level} + 1) - 1}$	$L_i = 2^{(\text{Top Level} - i)}$	L0	L1	L2	L3	L4	L5	L6
0	1	1							
1	3	2 1							
2	7	4 2 1							
3	15	8 4 2 1							
4	31	16 8 4 2 1							
5	63	32 16 8 4 2 1							
6	127	64 32 16 8 4 2 1							

The *top level* of a skip list is the highest level in the entire list (the top level of the head). The *capacity* is the total number of elements (nodes) that the list can hold, while maintaining the proper distribution of node heights. In the above table,

- L0 is the maximum allowable number of *level 0 nodes* (nodes whose top level is 0)
- L1 is the maximum allowable number of *level 1 nodes* (nodes whose top level is 1)
- L2 is the maximum allowable number of *level 2 nodes* (nodes whose top level is 2), etc.

As shown in the table, the top level determines both the capacity and the maximum allowable number of nodes of each height. We must therefore choose the top level before inserting any elements.

For a top level of 0, for example:

- Capacity $= 2^{(0+1)} - 1$ = 1 element
- L0 $= 2^{(0-0)}$ = 1 level 0 node

For a top level of 1:

- Capacity $= 2^{(1+1)} - 1$ = 3 elements (2 + 1)
- L0 $= 2^{(1-0)}$ = 2 level 0 nodes
- L1 $= 2^{(1-1)}$ = 1 level 1 node

For a top level of 2:

- Capacity $= 2^{(2+1)} - 1$ = 7 elements (4 + 2 + 1)
- L0 $= 2^{(2-0)}$ = 4 level 0 nodes
- L1 $= 2^{(2-1)}$ = 2 level 1 nodes
- L2 $= 2^{(2-2)}$ = 1 level 2 node

For a top level of 3:

- Capacity $= 2^{(3+1)} - 1$ = 15 elements (8 + 4 + 2 + 1)
- L0 $= 2^{(3-0)}$ = 8 level 0 nodes
- L1 $= 2^{(3-1)}$ = 4 level 1 nodes
- L2 $= 2^{(3-2)}$ = 2 level 2 nodes
- L3 $= 2^{(3-3)}$ = 1 level 3 node

Each skip list contains a vector, *availableNodes*, to keep track of the number of available nodes of each level. Upon constructing a list with a top level of 3, for example,

- *availableNodes[0]* = 8 (8 available level 0 nodes)
- *availableNodes[1]* = 4 (4 available level 1 nodes)
- *availableNodes[2]* = 2 (2 available level 2 nodes)
- *availableNodes[3]* = 1 (1 available level 3 nodes)

When inserting an element, we randomly choose the new node's top level (0, 1, 2, or 3) and decrement *availableNodes* accordingly. If we begin by inserting a level 1 node, for example, the table becomes

- *availableNodes[0]* = 8
- *availableNodes[1]* = 3 (1 less available level 1 node)
- *availableNodes[2]* = 2
- *availableNodes[3]* = 1

If we then insert a level 0 node, the table becomes

- *availableNodes[0] = 7* (1 less available level 0 node)
- *availableNodes[1] = 3*
- *availableNodes[2] = 2*
- *availableNodes[3] = 1*

Conversely, when erasing an element, we increment *availableNodes*. If, for example, we erase the first element (the level 1 node), the table becomes

- *availableNodes[0] = 7*
- *availableNodes[1] = 4* (1 more available level 1 node)
- *availableNodes[2] = 2*
- *availableNodes[3] = 1*

When the list becomes full, we must increase the capacity before inserting any additional elements. We'll examine that procedure later on, but for now let's begin by creating the node class (*SkipListNode.h*). The 3 data members are:

- *element*, a *pair<const Key, Mapped>* (lines 13, 24)
- *left*, a pointer to the node's left neighbor (lines 14, 25)
- *right*, a *NodeVector* (vector of pointers to the node's right neighbors) (lines 15, 26)

Level (line 16) is an alias of *size_t* (vector index value). The constructor (line 18),

```
SkipListNode(Level topLevel, const value_type& element);
```

creates a node with the given *topLevel* (highest level), containing the given *element* (lines 29-37):

- a node with a *topLevel* of 0 has a height of 1 (level 0)
- a node with a *topLevel* of 1 has a height of 2 (levels 0 and 1)
- a node with a *topLevel* of 2 has a height of 3 (levels 0, 1, and 2), etc.

The *topLevel* member function (line 20) simply returns the node's top level (lines 39-44).

The function (line 22)

```
void setTopLevel(Level newTopLevel);
```

changes the node's height such that it has the *newTopLevel* (lines 46-50). Given the node

```
r[0] -> // topLevel = 0 (height = 1)
          08
<- left
```

for example, if we *setTopLevel* to 2, it increases the height to 3 by inserting new pointers at levels 1 and 2:

```
r[2]      // topLevel = 2 (height = 3)
r[1]
r[0] ->
  08
<- left
```

Similarly, given the node

```
r[3] ->    // topLevel = 3 (height = 4)
r[2] ->
r[1] ->
r[0] ->
  05
<- left
```

if we *setTopLevel* to 1, it decreases the height to 2 by removing the pointers at levels 3 and 2:

```
r[1] ->    // topLevel = 1 (height = 2)
r[0] ->
  05
<- left
```

The *SkipList* class (*SkipList.h*) has 6 data members (lines 28-32, 81-86):

- *_head*, a vector of pointers to the first (leftmost) node at each level (*_head[0]* points to the first node at level 0, *_head[1]* points to the first node at level 1, etc.)
- *_tail*, a pointer to the rightmost node in the list (the node containing the *back* element)
- *_size*, the total number of elements
- *_availableNodes*, a vector of the number of available nodes of each level
- *_predicate*, the function object used to compare key values
- *_alloc*, an allocator used to create and destroy nodes

Before moving on to the constructor and *insert* function, let's write some of the simpler methods (*SkipList.h*, lines 37-48 / *memberFunctions_I.h*, lines 24-98):

- *empty*: Returns *true* if *_size* is 0
- *size*: Returns the *_size*
- *capacity*: Returns the capacity, using the formula described earlier
(Capacity = $2^{(\text{Top level} + 1)} - 1$)
- *front*: Returns a (*const*) reference to the first (leftmost) element in the list
- *back*: Returns a (*const*) reference to the last (rightmost) element in the list
- *key_comp*: Returns the *_predicate*
- *head*: Returns a *const* reference to the head (used for testing)
- *topLevel*: Returns the highest level in the entire list (the top level of the *_head*)
- *availableNodes*: Returns a *const* reference to the *_availableNodes* table (used for testing)

The static constant *_initialTopLevel* (*SkipList.h*, line 79) is the initial top level of the *_head* (3).

The default constructor (line 34) constructs the `_head` with 4 levels (0, 1, 2, and 3), initializes the `_tail` to null, and the `_size` to 0 (*memberFunctions_1.h*, lines 11-13). It then calls the private member function (*SkipList.h*, line 64)

```
void _initAvailableNodeCount();
```

which initializes the `_availableNodes` table. We first resize the table to the size of the `_head` (4 levels), then populate it using the formula described earlier (*memberFunctions_1.h*, lines 123-131):

```
for each Level i,  
    _availableNodes[i] = 2^(topLevel - i);
```

Now that the list is initialized, we'll need functions to create and destroy nodes:

- `_createNode` (*SkipList.h*, line 75): Creates a new node with the given `topLevel` and `element`, then returns a pointer to the new node (*memberFunctions_1.h*, lines 222-231)
- `_destroyNode` (*SkipList.h*, line 76): Destroys the given node `n` and deallocates the corresponding memory block (*memberFunctions_1.h*, lines 233-238)
- `_destroyAllNodes` (*SkipList.h*, line 77): Destroys each node in the list, beginning at the front (`_head[0]`) (*memberFunctions_1.h*, lines 240-251)

The destructor (*SkipList.h*, line 35) simply calls `_destroyAllNodes` (*memberFunctions_1.h*, lines 18-22).

We can now implement the first version of `insert` (*SkipList.h*, line 49),

```
void insert(const value_type& newElement);
```

We'll update its return type to `pair<iterator, bool>` later on, after implementing the iterator class. But for now, the procedure is (*insert_1.h*, lines 3-23)

```
void insert(const value_type& newElement)  
{  
    if the list is full,                                // lines 6-7  
        return;  
  
    find the insertion point (location) for the new key; // lines 9-14  
  
    if the new key already exists,                      // lines 16-17  
        return;  
  
    create a node with a randomly chosen top level,      // line 19  
        containing the newElement;  
  
    attach the new node to its neighbors;                // lines 21-22  
    update the _size;  
}
```

As mentioned earlier, if the list is full, we need to increase the capacity before inserting any new

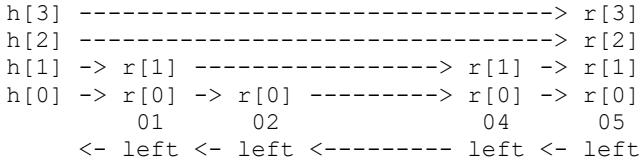
elements. We'll implement that functionality later on, but for now we'll simply abort the procedure (lines 6-7).

Assuming that we haven't reached full capacity, the first step to inserting a new element is to search for its key value as if it already existed in the list. This will give us the proper location (*insertion point*) of the new element (node).

While searching for the insertion point, however, we also need to keep track of 3 additional things:

- The right link at each level that will point to the new node:
 - To save these links, we'll use a *LinkVector* (*vector<Node**>*) (*SkipList.h*, line 53), called *rightLinks* (*insert_1.h*, line 9). Each *rightLink* is a pointer to a right link (*Node**), which will let us update the right link to point to the new node.
 - While searching for the insertion point, before going right, we'll store a pointer to the current right link in the corresponding level of *rightLinks*.
- The node at each level that will become the right neighbor of the new node:
 - To save these nodes, we'll use a *NodeVector* (*vector<Node*>*) (*SkipList.h*, line 29), called *rightNeighbors* (*insert_1.h*, line 10). After creating the new node, we'll set each of its right links to point to the corresponding *rightNeighbor*.
 - While searching for the insertion point, before going down a level, we'll store a pointer to the current node in the corresponding level of *rightNeighbors*.
- The key value of each visited node:
 - If the key value of the current node matches the new key, we won't insert the new element because duplicate keys aren't allowed. To indicate whether or not the new key already exists, we'll use a *NewKeySearch* object (*SkipList.h*, lines 56-62 / *memberFunctions_1.h*, lines 113-121), which contains:
 - *newKeyAlreadyExists*, a boolean flag that indicates whether the new key value already exists
 - *node*, a pointer to the node already containing the new key value (if *newKeyAlreadyExists* is *true*)

Consider, for example, the list



and suppose that the *newElement*'s key value is 3.

We begin at level 3 of the *_head*. Before going right (to node 5), we save a pointer to h[3] in rightLinks[3]:

```
rightLinks[3] = &h[3]
```

Node 5's key value is not less than 3. Before going down (to level 2), we save a pointer to node 5 in rightNeighbors[3]:

```
rightLinks[3] = &h[3]           rightNeighbors[3] = &(node 5)
```

At level 2 of the *_head*, before going right (to node 5), we save a pointer to h[2] in rightLinks[2]:

```
rightLinks[3] = &h[3]           rightNeighbors[3] = &(node 5)
rightLinks[2] = &h[2]
```

Node 5's key value is not less than 3. Before going down (to level 1), we save a pointer to node 5 in rightNeighbors[2]:

```
rightLinks[3] = &h[3]           rightNeighbors[3] = &(node 5)
rightLinks[2] = &h[2]           rightNeighbors[2] = &(node 5)
```

At level 1 of the *_head*, before going right (to node 1), we save a pointer to h[1] in rightLinks[1]:

```
rightLinks[3] = &h[3]           rightNeighbors[3] = &(node 5)
rightLinks[2] = &h[2]           rightNeighbors[2] = &(node 5)
rightLinks[1] = &h[1]
```

Node 1's key value is less than 3. Before going right (to node 4), we save a pointer to r[1] in rightLinks[1]:

```
rightLinks[3] = &h[3]           rightNeighbors[3] = &(node 5)
rightLinks[2] = &h[2]           rightNeighbors[2] = &(node 5)
rightLinks[1] = &(node 1, r[1])
```

Node 4's key value is not less than 3. Before going down (to level 0), we save a pointer to node 4 in rightNeighbors[1]:

```
rightLinks[3] = &h[3]           rightNeighbors[3] = &(node 5)
rightLinks[2] = &h[2]           rightNeighbors[2] = &(node 5)
rightLinks[1] = &(node 1, r[1]) rightNeighbors[1] = &(node 4)
```

At level 0 of the `_head`, before going right (to node 1), we save a pointer to `h[0]` in `rightLinks[0]`:

```
rightLinks[3] = &h[3]           rightNeighbors[3] = &(node 5)
rightLinks[2] = &h[2]           rightNeighbors[2] = &(node 5)
rightLinks[1] = &(node 1, r[1]) rightNeighbors[1] = &(node 4)
rightLinks[0] = &h[0]
```

Node 1's key value is less than 3. Before going right (to node 2), we save a pointer to `r[0]` in `rightLinks[0]`:

```
rightLinks[3] = &h[3]           rightNeighbors[3] = &(node 5)
rightLinks[2] = &h[2]           rightNeighbors[2] = &(node 5)
rightLinks[1] = &(node 1, r[1]) rightNeighbors[1] = &(node 4)
rightLinks[0] = &(node 1, r[0])
```

Node 2's key value is less than 3. Before going right (to node 4), we save a pointer to `r[0]` in `rightLinks[0]`:

```
rightLinks[3] = &h[3]           rightNeighbors[3] = &(node 5)
rightLinks[2] = &h[2]           rightNeighbors[2] = &(node 5)
rightLinks[1] = &(node 1, r[1]) rightNeighbors[1] = &(node 4)
rightLinks[0] = &(node 2, r[0])
```

Node 4's key value is not less than 3. Before going down (to level -1), we save a pointer to node 4 in `rightNeighbors[0]`:

```
rightLinks[3] = &h[3]           rightNeighbors[3] = &(node 5)
rightLinks[2] = &h[2]           rightNeighbors[2] = &(node 5)
rightLinks[1] = &(node 1, r[1]) rightNeighbors[1] = &(node 4)
rightLinks[0] = &(node 2, r[0]) rightNeighbors[0] = &(node 4)
```

We've dropped below level 0, so the search is complete. We can now create node 3 (the new node) and attach it to the list by updating the appropriate `rightLinks` (to point to node 3) and setting each of node 3's own right links to point to the corresponding `rightNeighbor` at each level.

We'll write the `_attachNewNode` method later; for now, let's walk through another example before implementing the search function. Given the list

```
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
      01      02      03      04      05      06          08      09
      <- left <- left
```

suppose that the `newElement`'s key value is 7.

We begin at level 3 of the `_head`. Before going right (to node 5), we save a pointer to `h[3]` in `rightLinks[3]`:

```
rightLinks[3] = &h[3]
```

Node 5's key value is less than 7. Before going right (to node 9), we save a pointer to r[3] in rightLinks[3]:

```
rightLinks[3] = &(node 5, r[3])
```

Node 9's key value is not less than 7. Before going down (to level 2), we save a pointer to node 9 in rightNeighbors[3]:

```
rightLinks[3] = &(node 5, r[3])      rightNeighbors[3] = &(node 9)
```

At level 2 of node 5, before going right (to node 9), we save a pointer to r[2] in rightLinks[2]:

```
rightLinks[3] = &(node 5, r[3])      rightNeighbors[3] = &(node 9)
rightLinks[2] = &(node 5, r[2])
```

Node 9's key value is not less than 7. Before going down (to level 1), we save a pointer to node 9 in rightNeighbors[2]:

```
rightLinks[3] = &(node 5, r[3])      rightNeighbors[3] = &(node 9)
rightLinks[2] = &(node 5, r[2])      rightNeighbors[2] = &(node 9)
```

At level 1 of node 5, before going right (to node 6), we save a pointer to r[1] in rightLinks[1]:

```
rightLinks[3] = &(node 5, r[3])      rightNeighbors[3] = &(node 9)
rightLinks[2] = &(node 5, r[2])      rightNeighbors[2] = &(node 9)
rightLinks[1] = &(node 5, r[1])
```

Node 6's key value is less than 7. Before going right (to node 9), we save a pointer to r[1] in rightLinks[1]:

```
rightLinks[3] = &(node 5, r[3])      rightNeighbors[3] = &(node 9)
rightLinks[2] = &(node 5, r[2])      rightNeighbors[2] = &(node 9)
rightLinks[1] = &(node 6, r[1])
```

Node 9's key value is not less than 7. Before going down (to level 0), we save a pointer to node 9 in rightNeighbors[1]:

```
rightLinks[3] = &(node 5, r[3])      rightNeighbors[3] = &(node 9)
rightLinks[2] = &(node 5, r[2])      rightNeighbors[2] = &(node 9)
rightLinks[1] = &(node 6, r[1])      rightNeighbors[1] = &(node 9)
```

At level 0 of node 6, before going right (to node 8), we save a pointer to r[0] in rightLinks[0]:

```
rightLinks[3] = &(node 5, r[3])      rightNeighbors[3] = &(node 9)
rightLinks[2] = &(node 5, r[2])      rightNeighbors[2] = &(node 9)
rightLinks[1] = &(node 6, r[1])      rightNeighbors[1] = &(node 9)
rightLinks[0] = &(node 6, r[0])
```

Node 8's key value is not less than 7. Before going down (to level -1), we save a pointer to node 8 in `rightNeighbors[0]`:

```
rightLinks[3] = &(node 5, r[3])      rightNeighbors[3] = &(node 9)
rightLinks[2] = &(node 5, r[2])      rightNeighbors[2] = &(node 9)
rightLinks[1] = &(node 6, r[1])      rightNeighbors[1] = &(node 9)
rightLinks[0] = &(node 6, r[0])      rightNeighbors[0] = &(node 8)
```

We've dropped below level 0, so the search is complete. We can now create node 7 (the new node) and attach it to the list by updating the appropriate `rightLinks` (to point to node 7) and setting each of node 7's own right links to point to the corresponding `rightNeighbor` at each level.

The function (*SkipList.h*, lines 67-69)

```
NewKeySearch _findInsertionPointForNewKey(const key_type& newKey,
LinkVector& rightLinks,
NodeVector& rightSiblings);
```

finds the insertion point (proper location) for the `newKey`. In `rightLinks`, it saves pointers to the right links that will need to be updated to point to the new node. In `rightSiblings`, it saves pointers to the nodes that will become the right siblings of the new node. While finding the insertion point, it also checks each visited node to see if the `newKey` already exists. When the search is complete, the function returns a `NewKeySearch` object, containing

- `newKeyAlreadyExists`, a boolean flag that indicates whether the `newKey` already exists
- `node`, a pointer to the node already containing the `newKey` (if `newKeyAlreadyExists` is *true*)

We begin by resizing `rightLinks` and `rightSiblings` to the height of the `_head` (*memberFunctions_1.h*, lines 164-165).

We then initialize the `currentLevel` to the top level of the `_head` (line 167). To keep track of the `currentLevel`, we need to use a `SignedLevel` (plain `int`, as opposed to and `unsigned Level`) (*SkipList.h*, line 54) because (as shown in the above examples) a `currentLevel` of -1 indicates that the search is complete.

The current node `n` is initialized to the leftmost node at the top level (line 168), and the current `NodeVector` (vector of right links) `v` is initialized to the `_head` (line 169).

In each iteration of the loop:

- We first determine whether the current node's key value matches the `newKey`. If it does, we immediately terminate the search, returning a result of `newKeyAlreadyExists true`, along with the current node (which already contains the `newKey`) (lines 173-175).
- If the current node's key value is less than the `newKey`, we go right: `v` becomes the current node's `NodeVector`, and `n` becomes the current node's right neighbor at the current level. We

then begin the next iteration (lines 177-183).

- If the current node's key value is not less than the *newKey* (or the current node is null because we've reached the end of the list), we go down to the next level. But before doing so, we save a pointer to the right link at the current level, as well as a pointer to the current node, in *rightLinks* and *rightNeighbors* respectively (lines 185-188).
- If we've dropped below level 0 (to level -1), the search is complete. *rightLinks* and *rightNeighbors* contain pointers to all of the appropriate links and nodes. We return a result of *newKeyAlreadyExists* *false*, along with a null pointer (lines 190-191, 196).
- If we haven't dropped below level 0, we prepare for the next iteration: the current node becomes the current *NodeVector*'s right neighbor at the next level (lines 188, 193).

Let's walk through this code, using a slight variation of the last example:

```

h[5] -> null
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
      01      02      03      04      05      06      08      09
      <- left <- left

```

The only difference here is that the top level of the *_head* is higher than the top level of the tallest nodes (5 and 9), so *h[4]* and *h[5]* are both null pointers. Our *newKey*, like last time, will be 7. In the following diagrams, the *left* links are omitted because they aren't relevant to the code.

```

currentLevel = 5;
n = _head[currentLevel] = _head[5] = nullptr;
v = &_head;

h[5] -> null <- n
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
      01      02      03      04      05      06      08      09
      v

```

Iteration 1:

```
// n is null, so save the current right link and right neighbor
```

```
rightLinks[5]      = &(*v)[5] = &_head[5];
rightNeighbors[5] = n        = nullptr;
```

```

// go down to the next level

--currentLevel;      // currentLevel = 4;

// n becomes v's right neighbor at the next level

n = (*v)[currentLevel] = _head[4] = nullptr;

h[5] -> null
h[4] -> null <- n
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
          01     02     03     04     05     06           08     09

v

```

Iteration 2:

```

// n is null, so save the current right link and right neighbor

rightLinks[4]      = &(*v)[4] = &_head[4];
rightNeighbors[4] = n      = nullptr;

// go down to the next level

--currentLevel;      // currentLevel = 3;

// n becomes v's right neighbor at the next level

n = (*v)[currentLevel] = _head[3] = &(node 5);

h[5] -> null
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
          01     02     03     04     05     06           08     09

v

```

Iteration 3:

```

// n's key value is less than newKey, so go right:
//   v becomes n's NodeVector
//   n becomes n's right neighbor at the current level

v = &n->right
n = n->right[currentLevel] = (node 5)->right[3] = &(node 9);

```

```

h[5] -> null
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
      01      02      03      04      05      06          08      09
                           v                               n

```

Iteration 4:

```

// n's key value is not less than newKey,
// so save the current right link and right neighbor

rightLinks[3]      = &(*v)[3] = &(node 5)->right[3];
rightNeighbors[3] = n      = &(node 9);

// go down to the next level

--currentLevel;    // currentLevel = 2;

// n becomes v's right neighbor at the next level

n = (*v)[currentLevel] = (node 5)->right[2] = &(node 9);

h[5] -> null
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
      01      02      03      04      05      06          08      09
                           v                               n

```

Iteration 5:

```

// n's key value is not less than newKey,
// so save the current right link and right neighbor

rightLinks[2]      = &(*v)[2] = &(node 5)->right[2];
rightNeighbors[2] = n      = &(node 9);

// go down to the next level

--currentLevel;    // currentLevel = 1;

// n becomes v's right neighbor at the next level

n = (*v)[currentLevel] = (node 5)->right[1] = &(node 6);

```

```

h[5] -> null
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
          01      02      03      04      05      06           08      09
                           v           n

```

Iteration 6:

```

// n's key value is less than newKey, so go right:
//   v becomes n's NodeVector
//   n becomes n's right neighbor at the current level

v = &n->right                               = &(node 6)->right;
n = n->right[currentLevel] = (node 6)->right[1] = &(node 9);

h[5] -> null
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
          01      02      03      04      05      06           08      09
                           v           n

```

Iteration 7:

```

// n's key value is not less than newKey,
// so save the current right link and right neighbor

rightLinks[1]      = &(*v)[1] = &(node 6)->right[1];
rightNeighbors[1] = n        = &(node 9);

// go down to the next level

--currentLevel;    // currentLevel = 0;

// n becomes v's right neighbor at the next level

n = (*v)[currentLevel] = (node 6)->right[0] = &(node 8);

h[5] -> null
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
          01      02      03      04      05      06           08      09
                           v           n

```

Iteration 8:

```
// n's key value is not less than newKey,
// so save the current right link and right neighbor

rightLinks[0]      = &(*v)[0] = &(node 6)->right[0];
rightNeighbors[0] = n        = &(node 8);

// go down to the next level

--currentLevel;    // currentLevel = -1;

// we've dropped below level 0, so exit the loop

if (currentLevel < 0)
    break;

return NewKeySearch(false, nullptr);

rightLinks[5] = &_head[5]           rightNeighbors[5] = nullptr
rightLinks[4] = &_head[4]           rightNeighbors[4] = nullptr
rightLinks[3] = &(node 5)->right[3] rightNeighbors[3] = &(node 9)
rightLinks[2] = &(node 5)->right[2] rightNeighbors[2] = &(node 9)
rightLinks[1] = &(node 6)->right[1] rightNeighbors[1] = &(node 9)
rightLinks[0] = &(node 6)->right[0] rightNeighbors[0] = &(node 8)
```

After calling `_findInsertionPointForNewKey` and verifying that the `newElement`'s key value doesn't already exist (`insert_1.h`, lines 9-17), we're ready to create and attach the new node. To choose the new node's height (top level), we'll use a separate function (`SkipList.h`, line 65),

```
Level _getLevelForNewNode();
```

This function randomly chooses a level from the `_availableNodes` table, updates the table, and returns the chosen level. We begin by constructing our return value, `randomLevel` (`memberFunctions_1.h`, line 139). Line 141,

```
srand(static_cast<size_t>(time(nullptr)));
```

`seeds` (initializes) the Standard Library's random number generator. The `srand` function, pronounced “s-rand” (short for “seed random”), is provided by the `<cstdlib>` header (line 2). `srand` takes a single argument, an `unsigned int` used to seed the generator. We'll use the current time,

```
time(nullptr)
```

The Standard Library `time` function returns the current *Unix time* (the number of seconds that have passed since midnight on January 1st, 1970). This function, provided by the `<ctime>` header (line 3), takes a single argument, a pointer to the variable in which we'd like the time to be written directly. Since we don't need this functionality, we can just pass a null pointer here. We then use `static_cast` to convert the returned Unix time to a `size_t` (`unsigned int`) for `srand`.

After seeding the random number generator, we can choose a *randomLevel* (line 145):

```
randomLevel = rand() % _availableNodes.size();
```

The Standard Library *rand* function (short for “random”) returns a random integer value. *rand*, like *srand*, is provided by the `<cstdlib>` header. Modulus dividing the *rand()* value by `_availableNodes.size()` yields an integer in the set $[0, _availableNodes.size() - 1]$, which is equivalent to $[0, \text{topLevel}()]$.

If there is at least 1 available node at the currently chosen *randomLevel*, we decrement the *NodeCount* at that level and return the chosen level (lines 147-151, 154); otherwise, we choose another *randomLevel* and try again, until we find an available level. Since we've already verified that the list isn't full (*insert_1.h*, lines 6-7), we're guaranteed to have at least 1 available level.

To create the *newNode* (*insert_1.h*, line 19), we call *_createNode*, directly passing the return value from *_getLevelForNewNode* as the *topLevel* argument.

To attach the new node to the list (*insert_1.h*, line 21), we'll write another separate function (*SkipList.h*, lines 71-73),

```
void _attachNewNode(Node* newNode,
    const LinkVector& rightLinks,
    const NodeVector& rightNeighbors);
```

Let's walk through the code (*memberFunctions_1.h*, lines 204-219) using the previous example, in which the *newNode*'s key value is 7, and suppose that the *newNode*'s top level is 4 (its height is 5):

```

h[5] -> null
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0]
      01      02      03      04      05      06          08      09
      <- left <- left

rightLinks[5] = &_head[5]           rightNeighbors[5] = nullptr
rightLinks[4] = &_head[4]           rightNeighbors[4] = nullptr
rightLinks[3] = &(node 5)->right[3] rightNeighbors[3] = &(node 9)
rightLinks[2] = &(node 5)->right[2] rightNeighbors[2] = &(node 9)
rightLinks[1] = &(node 6)->right[1] rightNeighbors[1] = &(node 9)
rightLinks[0] = &(node 6)->right[0] rightNeighbors[0] = &(node 8)

if (rightNeighbors[0] != nullptr)
{
    newNode->left = rightNeighbors[0]->left;      // (node 7)->left = node 6;
    rightNeighbors[0]->left = newNode;              // (node 8)->left = node 7;
}
```


Iteration 4 (level 3):

```
*rightLinks[3] = newNode; // (node 5)->right[3] = node 7;
newNode->right[3] = rightNeighbors[3]; // (node 7)->right[3] = node 9;

h[5] -> null
h[4] -> null
h[3] -----> r[3] -----> r[3] -----> r[3]
r[4]
h[2] -----> r[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0]
    01      02      03      04      05      06      07      08      09
    <- left <- left
```

Iteration 5 (level 4):

```
*rightLinks[4] = newNode; // _head[4] = node 7;
newNode->right[4] = rightNeighbors[4]; // (node 7)->right[4] = nullptr;

h[5] -> null
h[4] -----> r[4] -> null
r[4]
h[3] -----> r[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0]
    01      02      03      04      05      06      07      08      09
    <- left <- left
```

If the *newNode* is being inserted at the end of the list, then *rightNeighbors[0]* is null, so we set the *newNode*'s left link to point to the *_tail*, then update the *_tail* to point to the *newNode* (lines 209-213). Given the list

```
          tail
          /-----\_
h[3] -----> r[3]
h[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -> r[1] -> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0]
    01      02      03      04      05
    <- left <- left <- left <- left
```

for example, suppose that the *newNode*'s key value is 6, and its top level is 1 (its height is 2):

```
rightLinks[3] = &(node 5)->right[3]    rightNeighbors[3] = nullptr
rightLinks[2] = &(node 5)->right[2]    rightNeighbors[2] = nullptr
rightLinks[1] = &(node 5)->right[1]    rightNeighbors[1] = nullptr
rightLinks[0] = &(node 5)->right[0]    rightNeighbors[0] = nullptr
```

```

else                                // if (rightNeighbors[0] == nullptr)
{
    newNode->left = _tail;      // (node 6)->left = node 5;
    _tail = newNode;            // _tail           = node 6;
}

h[3] -----> r[3]          tail
h[2] -----> r[2] -----> r[2]   /-
h[1] -> r[1] -----> r[1] -> r[1]   r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0]   r[0]
    01     02     03     04     05     06
    <- left <- left <- left <- left <- left <- left

```

// The remainder of the process (lines 215-219) is unchanged

After attaching the new node, we update the `_size` (*insert_1.h*, line 22) and the insertion is complete.

To verify the integrity of a list, let's write a function that prints out the entire structure (*printSkipList.h*, lines 8-9):

```

template <class SkipList>
void printSkipList(const SkipList& skipList);

```

This function prints all of the left / right links in addition to the elements. We begin by printing each link in the head, starting at level 0 (lines 19-32). If the current link *head[i]* isn't null, we print the referent node's key value; otherwise, we print "null." The extra whitespace (line 26) formats the output, for greater readability.

Beginning at the first (leftmost) node, we then traverse the list one node at a time (line 36). In each iteration, we print the current node *n*'s key value, followed by the key value of *n*'s left neighbor (lines 38-44). We then print the key value of each of *n*'s right neighbors, beginning at level 0 (lines 46-54).

In addition to the structural integrity of the list itself, we should also have a way to verify that the height distribution (the `_availableNodes` table) is intact. Let's write a separate function (lines 11-12),

```

template <class SkipList>
void printAvailableNodes(const SkipList& skipList);

```

which prints the `_availableNodes` table. For each level in the table, we simply print the corresponding `NodeCount` (the number of available nodes remaining) (lines 60-72).

Putting it all together, our test program (*main.cpp*) begins by constructing a *SkipList<Traceable<int>, int>>* *s* (lines 9-14). We then insert the key values [4, 0, 3, 1, 5, 2, 7, 6] (lines 16-23), and verify the contents (lines 26-27). The output will vary (because the height of each new node is chosen randomly), but here's one possible run:

```
// construct s
```

```

h[3] -> null
h[2] -> null
h[1] -> null
h[0] -> null

```

```
// insert 4 (level 3 node)
```

rightLinks[3] = &_head[3]	rightNeighbors[3] = nullptr
rightLinks[2] = &_head[2]	rightNeighbors[2] = nullptr
rightLinks[1] = &_head[1]	rightNeighbors[1] = nullptr
rightLinks[0] = &_head[0]	rightNeighbors[0] = nullptr

```

h[3] -----> r[3] -----> null
h[2] -----> r[2] -----> null
h[1] -----> r[1] -----> null
h[0] -----> r[0] -----> null
          04
null <----- left

```

```
// insert 0 (level 0 node)
```

rightLinks[3] = &_head[3]	rightNeighbors[3] = &(node 4)
rightLinks[2] = &_head[2]	rightNeighbors[2] = &(node 4)
rightLinks[1] = &_head[1]	rightNeighbors[1] = &(node 4)
rightLinks[0] = &_head[0]	rightNeighbors[0] = &(node 4)

```

h[3] -----> r[3] -----> null
h[2] -----> r[2] -----> null
h[1] -----> r[1] -----> null
h[0] -> r[0] -----> r[0] -----> null
          00          04
null <- left <----- left

```

```
// insert 3 (level 0 node)
```

rightLinks[3] = &_head[3]	rightNeighbors[3] = &(node 4)
rightLinks[2] = &_head[2]	rightNeighbors[2] = &(node 4)
rightLinks[1] = &_head[1]	rightNeighbors[1] = &(node 4)
rightLinks[0] = &(node 0) -> right[0]	rightNeighbors[0] = &(node 4)

```

h[3] -----> r[3] -----> null
h[2] -----> r[2] -----> null
h[1] -----> r[1] -----> null
h[0] -> r[0] -----> r[0] -> r[0] -----> null
          00          03          04
null <- left <----- left <- left

```

```
// insert 1 (level 1 node)
```

rightLinks[3] = &_head[3]	rightNeighbors[3] = &(node 4)
---------------------------	-------------------------------

```

rightLinks[2] = &_head[2]           rightNeighbors[2] = &(node 4)
rightLinks[1] = &_head[1]           rightNeighbors[1] = &(node 4)
rightLinks[0] = &(node 0)->right[0] rightNeighbors[0] = &(node 3)

h[3] -----> r[3] -----> null
h[2] -----> r[2] -----> null
h[1] -----> r[1] -----> r[1] -----> null
h[0] -> r[0] -> r[0] -----> r[0] -> r[0] -----> null
    00      01          03      04
null <- left <- left <----- left <- left

```

```

// insert 5 (level 1 node)

rightLinks[3] = &(node 4)->right[3]   rightNeighbors[3] = nullptr
rightLinks[2] = &(node 4)->right[2]   rightNeighbors[2] = nullptr
rightLinks[1] = &(node 4)->right[1]   rightNeighbors[1] = nullptr
rightLinks[0] = &(node 4)->right[0]   rightNeighbors[0] = nullptr

h[3] -----> r[3] -----> null
h[2] -----> r[2] -----> null
h[1] -----> r[1] -----> r[1] -> r[1] -----> null
h[0] -> r[0] -> r[0] -----> r[0] -> r[0] -> r[0] -----> null
    00      01          03      04      05
null <- left <- left <----- left <- left <- left

```

```

// insert 2 (level 2 node)

rightLinks[3] = &_head[3]           rightNeighbors[3] = &(node 4)
rightLinks[2] = &_head[2]           rightNeighbors[2] = &(node 4)
rightLinks[1] = &(node 1)->right[1] rightNeighbors[1] = &(node 4)
rightLinks[0] = &(node 1)->right[0] rightNeighbors[0] = &(node 3)

h[3] -----> r[3] -----> null
h[2] -----> r[2] -----> r[2] -----> null
h[1] -----> r[1] -> r[1] -----> r[1] -> r[1] -----> null
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> null
    00      01      02      03      04      05
null <- left <- left <- left <- left <- left <- left

```

```

// insert 7 (level 2 node)

rightLinks[3] = &(node 4)->right[3]   rightNeighbors[3] = nullptr
rightLinks[2] = &(node 4)->right[2]   rightNeighbors[2] = nullptr
rightLinks[1] = &(node 5)->right[1]   rightNeighbors[1] = nullptr
rightLinks[0] = &(node 5)->right[0]   rightNeighbors[0] = nullptr

```

```

h[3] -----> r[3] -----> null
h[2] -----> r[2] -----> r[2] -----> r[2] -> null
h[1] -----> r[1] -> r[1] -----> r[1] -> r[1] -----> r[1] -> null
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> null
      00      01      02      03      04      05          07
null <- left <----- left

```

```

// insert 6 (level 0 node)

rightLinks[3] = &(node 4)->right[3]    rightNeighbors[3] = nullptr
rightLinks[2] = &(node 4)->right[2]    rightNeighbors[2] = &(node 7)
rightLinks[1] = &(node 5)->right[1]    rightNeighbors[1] = &(node 7)
rightLinks[0] = &(node 5)->right[0]    rightNeighbors[0] = &(node 7)

h[3] -----> r[3] -----> null
h[2] -----> r[2] -----> r[2] -----> r[2] -> null
h[1] -----> r[1] -> r[1] -----> r[1] -> r[1] -----> r[1] -> null
h[0] -> r[0] -> null
      00      01      02      03      04      05      06      07
null <- left <- left

head[0]  0
head[1]  1
head[2]  2
head[3]  4

node    0
left    null
right[0] 1

node    1
left    0
right[0] 2
right[1] 2

node    2
left    1
right[0] 3
right[1] 4
right[2] 4

node    3
left    2
right[0] 4

node    4
left    3
right[0] 5
right[1] 5
right[2] 7
right[3] null

node    5

```

```
left      4
right[0]  6
right[1]  7

node      6
left      5
right[0]  7

node      7
left      6
right[0]  null
right[1]  null
right[2]  null

availableNodes[3] = 0
availableNodes[2] = 0
availableNodes[1] = 2
availableNodes[0] = 5
```

```
// destroy s
```

```
~0
~1
~2
~3
~4
~5
~6
~7
```

```
All Traceables destroyed
```


8.2: Increasing the Capacity

Source files and folders

- *SkipList/2*
- *SkipList/common/memberFunctions_2.h*
- *SkipList insert/insert_2.h*

Chapter outline

- *Automatically increasing the capacity when inserting an element into a full list*
- *Manually reserving a specific capacity in advance*

Our skip list, in its current form, can only hold 15 elements ($8 + 4 + 2 + 1$ nodes), after which subsequent calls to *insert* won't do anything. To increase the capacity, we need to increase the list's *topLevel*. This is a relatively simple procedure, handled by the function (*SkipList.h*, line 49)

```
void increaseTopLevel(Level newTopLevel);
```

If the *newTopLevel* isn't greater than the current *topLevel*, we exit without doing anything (*memberFunctions_2.h*, lines 10-11). Otherwise, we update the *_availableNodes* table by visiting each level *i* (line 13):

- The number of existing nodes at the current level is equal to the original capacity minus the number of available nodes (lines 15-16)
- The new number of available nodes at the current level is equal to the new capacity minus the current number of existing nodes (lines 18-20)

We then insert the new levels, (*topLevel* + 1) through the *newTopLevel*. The number of *_availableNodes* at each new level *i* is equal to the initial capacity, $2^{(newTopLevel - i)}$, since there aren't any existing nodes to subtract (lines 23-25). Lastly, we insert the new links in the *_head*, initialized to null (line 27).

Consider, for example, the list

```
h[3] -----> r[3]
h[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -> r[1] -> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0]
          01      02      03      04      05
          <- left <- left <- left <- left <- left
```

which has a *topLevel* of 3, with room for 10 more elements:

```
_availableNodes[3] = 0
_availableNodes[2] = 1
```

```
_availableNodes[1] = 2
_availableNodes[0] = 7
```

To *increaseTopLevel* to 5, we begin by calculating the new number of *_availableNodes* at levels 0 to 3:

level	capacity (original)	$2^{(topLevel - i)}$	existingNodes	$2^{(newTopLevel - i)}$	newCapacity	$availableNodes$ (new)
0	8	1	1	32	31	
1	4	2	2	16	14	
2	2	1	1	8	7	
3	1	0	1	4	3	

We then insert the new levels, 4 and 5:

level	$2^{(newTopLevel - i)}$	$availableNodes$ (new)
4		2
5		1

The list now has room for 58 more elements:

```
_availableNodes[5] = 1
_availableNodes[4] = 2
_availableNodes[3] = 3
_availableNodes[2] = 7
_availableNodes[1] = 14
_availableNodes[0] = 31
```

We can now update the *insert* function to automatically increase the capacity as needed, by simply increasing the top level to the current *topLevel* + 1 (*insert_2.h*, lines 6-7). The remainder of the function is unchanged.

Although *increaseTopLevel* is a public method, from a user's perspective it would be more practical to have an additional function for reserving a specific capacity (number of elements) (*SkipList.h*, line 50):

```
void reserve(size_type newCapacity);
```

This function increases the capacity such that it meets or exceeds the desired *newCapacity*, by increasing the top level. If the *newCapacity* is less than or equal to the current capacity, the function doesn't do anything.

Recall that the formula for determining the total capacity is

```
capacity           = 2^(topLevel + 1) - 1
```

Solving this equation for *topLevel*, the formula becomes

```
capacity + 1      = 2^(topLevel + 1)
log2(capacity + 1) = topLevel + 1
log2(capacity + 1) - 1 = topLevel
topLevel          = log2(capacity + 1) - 1
```

The *topLevel* must be a whole number, so we also need to take the ceiling (recall that we used the ceiling function in Chapter 1.1, when implementing the Heap class):

```
topLevel           = ceiling(log2(capacity + 1) - 1)
```

To implement *reserve*, we use this formula to calculate the *newTopLevel* (the lowest top level that will meet or exceed the desired *newCapacity*) (*memberFunctions_2.h*, line 37). The Standard Library function *log2*, provided by the <*cmath*> header (line 1), returns the binary logarithm of the given value. We then *increaseTopLevel* to the *newTopLevel* (line 38).

Our test program (*main.cpp*) begins by constructing an empty list *s*, filling it to capacity with the key values [0, 14], and printing *s* (lines 12-21). We then insert the key value 15 (line 23), which automatically increases the *topLevel* to 4 and the capacity to 31 elements, after which we print *s* again (lines 26-27).

We then manually reserve a capacity of 32 elements (line 29), which increases the *topLevel* to 5 and the capacity to 63 elements, after which we print *s* once again to verify the changes (lines 30-31).

```
// insert [0, 14] (fill s to capacity)

head[0]      0
head[1]      8
head[2]      8
head[3]      8

node         0
  left       null
  right[0]   1

node         1
  left       0
  right[0]   2

node         2
  left       1
  right[0]   3

node         3
  left       2
  right[0]   4

node         4
```

```
left      3
right[0]  5

node      5
left      4
right[0]  6

node      6
left      5
right[0]  7

node      7
left      6
right[0]  8

node      8
left      7
right[0]  9
right[1]  9
right[2]  9
right[3]  null

node      9
left      8
right[0]  10
right[1]  10
right[2]  10

node      10
left      9
right[0]  11
right[1]  11
right[2]  null

node      11
left      10
right[0]  12
right[1]  12

node      12
left      11
right[0]  13
right[1]  13

node      13
left      12
right[0]  14
right[1]  14

node      14
left      13
right[0]  null
right[1]  null
```

```
availableNodes[0] = 0
availableNodes[1] = 0
availableNodes[2] = 0
availableNodes[3] = 0

// insert 15 (topLevel increases to 4, capacity increases to 31)

head[0]      0
head[1]      8
head[2]      8
head[3]      8
head[4]      15

node        0
  left      null
  right[0]  1

node        1
  left      0
  right[0]  2

node        2
  left      1
  right[0]  3

node        3
  left      2
  right[0]  4

node        4
  left      3
  right[0]  5

node        5
  left      4
  right[0]  6

node        6
  left      5
  right[0]  7

node        7
  left      6
  right[0]  8

node        8
  left      7
  right[0]  9
  right[1]  9
  right[2]  9
  right[3]  15

node        9
```

```

left      8
right[0] 10
right[1] 10
right[2] 10

node      10
left      9
right[0] 11
right[1] 11
right[2] 15

node      11
left      10
right[0] 12
right[1] 12

node      12
left      11
right[0] 13
right[1] 13

node      13
left      12
right[0] 14
right[1] 14

node      14
left      13
right[0] 15
right[1] 15

node      15
left      14
right[0] null
right[1] null
right[2] null
right[3] null
right[4] null

availableNodes[0] = 8
availableNodes[1] = 4
availableNodes[2] = 2
availableNodes[3] = 1
availableNodes[4] = 0

```

```

// reserve(32) (topLevel increases to 5, capacity increases to 63)

head[0]    0
head[1]    8
head[2]    8
head[3]    8
head[4]    15
head[5]    null

```

```
node      0
  left    null
  right[0] 1
```

```
node      1
  left    0
  right[0] 2
```

```
node      2
  left    1
  right[0] 3
```

```
node      3
  left    2
  right[0] 4
```

```
node      4
  left    3
  right[0] 5
```

```
node      5
  left    4
  right[0] 6
```

```
node      6
  left    5
  right[0] 7
```

```
node      7
  left    6
  right[0] 8
```

```
node      8
  left    7
  right[0] 9
  right[1] 9
  right[2] 9
  right[3] 15
```

```
node      9
  left    8
  right[0] 10
  right[1] 10
  right[2] 10
```

```
node      10
  left    9
  right[0] 11
  right[1] 11
  right[2] 15
```

```
node      11
  left    10
  right[0] 12
```

```
    right[1] 12

node      12
  left     11
  right[0] 13
  right[1] 13

node      13
  left     12
  right[0] 14
  right[1] 14

node      14
  left     13
  right[0] 15
  right[1] 15

node      15
  left     14
  right[0] null
  right[1] null
  right[2] null
  right[3] null
  right[4] null

availableNodes[0] = 24
availableNodes[1] = 12
availableNodes[2] = 6
availableNodes[3] = 3
availableNodes[4] = 1
availableNodes[5] = 1
```

```
// destroy s
```

```
~ 0
~ 1
~ 2
~ 3
~ 4
~ 5
~ 6
~ 7
~ 8
~ 9
~ 10
~ 11
~ 12
~ 13
~ 14
~ 15
```

```
All Traceables destroyed
```

8.3: Implementing the Iterators

Source files and folders

- *SkipList/3*
- *SkipList/common/memberFunctions_3.h*
- *SkipList/common/SkipListIter.h*
- *SkipList/insert/insert_3.h*

Chapter outline

- *Implementing SkipList's iterator, reverse_iterator, const_iterator, and const_reverse_iterator*
- *Implementing SkipList::find*
- *Updating SkipList::insert with the standard return type (pair<iterator, bool>)*

A skip list iterator is nearly identical to a linked list iterator: it contains a pointer to the referent element's node, along with a pointer to the corresponding list (*SkipListIter.h*, lines 8-9, 33, 37-38).

The private constructor (lines 12, 35) will be used from within *SkipList*'s member functions (*begin*, *end*, *insert*, etc.). It simply initializes the *_skipList* and *_node* pointers according to the given arguments (lines 112-118). The default constructor (line 20) is left blank (lines 41-45).

The comparison operators (lines 22-23) determine whether two iterators point to the same element by comparing their *_node* pointers (lines 47-57).

The dereference and member access operators (lines 24-25) return a reference / pointer to the *_node*'s element (lines 59-71).

The increment operators (lines 27, 29) advance the iterator to the next element in the sequence by following the *_node*'s right link at level 0 (lines 73-79, 92-100).

The decrement operators (lines 28, 30) point to the previous element in the sequence by following the *_node*'s left link. If we're currently at the *end*, we back up to the *_tail* via the *_skipList* pointer (lines 81-90, 102-110).

Moving on to the *SkipList* class (*SkipList.h*), we'll apply the same technique that we used for *BTree*'s iterators in Chapter 6.7:

- *iterator* is an alias of *SkipListIter<SkipList>* (line 34)
- *reverse_iterator* is an alias *dss::ReverseIter<iterator>* (line 35)
- *const_iterator* is an alias of *dss::ConstIter<SkipList>* (line 30)
- *const_reverse_iterator* is an alias of *dss::ReverseIter<const_iterator>* (line 31)

begin (lines 50, 62) returns an *iterator* / *const_iterator* to the *front* element (*memberFunctions_3.h*,

lines 5-10, 40-45).

end (*SkipList.h*, lines 51, 63) returns an *iterator / const_iterator* to the one-past-the-last element (*memberFunctions_3.h*, lines 12-17, 47-52).

rbegin (*SkipList.h*, lines 52, 64) returns a *reverse_iterator / const_reverse_iterator* to the first element of the reverse sequence (*memberFunctions_3.h*, lines 19-24, 54-59).

rend (*SkipList.h*, lines 53, 65) returns a *reverse_iterator / const_reverse_iterator* to the one-past-the-last element of the reverse sequence (*memberFunctions_3.h*, lines 26-31, 61-66).

find (*SkipList.h*, lines 56, 68) returns an *iterator / const_iterator* to the element with the given *key* value (or *end*, if the *key* doesn't exist). The implementation is nearly identical to that of *findInsertionPointForNewKey*. The only differences are that we aren't saving the *rightLinks* or *rightNeighbors*, and the return value is an *iterator* (as opposed to a *NewKeySearch*) (*memberFunctions_3.h*, lines 33-38, 68-99). If we find a match for the desired *key* value, we return an iterator to the current element (node) (lines 78-82). If we drop below level 0 before finding a match, we return an iterator to the *end* (lines 94-95).

We can now update *insert* to use the standard return type, *pair<iterator, bool>* (*SkipList.h*, line 71):

- If the *newElement* was inserted, the *pair::second* is *true* and *pair::first* points to the newly inserted element
- If the *newElement* was not inserted (because the tree already contains an element with the same key value), the *pair::second* is *false* and its *pair::first* points to the preexisting element

The implementation (*insert_3.h*) is nearly identical to the previous version, with only 2 minor differences:

- If the new key already exists, we return a pair containing an iterator to the preexisting element and *false* (lines 19-20)
- After inserting the new element, we return a pair containing an iterator to the new element (*newNode*) and *true* (line 27)

Our test program (*main.cpp*) begins by constructing a list *s* and inserting the key values {0, 2, 4, 6, 8} (lines 14-19). We then call *printContainer* and *printContainerReverse*, which print *s* using *const_iterator* and *const_reverse_iterator* (lines 21-22). The resultant output is

```
(0,0) (2,0) (4,0) (6,0) (8,0)
(8,0) (6,0) (4,0) (2,0) (0,0)
```

We then insert the key value 7, saving the returned *pair p* (line 24). Lines 26-27 check whether the new element was inserted, and if so, print the new element's key value, generating the output

```
Inserted element (7,0)
```

Lines 29-30 call *printContainer* and *printContainerReverse* once again, generating the output

```
(0,0) (2,0) (4,0) (6,0) (7,0) (8,0)  
(8,0) (7,0) (6,0) (4,0) (2,0) (0,0)
```

We then try to insert the element (4, 1), which contains the duplicate key 4 (line 32). *p.first* is therefore *false*, and *p.second* is an iterator to the preexisting element, (4, 0). Lines 34-35 check whether the new element was not inserted, and if so, print the preexisting element, generating the output

```
Element (4,0) already exists
```

To demonstrate the *find* method, we then search for the key value 6 (line 37). Lines 39-40 check whether the desired key was found, and if so, print the element, generating the output

```
Found element (6,0)
```

We then search for the nonexistent key value 9 (line 42). Because the desired key value was not found, lines 44-45 generate the output

```
Key 9 not found
```


8.4: Erasing Elements

Source files and folders

- *SkipList/4*
- *SkipList/common/memberFunctions_4.h*

Chapter outline

- *Implementing SkipList's erase method*

Now that we have working iterators, we can implement the function (*SkipList.h*, line 72)

```
iterator erase(const_iterator element);
```

which erases the given *element* and returns an iterator to the next element in the list. Consider, for example, the list

```
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0]
      01      02      03      04      05      06      07      08      09
      <- left <- left
```

and suppose that we're given an iterator to element 5, which we'd like to erase. We can easily obtain a pointer to node 5 from within the iterator, but to detach node 5 from the list, we also need to know which right links point to node 5 at each level (0, 1, 2, and 3). Each of these links must be updated to point to node 5's right neighbor at the corresponding level:

```
_head[3]           must be updated to point to (node 5)->right[3] (node 9)
(node 3)->right[2] must be updated to point to (node 5)->right[2] (node 7)
(node 3)->right[1] must be updated to point to (node 5)->right[1] (node 6)
(node 4)->right[0] must be updated to point to (node 5)->right[0] (node 6)
```

To find these links we'll write a separate function (*SkipList.h*, line 94),

```
void _findRightLinksOnErase(Node* trash, LinkVector& rightLinks);
```

where *trash* is the node that we plan to erase, and *rightLinks* is the vector in which we'll save (pointers to) the right links that point to *trash*.

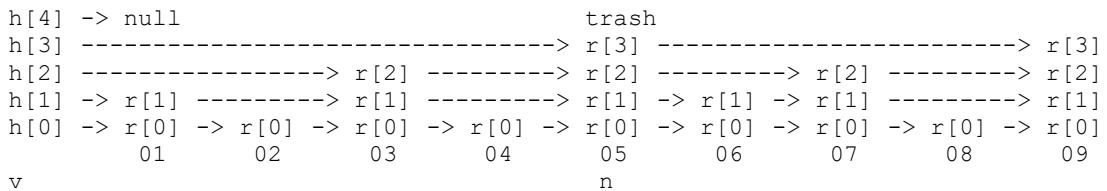
We begin by resizing *rightLinks* to the same height as *trash* (*memberFunctions_4.h*, line 26). The *currentLevel* is initialized to *trash*'s top level, the current node *n* is initialized to the *_head* node at *trash*'s top level, and the current *NodeVector* *v* is initialized to the *_head* (lines 28-30).

We then search for *trash*, saving (pointers to) the right links that point to *trash* at each level:

- If the current node is *trash*, we save (a pointer to) the current *NodeVector*'s right link at the current level, then go down to the next level (lines 34-37). If we've dropped below level 0, the search is complete (lines 39-40); otherwise, the current node becomes the current *NodeVector*'s right neighbor at the next level, and we begin the next iteration (lines 42-43).
- If the current node isn't *trash*, we go right: *v* becomes the current node's *NodeVector*, and *n* becomes the current node's right neighbor at the current level (lines 46-47). We then begin the next iteration.

Let's walk through this code, using the above example. In the following diagrams, the left links are omitted since we don't need them:

```
currentLevel = 3;
n = _head[3];
v = &_head;
```

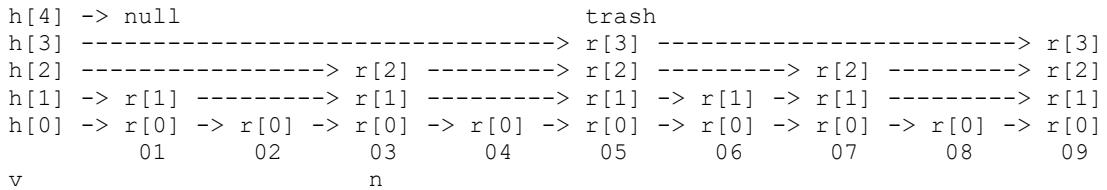


Iteration 1:

```
// n is trash, so save the current right link
rightLinks[3] = &(*v)[3] = &head[3];

// go down to the next level
--currentLevel; // currentLevel = 2;

// n becomes v's right neighbor at the next level
n = (*v)[currentLevel] = _head[2] = &(node 3);
```



Iteration 2:

```

// n isn't trash, so go right:
//   v becomes n's NodeVector
//   n becomes n's right neighbor at the current level

v = &n->right                                = &(node 3)->right;
n = n->right[currentLevel] = (node 3)->right[2] = &(node 5);

n[4] -> null                               trash
h[3] -----> r[2] -----> r[2] -----> r[2] -----> r[3]
h[2] -----> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[2]
h[1] -> r[0] -> r[1]
h[0] -> r[0] -> r[0]
    01      02      03      04      05      06      07      08      09
          v           n

```

Iteration 3:

```

// n is trash, so save the current right link

rightLinks[2] = &(*v)[2] = &(node 3)->right[2];

// go down to the next level

--currentLevel; // currentLevel = 1;

// n becomes v's right neighbor at the next level

n = (*v)[currentLevel] = (node 3)->right[1] = &(node 5);

h[4] -> null                               trash
h[3] -----> r[2] -----> r[2] -----> r[2] -----> r[3]
h[2] -----> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[2]
h[1] -> r[0] -> r[1]
h[0] -> r[0] -> r[0]
    01      02      03      04      05      06      07      08      09
          v           n

```

Iteration 4:

```

// n is trash, so save the current right link

rightLinks[1] = &(*v)[1] = &(node 3)->right[1];

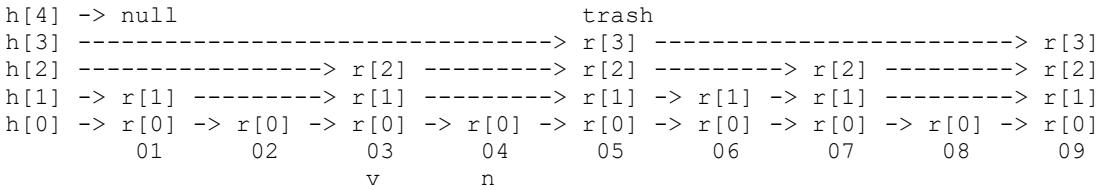
// go down to the next level

--currentLevel; // currentLevel = 0;

// n becomes v's right neighbor at the next level

n = (*v)[currentLevel] = (node 3)->right[0] = &(node 4);

```



Iteration 5:

```

// n isn't trash, so go right:
//   v becomes n's NodeVector
//   n becomes n's right neighbor at the current level

v = &n->right           = &(node 4)->right;
n = n->right[currentLevel] = (node 4)->right[0] = &(node 5);

h[4] -> null           trash
h[3] -----> r[2] -----> r[3] -----> r[3]
h[2] -----> r[1] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0]
    01      02      03      04      05      06      07      08      09
        v          n

```

Iteration 6:

```

// n is trash, so save the current right link

rightLinks[0] = &(*v)[0] = &(node 4)->right[0];

// go down to the next level

--currentLevel; // currentLevel = -1;

// we've dropped below level 0, so exit the loop

if (currentLevel < 0)
    break;

```

```

rightLinks[3] = &_head[3]
rightLinks[2] = &(node 3)->right[2]
rightLinks[1] = &(node 3)->right[1]
rightLinks[0] = &(node 4)->right[0]

```

Now that we know which right links to update, we can easily detach the unwanted node (*trash*) from the list. Let's write a separate function to handle this (*SkipList.h*, line 100):

```
void _detachNode(Node* trash, const LinkVector& rightLinks);
```

This function detaches *trash* from the list by updating each of the *rightLinks* that currently point to *trash* (*memberFunctions_4.h*, lines 51-62):

- If *trash* is currently the tail node, we begin by updating the *_tail* to point to the new tail node (*trash*'s left neighbor) (lines 55-56); otherwise, we update the left link currently pointing to *trash* to point to its new left neighbor (*trash*'s left neighbor) (lines 57-58).
- We then update each *rightLink* currently pointing to *trash* to point to its new right neighbor (*trash*'s right neighbor at the corresponding level) (lines 60-61).

Continuing with the above example:

```

h[4] -> null          trash           _tail
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0]
      01      02      03      04      05      06      07      08      09
      <- left <- left

rightLinks[3] = &_head[3]
rightLinks[2] = &(node 3)->right[2]
rightLinks[1] = &(node 3)->right[1]
rightLinks[0] = &(node 4)->right[0]

// trash isn't the _tail, so update the left link currently pointing to
// trash to point to its new left neighbor (trash's left neighbor):
trash->right[0]->left = trash->left; // (node 6)->left = (node 4);

h[4] -> null          trash           _tail
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0]
      01      02      03      04      05      06      07      08      09
      <- left <- left

// update each right link currently pointing to trash to point to its new
// right neighbor (trash's right neighbor at the corresponding level)

Iteration 1:
*rightLinks[0] = trash->right[0]; // (node 4)->right[0] = (node 6);

```

```

h[4] -> null           trash
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0] -> r[0] -> r[0]
          01      02      03      04      05      06      07      08      09
          <- left <- left

```

Iteration 2:

```
*rightLinks[1] = trash->right[1]; // (node 3)->right[1] = (node 6);
```

```

h[4] -> null                               trash
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0] -> r[0] -> r[0]
          01      02      03      04      05      06      07      08      09
          <- left <- left <- left <- left <----- left <- left <- left <- left

```

Iteration 3:

```
*rightLinks[2] = trash->right[2]; // (node 3)->right[2] = (node 7);
```

```

h[4] -> null           trash
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0] -> r[0] -> r[0] -> r[0]
          01      02      03      04      05      06      07      08      09
          left    left    left    left    left    right   left    left    left
  
```

Iteration 4:

```
*rightLinks[3] = trash->right[3]; // head[3] = (node 9);
```

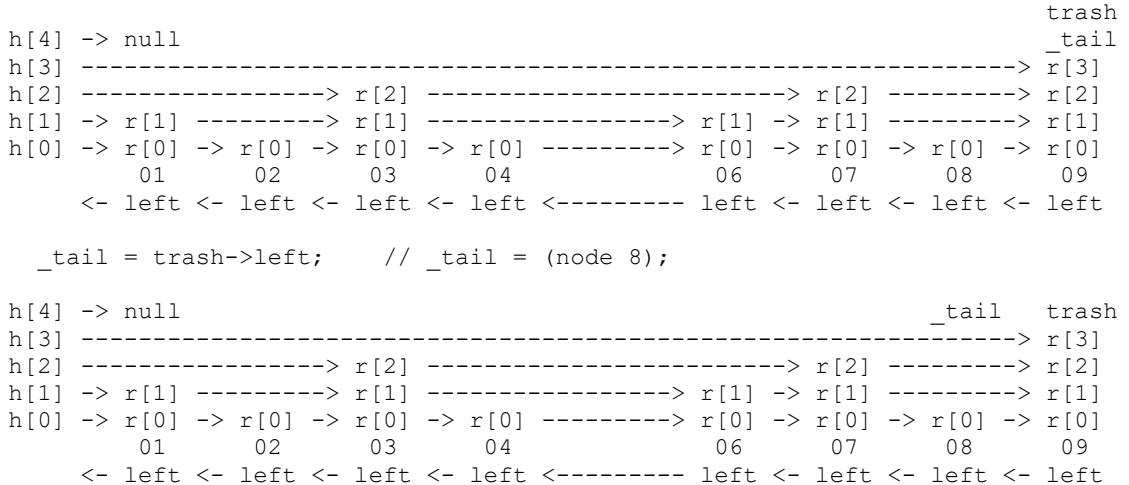
```

h[4] -> null
h[3] -----
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -----> r[1]
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -----> r[0] -> r[0] -> r[0] -> r[0]
          01      02      03      04           06      07      08      09
          <- left <- left <- left <- left <----- left <- left <- left <- left

```

```
trash
r[3] -> (node 9) // node 5 (trash)
r[2] -> (node 7) // can now be
r[1] -> (node 6) // safely destroy
r[0] -> (node 6)
    05
```

If *trash* is currently the tail node, then there is no left link pointing to *trash*; we simply update the *_tail* pointer to point to the new tail node (*trash*'s left neighbor) (lines 55-56). The remainder of the process is unchanged:



Putting it all together, the *erase* function begins by obtaining a pointer to the desired node *trash* from within the given iterator (*memberFunctions_4.h*, line 7). We then save a pointer to the *next* node in the list, *trash*'s right neighbor (line 8).

After detaching *trash* from the list (lines 9-12), we update the *_availableNodes* table at the corresponding level, which now contains 1 more available node (line 14). We then update the *_size*, destroy *trash*, and return an iterator to the next element (node) (lines 15-19).

Our test program (*main.cpp*) begins by constructing a list *s*, inserting the key values [0, 9], and printing the entire list (lines 10, 19-25). To test *SkipList*'s *erase* method, we call the function (line 12)

```
void eraseAndPrint(_SkipList* s, int key);
```

which erases the element with the desired *key* value from the list *s*, then prints the entire list:

- If the *key* was found (lines 43-45), we erase it from the list and obtain an iterator to the next element (line 47), then print the next element (lines 48-53).
- If the *key* wasn't found, we inform the user accordingly (lines 55-58).
- In both cases, we end the function by printing the entire list (lines 60-61).

In *main*, we proceed to *eraseAndPrint* the key values 5, 0, and 9 (lines 27-29).

```
// insert [0, 9]
```

```
h[3]----->r[3]----->
h[2]----->r[2]-->r[2]-->r[2]----->
h[1]-->r[1]-->r[1]-->r[1]-->r[1]-->r[1]-->r[1]----->
h[0]-->r[0]-->r[0]-->r[0]-->r[0]-->r[0]-->r[0]-->r[0]-->r[0]-->r[0]-->r[0]-->r[0]-->
      00      01      02      03      04      05      06      07      08      09
<--left<--left<--left<--left<--left<--left<--left<--left<--left<--left<--left<--left

head[0]    0
head[1]    0
head[2]    4
head[3]    4

node      0
  left    null
  right[0] 1
  right[1] 1

node      1
  left    0
  right[0] 2
  right[1] 2

node      2
  left    1
  right[0] 3
  right[1] 3

node      3
  left    2
  right[0] 4
  right[1] 4

node      4
  left    3
  right[0] 5
  right[1] 5
  right[2] 5
  right[3] null

node      5
  left    4
  right[0] 6
  right[1] 6
  right[2] 6

node      6
  left    5
  right[0] 7
  right[1] null
  right[2] null

node      7
  left    6
  right[0] 8
```

```

node      8
  left    7
  right[0] 9

node      9
  left    8
  right[0] null

availableNodes[0] = 5
availableNodes[1] = 0
availableNodes[2] = 0
availableNodes[3] = 0


---


// eraseAndPrint(&s, 5);

h[3]----->r[3]----->
h[2]----->r[2]----->r[2]----->
h[1]-->r[1]-->r[1]-->r[1]-->r[1]----->r[1]----->
h[0]-->r[0]-->r[0]-->r[0]-->r[0]----->r[0]-->r[0]-->r[0]-->
          00      01      02      03      04          06      07      08      09
<--left<--left<--left<--left<-----left<--left<--left<--left<--left
Erasing key 5...

Next element = (6,0)
head[0]    0
head[1]    0
head[2]    4
head[3]    4

node      0
  left    null
  right[0] 1
  right[1] 1

node      1
  left    0
  right[0] 2
  right[1] 2

node      2
  left    1
  right[0] 3
  right[1] 3

node      3
  left    2
  right[0] 4
  right[1] 4

node      4
  left    3
  right[0] 6

```

```

right[1] 6
right[2] 6
right[3] null

node      6
left      4
right[0] 7
right[1] null
right[2] null

node      7
left      6
right[0] 8

node      8
left      7
right[0] 9

node      9
left      8
right[0] null

availableNodes[0] = 5
availableNodes[1] = 0
availableNodes[2] = 1
availableNodes[3] = 0



---


// eraseAndPrint(&s, 0);

h[3]----->r[3]----->
h[2]----->r[2]----->r[2]----->
h[1]----->r[1]-->r[1]-->r[1]-->r[1]----->r[1]----->
h[0]----->r[0]-->r[0]-->r[0]-->r[0]----->r[0]-->r[0]-->r[0]-->r[0]-->
          01      02      03      04          06      07      08      09
<-----left<--left<--left<--left<-----left<--left<--left<--left<--left

```

Erasing key 0...

```

Next element = (1,0)
head[0]    1
head[1]    1
head[2]    4
head[3]    4

node      1
left      null
right[0]  2
right[1]  2

node      2
left      1
right[0]  3
right[1]  3

```

```

node      3
  left    2
  right[0] 4
  right[1] 4

node      4
  left    3
  right[0] 6
  right[1] 6
  right[2] 6
  right[3] null

node      6
  left    4
  right[0] 7
  right[1] null
  right[2] null

node      7
  left    6
  right[0] 8

node      8
  left    7
  right[0] 9

node      9
  left    8
  right[0] null

availableNodes[0] = 5
availableNodes[1] = 1
availableNodes[2] = 1
availableNodes[3] = 0



---


// eraseAndPrint(&s, 9);

h[3]----->r[3]----->
h[2]----->r[2]----->r[2]----->
h[1]----->r[1]-->r[1]-->r[1]-->r[1]----->r[1]----->
h[0]----->r[0]-->r[0]-->r[0]-->r[0]----->r[0]-->r[0]-->r[0]----->
          01      02      03      04          06      07      08
<-----left<--left<--left<--left<-----left<--left<--left

```

Erasing key 9...

```

Next element = end
head[0]    1
head[1]    1
head[2]    4
head[3]    4

```

```
node      1
  left    null
  right[0] 2
  right[1] 2

node      2
  left    1
  right[0] 3
  right[1] 3

node      3
  left    2
  right[0] 4
  right[1] 4

node      4
  left    3
  right[0] 6
  right[1] 6
  right[2] 6
  right[3] null

node      6
  left    4
  right[0] 7
  right[1] null
  right[2] null

node      7
  left    6
  right[0] 8

node      8
  left    7
  right[0] null

availableNodes[0] = 6
availableNodes[1] = 1
availableNodes[2] = 1
availableNodes[3] = 0
```

```
// destroy s
```

```
~ 1
~ 2
~ 3
~ 4
~ 6
~ 7
~ 8
```

```
All Traceables destroyed
```

8.5: Implementing Copy and Assignment

Source files and folders

- *SkipList/5*
- *SkipList/common/memberFunctions_5.h*

Chapter outline

- *Implementing SkipList's copy constructor and assignment operator*

Now that we've implemented *insert* and *erase*, we can complete our skip list implementation by writing the copy constructor and assignment operator.

SkipList's copy constructor (*SkipList.h*, line 45) creates an identical copy of the given *source* list. We begin by initializing the *_head* to the same size as that of the *source* list, with a null pointer at each level (*memberFunctions_5.h*, line 7). The *_tail* is initialized to null, and the *_size* and *_availableNodes* table are copied from the *source* list (lines 8-10).

The *LinkVector rightLinks*, initialized to the same size as the *_head* (line 12), will contain (a pointer to) the rightmost right link at each level as we copy the *source* nodes and connect them to the new list.

Before we begin copying the *source* nodes, we initialize each *rightLink* to the *_head* link at the corresponding level (lines 14-15). Beginning at the first *sourceNode* (node in the *source* list), we then traverse the *source* list, one node at a time (lines 17-19). In each iteration,

- We create a *newNode* with the same height and element as the *sourceNode* (line 21)
- For each level *i* of the *newNode*, we update the *rightLink* at level *i* to point to the *newNode* (line 25), after which the current *rightLink* at level *i* becomes *newNode*'s right link at level *i* (line 26).
- We then set the *newNode*'s left link to point to the *_tail* (line 29), after which the *_tail* becomes the *newNode* (line 30).

Consider, for example, the *source* list

```

h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0]
          01      02      03      04      05      06      07      08      09
null <- left <- left

```

Preparing for the main loop (lines 7-15), we have

```

    _head
  /-
h[4] -> null
h[3] -> null
h[2] -> null
h[1] -> null
h[0] -> null
      _tail

rightLinks[4] = &_head[4];
rightLinks[3] = &_head[3];
rightLinks[2] = &_head[2];
rightLinks[1] = &_head[1];
rightLinks[0] = &_head[0];

    _source._head
  /-
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0]
      01      02      03      04      05      06      07      08      09
null <- left <- left

```

We're now ready to copy the *source* nodes (lines 17-31):

```

// Iteration 1 (sourceNode = 1)

newNode = _createNode(sourceNode->topLevel(), sourceNode->element);

*rightLinks[0] = newNode;           // _head[0]      = &(node 1);
rightLinks[0] = &newNode->right[0]; // rightLinks[0] = &(node 1)->right[0];

*rightLinks[1] = newNode;           // _head[1]      = &(node 1);
rightLinks[1] = &newNode->right[1]; // rightLinks[1] = &(node 1)->right[1];

newNode->left = _tail;           // (node 1)->left = nullptr;
_tail            = newNode;         // _tail          = &(node 1);

    _head
  /-
h[4] -> null
h[3] -> null
h[2] -> null
h[1] -> r[1] -> null
h[0] -> r[0] -> null
      01
null <- left
      newNode
      _tail

```

```

rightLinks[4] = &_head[4];
rightLinks[3] = &_head[3];
rightLinks[2] = &_head[2];
rightLinks[1] = &(node 1)->right[1];
rightLinks[0] = &(node 1)->right[0];

// Iteration 2 (sourceNode = 2)

newNode = _createNode(sourceNode->topLevel(), sourceNode->element);

*rightLinks[0] = newNode;           // (node 1)->right[0] = &(node 2);
rightLinks[0] = &newNode->right[0]; // rightLinks[0] = &(node 2)->right[0];

newNode->left = _tail;           // (node 2)->left = &(node 1);
_tail          = newNode;         // _tail          = &(node 2);

      head
      /
h[4] -> null
h[3] -> null
h[2] -> null
h[1] -> r[1] -> null
h[0] -> r[0] -> r[0] -> null
      01      02
null <- left <- left
      newNode
      _tail

rightLinks[4] = &_head[4];
rightLinks[3] = &_head[3];
rightLinks[2] = &_head[2];
rightLinks[1] = &(node 1)->right[1];
rightLinks[0] = &(node 2)->right[0];

// Iteration 3 (sourceNode = 3)

newNode = _createNode(sourceNode->topLevel(), sourceNode->element);

*rightLinks[0] = newNode;           // (node 2)->right[0] = &(node 3);
rightLinks[0] = &newNode->right[0]; // rightLinks[0] = &(node 3)->right[0];

*rightLinks[1] = newNode;           // (node 1)->right[1] = &(node 3);
rightLinks[1] = &newNode->right[1]; // rightLinks[1] = &(node 3)->right[1];

*rightLinks[2] = newNode;           // _head[2]      = &(node 3);
rightLinks[2] = &newNode->right[2]; // rightLinks[2] = &(node 3)->right[2];

newNode->left = _tail;           // (node 3)->left = &(node 2);
_tail          = newNode;         // _tail          = &(node 3);

```

```

    head
  /-
h[4] -> null
h[3] -> null
h[2] -> -----> r[2] -> null
h[1] -> r[1] -----> r[1] -> null
h[0] -> r[0] -> r[0] -> r[0] -> null
      01      02      03
null <- left <- left <- left
          newNode
          _tail

rightLinks[4] = &_head[4];
rightLinks[3] = &_head[3];
rightLinks[2] = &(node 3)->right[2];
rightLinks[1] = &(node 3)->right[1];
rightLinks[0] = &(node 3)->right[0];



---


// Iteration 4 (sourceNode = 4)

newNode = _createNode(sourceNode->topLevel(), sourceNode->element);

*rightLinks[0] = newNode;           // (node 3)->right[0] = &(node 4);
rightLinks[0] = &newNode->right[0]; // rightLinks[0] = &(node 4)->right[0];

newNode->left = _tail;           // (node 4)->left = &(node 3);
_tail        = newNode;           // _tail        = &(node 4);

    head
  /-
h[4] -> null
h[3] -> null
h[2] -> -----> r[2] -> null
h[1] -> r[1] -----> r[1] -> null
h[0] -> r[0] -> r[0] -> r[0] -> null
      01      02      03      04
null <- left <- left <- left <- left
          newNode
          _tail

rightLinks[4] = &_head[4];
rightLinks[3] = &_head[3];
rightLinks[2] = &(node 3)->right[2];
rightLinks[1] = &(node 3)->right[1];
rightLinks[0] = &(node 4)->right[0];



---


// Iteration 5 (sourceNode = 5)

newNode = _createNode(sourceNode->topLevel(), sourceNode->element);

*rightLinks[0] = newNode;           // (node 4)->right[0] = &(node 5);
rightLinks[0] = &newNode->right[0]; // rightLinks[0] = &(node 5)->right[0];

```

```

*rightLinks[1] = newNode;           // (node 3)->right[1] = &(node 5);
rightLinks[1] = &newNode->right[1]; // rightLinks[1] = &(node 5)->right[1];

*rightLinks[2] = newNode;           // (node 3)->right[2] = &(node 5);
rightLinks[2] = &newNode->right[2]; // rightLinks[2] = &(node 5)->right[2];

*rightLinks[3] = newNode;           // _head[3] = &(node 5);
rightLinks[3] = &newNode->right[3]; // rightLinks[3] = &(node 5)->right[3];

newNode->left = _tail;           // (node 5)->left = &(node 4);
_tail          = newNode;         // _tail          = &(node 5);

head
/
h[4] -> null
h[3] -----> r[3] -> null
h[2] -> -----> r[2] -----> r[2] -> null
h[1] -> r[1] -----> r[1] -----> r[1] -> null
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> null
      01      02      03      04      05
null <- left <- left <- left <- left <- left
                           newNode
                           _tail

rightLinks[4] = &_head[4];
rightLinks[3] = &(node 5)->right[3];
rightLinks[2] = &(node 5)->right[2];
rightLinks[1] = &(node 5)->right[1];
rightLinks[0] = &(node 5)->right[0];

// Iteration 6 (sourceNode = 6)

newNode = _createNode(sourceNode->topLevel(), sourceNode->element);

*rightLinks[0] = newNode;           // (node 5)->right[0] = &(node 6);
rightLinks[0] = &newNode->right[0]; // rightLinks[0] = &(node 6)->right[0];

*rightLinks[1] = newNode;           // (node 5)->right[1] = &(node 6);
rightLinks[1] = &newNode->right[1]; // rightLinks[1] = &(node 6)->right[1];

newNode->left = _tail;           // (node 6)->left = &(node 5);
_tail          = newNode;         // _tail          = &(node 6);

```

```

    /-- head
h[4] -> null
h[3] -----> r[3] -> null
h[2] -> -----> r[2] -----> r[2] -> null
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> null
h[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> r[0] -> null
          01      02      03      04      05      06
null <- left <- left <- left <- left <- left <- left
                           newNode
                           _tail

rightLinks[4] = &_head[4];
rightLinks[3] = &(node 5)->right[3];
rightLinks[2] = &(node 5)->right[2];
rightLinks[1] = &(node 6)->right[1];
rightLinks[0] = &(node 6)->right[0];



---


// Iteration 7 (sourceNode = 7)

newNode = _createNode(sourceNode->topLevel(), sourceNode->element);

*rightLinks[0] = newNode;           // (node 6)->right[0] = &(node 7);
rightLinks[0] = &newNode->right[0]; // rightLinks[0] = &(node 7)->right[0];

*rightLinks[1] = newNode;           // (node 6)->right[1] = &(node 7);
rightLinks[1] = &newNode->right[1]; // rightLinks[1] = &(node 7)->right[1];

*rightLinks[2] = newNode;           // (node 5)->right[2] = &(node 7);
rightLinks[2] = &newNode->right[2]; // rightLinks[2] = &(node 7)->right[2];

newNode->left = _tail;           // (node 7)->left = &(node 6);
_tail      = newNode;             // _tail      = &(node 7);

    /-- head
h[4] -> null
h[3] -----> r[3] -> null
h[2] -> -----> r[2] -----> r[2] -----> r[2] -> null
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -> null
h[0] -> r[0] -> null
          01      02      03      04      05      06      07
null <- left <- left <- left <- left <- left <- left <- left
                           newNode
                           _tail

rightLinks[4] = &_head[4];
rightLinks[3] = &(node 5)->right[3];
rightLinks[2] = &(node 7)->right[2];
rightLinks[1] = &(node 7)->right[1];
rightLinks[0] = &(node 7)->right[0];

```

```

// Iteration 8 (sourceNode = 8)

newNode = _createNode(sourceNode->topLevel(), sourceNode->element);

*rightLinks[0] = newNode;           // (node 7)->right[0] = &(node 8);
rightLinks[0] = &newNode->right[0]; // rightLinks[0] = &(node 8)->right[0];

newNode->left = _tail;           // (node 8)->left = &(node 7);
_tail           = newNode;         // _tail           = &(node 8);

      head
      /
h[4] -> null
h[3] -----> r[3] -> null
h[2] -> -----> r[2] -----> r[2] -----> r[2] -> null
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -> null
h[0] -> r[0] -> null
      01      02      03      04      05      06      07      08
null <- left <- left
                                         newNode
                                         _tail

rightLinks[4] = &_head[4];
rightLinks[3] = &(node 5)->right[3];
rightLinks[2] = &(node 7)->right[2];
rightLinks[1] = &(node 7)->right[1];
rightLinks[0] = &(node 8)->right[0];

```

```

// Iteration 9 (sourceNode = 9)

newNode = _createNode(sourceNode->topLevel(), sourceNode->element);

*rightLinks[0] = newNode;           // (node 8)->right[0] = &(node 9);
rightLinks[0] = &newNode->right[0]; // rightLinks[0] = &(node 9)->right[0];

*rightLinks[1] = newNode;           // (node 7)->right[1] = &(node 9);
rightLinks[1] = &newNode->right[1]; // rightLinks[1] = &(node 9)->right[1];

*rightLinks[2] = newNode;           // (node 7)->right[2] = &(node 9);
rightLinks[2] = &newNode->right[2]; // rightLinks[2] = &(node 9)->right[2];

*rightLinks[3] = newNode;           // (node 5)->right[3] = &(node 9);
rightLinks[3] = &newNode->right[3]; // rightLinks[3] = &(node 9)->right[3];

newNode->left = _tail;           // (node 9)->left = &(node 8);
_tail           = newNode;         // _tail           = &(node 9);

```

```

      head          Node 9's right links all point to null
      /----- (not shown due to insufficient space) \
h[4] -> null
h[3] -----> r[3] -----> r[3]
h[2] -> -----> r[2] -----> r[2] -----> r[2]
h[1] -> r[1] -----> r[1] -----> r[1] -> r[1] -> r[1] -----> r[1]
h[0] -> r[0] -> r[0]
      01      02      03      04      05      06      07      08      09
null <- left <- left
                                         newNode
                                         _tail
rightLinks[4] = &_head[4];
rightLinks[3] = &(node 9)->right[3];
rightLinks[2] = &(node 9)->right[2];
rightLinks[1] = &(node 9)->right[1];
rightLinks[0] = &(node 9)->right[0];

```

```
// After iteration 9, the sourceNode becomes null so the loop terminates
```

To implement the assignment operator (*SkipList.h*, line 69), we'll apply the same technique that we used to write *BTree*'s assignment operator (*memberFunctions_5.h*, lines 34-50):

- Clear out the left operand
- Construct a *temp* copy of the right operand
- Steal *temp*'s contents (move them to the left operand) by swapping the *_head*, *_tail*, *_size*, and *_availableNodes*

To swap the contents of *_head* and *_availableNodes* (lines 44, 47), we're calling *vector*'s *swap* method, which swaps the contents of *_head* with *temp._head*, and *_availableNodes* with *temp._availableNodes*. Although we didn't implement this method in Volume 1, the procedure is relatively easy: it simply swaps the internal pointers to the memory blocks containing the elements, along with the *_size* and *_capacity*.

Our test program (*main.cpp*) begins by constructing a list *x* and inserting the key values [1, 9] (lines 13-18). We then copy construct another list *y* from *x* (line 20). After constructing another list *z* containing the key values [10, 15], we assign the contents of *x*, [1, 9], to *z* (lines 22-27). We then print *x*, *y*, and *z* (lines 29-36), showing that they're identical.

```

// print x

head[0]    1
head[1]    1
head[2]    5
head[3]    null

```

```
node      1
  left    null
  right[0] 2
  right[1] 2

node      2
  left    1
  right[0] 3
  right[1] 3

node      3
  left    2
  right[0] 4
  right[1] 4

node      4
  left    3
  right[0] 5
  right[1] 5

node      5
  left    4
  right[0] 6
  right[1] 6
  right[2] 6

node      6
  left    5
  right[0] 7
  right[1] null
  right[2] null

node      7
  left    6
  right[0] 8

node      8
  left    7
  right[0] 9

node      9
  left    8
  right[0] null

availableNodes[0] = 5
availableNodes[1] = 0
availableNodes[2] = 0
availableNodes[3] = 1
```

```
// print y

head[0]      1
head[1]      1
head[2]      5
head[3]      null

node         1
  left       null
  right[0]   2
  right[1]   2

node         2
  left       1
  right[0]   3
  right[1]   3

node         3
  left       2
  right[0]   4
  right[1]   4

node         4
  left       3
  right[0]   5
  right[1]   5

node         5
  left       4
  right[0]   6
  right[1]   6
  right[2]   6

node         6
  left       5
  right[0]   7
  right[1]   null
  right[2]   null

node         7
  left       6
  right[0]   8

node         8
  left       7
  right[0]   9

node         9
  left       8
  right[0]   null
```

```
availableNodes[0] = 5
availableNodes[1] = 0
availableNodes[2] = 0
availableNodes[3] = 1
```

```
// print z

head[0]      1
head[1]      1
head[2]      5
head[3]      null

node        1
  left      null
  right[0]   2
  right[1]   2

node        2
  left      1
  right[0]   3
  right[1]   3

node        3
  left      2
  right[0]   4
  right[1]   4

node        4
  left      3
  right[0]   5
  right[1]   5

node        5
  left      4
  right[0]   6
  right[1]   6
  right[2]   6

node        6
  left      5
  right[0]   7
  right[1]   null
  right[2]   null

node        7
  left      6
  right[0]   8

node        8
  left      7
  right[0]   9
```

```
node      9
left     8
right[0] null

availableNodes[0] = 5
availableNodes[1] = 0
availableNodes[2] = 0
availableNodes[3] = 1
```

```
~ 1      // destroy z
~ 2
~ 3
~ 4
~ 5
~ 6
~ 7
~ 8
~ 9
~ 1      // destroy y
~ 2
~ 3
~ 4
~ 5
~ 6
~ 7
~ 8
~ 9
~ 1      // destroy x
~ 2
~ 3
~ 4
~ 5
~ 6
~ 7
~ 8
~ 9
```

All Traceables destroyed

Part 9: Polymorphism and Smart Pointers

9.1: Abstract Classes and Virtual Functions

Source files and folders

- *polymorphism*
- *Shape*
- *Shape/Subtypes*

Chapter outline

- *Pure vs. non-pure virtual functions*
- *Abstract vs. concrete classes*
- *Dynamic binding*
- *Virtual destructors*

In this section we'll expand upon the concept of inheritance, introduced in Chapter 5.1, and lay the groundwork for Part 10, in which we'll use inheritance to implement forward lists, another type of data structure.

Suppose that we're writing a program to calculate the area of a shape. The user chooses the type of shape (circle, square, or triangle), after which we prompt them for the dimensions (radius, length, or base / height) and calculate the area. One way to achieve this would be to write a separate class for each type of shape:

```
class Circle
{
public:
    std::string subtype() const;      // return "Circle";
    double area() const;             // return _pi * pow(_radius, 2);
    void getDimensions();           // Prompt user for _radius

private:
    static const double _pi;        // 3.14159
    double _radius;
};

class Square
{
public:
    std::string subtype() const;      // return "Square";
    double area() const;             // return pow(_length, 2);

// continued on next page
```

```

void getDimensions();           // Prompt user for _length

private:
    double _length;
};

class Triangle
{
public:
    std::string subtype() const;   // return "Triangle";
    double area() const;          // return (_base * _height) / 2.0;
    void getDimensions();         // Prompt user for _base and _height

private:
    double _base;
    double _height;
};

```

In *main*, we would then construct the type of shape (*Circle*, *Square*, *Triangle*) corresponding to the user's choice:

```

char subtype;      // type of shape ('c' for circle, 's' for square, 't' for
                  // triangle)

cout << "(c)ircle (s)quare (t)riangle: ";
cin >> subtype;

if (subtype == 'c')
{
    Circle c;
    c.getDimensions();
    cout << c.subtype() << " area = " << c.area();
}
else if (subtype == 's')
{
    Square s;
    s.getDimensions();
    cout << s.subtype() << " area = " << s.area();
}
else if (subtype == 't')
{
    Triangle t;
    t.getDimensions();
    cout << t.subtype() << " area = " << t.area();
}
else
{
    cout << "invalid subtype";
}

```

We could improve upon this through the use of a function template:

```
template <class Shape> // Shape = Circle, Square, Triangle, etc.
void getDimensionsAndPrintArea()
{
    Shape s;
    s.getDimensions();
    std::cout << s.subtype() << " area = " << s.area() << std::endl;
}
```

Each of the *if/ else if* blocks in *main* would then be reduced to a single line:

```
char subtype;

cout << "(c)ircle (s)quare (t)riangle: ";
cin >> subtype;

if (subtype == 'c')
    getDimensionsAndPrintArea<Circle>();
else if (subtype == 's')
    getDimensionsAndPrintArea<Square>();
else if (subtype == 't')
    getDimensionsAndPrintArea<Triangle>();
else
    cout << "invalid subtype";
```

Although this version is more compact, anytime we add a new type of shape we have to modify and recompile *main*. And the more types we add (*Sphere*, *Cube*, *Cone*, etc.), the less manageable and more bug-prone *main* becomes. Another drawback to this approach is that if we want to store multiple shapes in a single container (like a *vector*), they must all be the exact same type (all *Circles*, all *Squares*, etc.).

To overcome these limitations, we can create a separate class called *Shape* (*Shape.h*, lines 9-18):

```
class Shape
{
public:
    virtual ~Shape();

    virtual std::string subtype() const = 0;
    virtual double area() const = 0;

    virtual void getDimensions() = 0;
};
```

In lines 14-17, the *virtual* and *= 0* indicate that *subtype*, *area*, and *getDimensions* are *pure virtual functions*. A pure virtual function must be implemented by a *derived class*, also known as a *subclass* or *subtype*.

Shape's destructor (line 12) is also virtual, though it's not *pure virtual* since it's missing the *= 0*. A non-pure virtual function has an implementation, which can optionally be *overridden* (implemented) by a derived class. By default, *Shape*'s destructor simply prints the message

```
~Shape()
```

to indicate that *Shape*'s destructor was called (lines 20-23). To summarize the difference between pure virtual and non-pure virtual functions:

- A *pure* virtual function has *no* implementation; it *must* be implemented by a subclass
- A *non-pure* virtual function *has* an implementation, which can *optionally* be overridden (implemented) by a subclass

Our first derived class is *Circle* (*Circle.h*, lines 8-22):

```
class Circle : public Shape
{
public:
    virtual ~Circle();

    virtual std::string subtype() const;
    virtual double area() const;

    virtual void getDimensions();

private:
    static const double _pi;

    double _radius;
};
```

Line 8,

```
class Circle : public Shape
```

can be read as “*Circle* is a type (subtype / subclass / derived class) of *Shape*.” *Shape*, the class from which *Circle* is derived, is the *base class*. *Circle*'s two data members are:

- *_pi* (*Circle.h*, line 19), a static constant, initialized to 3.14159 (*Circle.cpp*, line 31)
- *_radius* (*Circle.h*, line 21), the radius of the *Circle*, used to calculate the area

Circle implements all 3 of *Shape*'s pure virtual methods:

- *subtype* (*Circle.h*, line 13) returns the string “*Circle*” (*Circle.cpp*, lines 13-16)
- *area* (*Circle.h*, line 14) returns the area (*_pi * _radius^2*) (*Circle.cpp*, lines 18-21)
- *getDimensions* (*Circle.h*, line 16) prompts the user to enter the *_radius* (*Circle.cpp*, lines 23-29)

Circle also overrides *Shape*'s destructor (*Circle.h*, line 11), printing the message

```
~Circle()
```

to indicate that *Circle*'s destructor was called (*Circle.cpp*, lines 8-11).

A derived class doesn't actually require the *virtual* keyword when implementing / overriding the virtual functions of its base class. In other words, lines 11-16 (*Circle.h*),

```
virtual ~Circle();

virtual std::string subtype() const;
virtual double area() const;

virtual void getDimensions();
```

could simply be written as

```
~Circle();

std::string subtype() const;
double area() const;

void getDimensions();
```

It's still considered good practice, however, to use the *virtual* keyword wherever applicable, even in derived classes. This lets other users know exactly which functions are virtual without having to look at the base class.

Our next derived class, *Square*, follows the exact same pattern as *Circle*. The only differences are:

- It has 1 data member, *_length* (the length of a side) (*Square.h*, line 19)
- *subtype* returns "Square" (*Square.cpp*, lines 13-16)
- *area* returns *_length^2* (*Square.cpp*, lines 18-21)
- *getDimensions* prompts the user to enter the *_length* (*Square.cpp*, lines 23-29)

In our third (and final) derived class, *Triangle*:

- The 2 data members are *_base* and *_height* (*Triangle.h*, lines 19-20)
- *subtype* returns "Triangle" (*Triangle.cpp*, lines 12-15)
- *area* returns $(\text{_base} * \text{_height}) / 2.0$ (*Triangle.cpp*, lines 17-20)
- *getDimensions* prompts the user to enter the *_base* and *_height* (*Triangle.cpp*, lines 22-31)

Shape is an example of an *abstract class*: because it contains one or more pure virtual functions (which have no implementation), we can't directly *instantiate* (construct) an object of type *Shape*:

```
Shape p; // compiler error: p.subtype(), p.area(), and
          // p.getDimensions() are undefined
```

Conversely, *Circle*, *Square*, and *Triangle* are *concrete classes* because they *can* be constructed directly:

```
Circle c; // ok
```

```
Square s;           // ok
Triangle t;        // ok
```

The power of virtual functions lies in the fact that a base class pointer (*Shape**) can point to any object of a derived class (*Circle*, *Square*, *Triangle*, etc.). And any virtual functions that we call through the base class pointer will automatically redirect to the derived class implementation. This “automatic redirect” is more commonly known as *dynamic dispatch*, *dynamic binding*, or *late binding*:

```
Circle c;
Square s;
Triangle t;

Shape* p;           // p is a pointer to a Shape

p = &c;             // p points to Circle c
p->getDimensions(); // calls Circle::getDimensions() on c

p = &s;             // p points to Square s
p->getDimensions(); // calls Square::getDimensions() on s

p = &t;             // p points to Triangle t
p->getDimensions(); // calls Triangle::getDimensions() on t
```

This lets us (indirectly) store multiple *Shapes* of differing subtypes, all within the same container:

```
vector<Shape*> v;

v.push_back(&c);      // v[0] points to Circle c
v.push_back(&s);      // v[1] points to Square s
v.push_back(&t);      // v[2] points to Triangle t

for (int i = 0; i != v.size(); ++i)
    cout << "area = " << v[i]->area() << endl;

// iteration 1 calls Circle::area() on c
// iteration 2 calls Square::area() on s
// iteration 3 calls Triangle::area() on t
```

Now that we've implemented a few different types of *Shape*, let's write a pair of stand-alone functions to create a *Shape* based on the user's choice. The first of these (*getSubtype.h*, line 6),

```
char getSubtype();
```

prompts the user to enter a character (*c* for circle, *s* for square, *t* for triangle, or *d* for done), and returns the character entered by the user (*getSubtype.cpp*, lines 7-17). The second (*createShape.h*, line 8),

```
Shape* createShape(char subtype);
```

creates a new (dynamically-allocated) *Shape* of the desired *subtype*, and returns a pointer to the new *Shape*. If the *subtype* doesn't correspond to a valid *Shape*, the function returns a null pointer (*createShape.cpp*, lines 8-18).

Putting it all together, our test program (*main.cpp*) begins by constructing a *vector*<*Shape**> *shapes* (line 12). In each iteration of the loop,

- We call *getSubtype* to get the user's choice of *Shape* (*c*, *s*, or *t*, for *Circle*, *Square*, or *Triangle*) (line 16).
- If the user entered *d* (for “done”), we exit the loop (lines 18-19); otherwise, we call *createShape* (line 21), which returns a pointer to the newly created *Shape* (or null, if the user entered an invalid *subtype*).
- If we have a valid *Shape* we call *getDimensions* (lines 23-25), which automatically redirects to the subclass implementation. We then store (a pointer to) the new *Shape* in *shapes* (line 26), and begin the next iteration.
- If we don't have a valid *Shape* (because the user entered a character other than *c*, *s*, or *t*), we print the message “invalid subtype” before beginning the next iteration (lines 28-31).

After creating all of the desired *Shapes* and getting their dimensions, we traverse *shapes*. In each iteration of the loop, we print the *subtype* and *area* of the current *Shape* (lines 35-36), where *shape* is a *const* reference to the current element (*Shape**). Recall that we discussed this type of loop (range-based *for* loop) at the end of Chapter 4.2. Using a traditional-style *for* loop, we could've written this as

```
for (vector<Shape*>::const_iterator shape = shapes.begin();
    shape != shapes.end();
    ++shape)
{
    cout << (*shape)->subtype() << " area = " << (*shape)->area() << endl;
}
```

where *shape* is a *const_iterator* to the current element (*Shape**), and **shape* returns a *const* reference to the current element (*Shape**). We then apply the member access operator (->) to call the *subtype* and *area* methods on the *Shape* itself.

Before exiting *main*, we need to manually destroy all of the *Shapes* since they were dynamically allocated via *new*. To do this we simply traverse *shapes* once more, deleting the referent object of each pointer (lines 39-40). This is where *Shape*'s virtual destructor comes in: it automatically redirects to the appropriate derived class destructor (*~Circle()*, *~Square()*, or *~Triangle()*). The derived class destructor first executes its own code, then automatically calls the base class destructor (*~Shape()*). If, for example, we write

```
Shape* p;

p = new Circle();           // automatically redirects to Circle's destructor
delete p;                  // Circle's destructor first executes its own code,
                           // then automatically calls Shape's destructor

p = new Square();           // automatically redirects to Square's destructor
delete p;
```

```
// Square's destructor first executes its own code,  
// then automatically calls Shape's destructor  
  
p = new Triangle();  
delete p; // automatically redirects to Triangle's destructor  
  
// Triangle's destructor first executes its own code,  
// then automatically calls Shape's destructor
```

the resultant output is

```
~Circle()
~Shape()

~Square()
~Shape()

~Triangle()
~Shape()
```

As a general rule, if a class has any virtual functions, then its destructor should also be virtual. This ensures that the appropriate redirect can be made when destroying a derived class object (*Circle*, *Square*, *Triangle*, etc.) through a base class pointer (*Shape**).

The reason that all member functions aren't simply virtual by default is that dynamic dispatch does incur a small overhead, and one of the chief design goals of C++ was to give the programmer as much choice as possible.

One possible run of our test program is

```

~Circle()           // delete shapes[1]
~Shape()

~Square()          // delete shapes[2]
~Shape()

```

Creating a derived class from a base class is known as *inheritance* because the derived class is said to *inherit* the traits / functionality of its base (parent) class. It's also commonly known as *polymorphism* (literally, “many forms”) because a single base type (*Shape*) can give rise to many different subtypes (*Circle*, *Square*, *Triangle*, etc.).

Polymorphism helps us separate the parts of a program that change from the parts that stay the same, thereby making it easier to add or modify functionality. We could add support for cones, for example, by simply creating a new *Cone* class, derived from *Shape*. The only modifications to existing code would be:

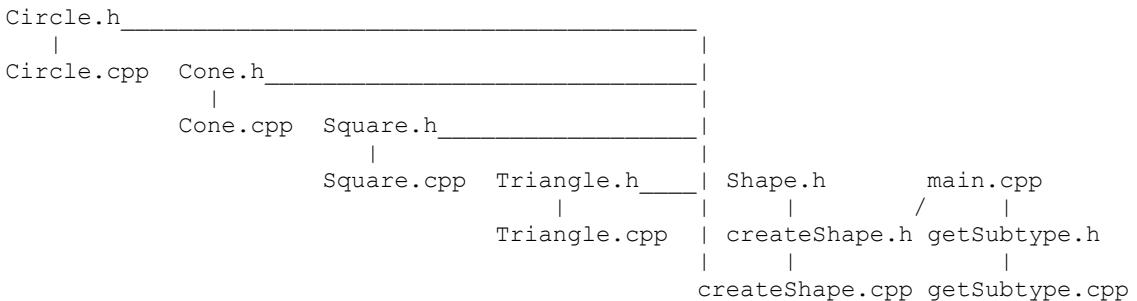
- Updating the user prompt in *getSubtype.cpp* (line 13):

```
cout << "\n(c)ircle co(n)e (s)quare (t)riangle (d)one: ";
```

- Including *Cone.h* in *createShape.cpp* (line 3) and adding the appropriate *else if* branch (lines 12-13):

```
else if (subtype == 'n')
    return new Cone();
```

main.cpp wouldn't even need to be recompiled, since it only *#includes* *Shape.h*, *createShape.h*, and *getSubtype.h*, as shown in the following dependency diagram:



Inheritance and polymorphism have a wide variety of applications in software engineering, and there are entire books dedicated to the subject. This chapter was by no means intended to be a comprehensive resource; as mentioned earlier, our main objective was to lay the groundwork for Part 10, in which we'll use inheritance to implement forward lists.

9.2: Introducing the *SharedPtr* Class

Source files and folders

- *SharedPtr*

Chapter outline

- *Implementing a shared pointer class for automatic resource management*

The more dynamically-allocated objects we create, the more likely we are to either forget to delete them (causing memory leaks), or accidentally delete them more than once (causing undefined behavior). We can, however, automate the deletion process through the use of shared pointers.

A *shared pointer* is an object that contains a pointer to a dynamically-allocated resource (object) *r*. Any number of shared pointers can *share ownership* of (simultaneously point to) *r*. When a shared pointer goes out of scope, its destructor checks to see if any other owners still exist, and deletes *r* only if no other owners remain.

Shared pointers belong to a broader class of objects known as *smart pointers*, which get their name from the added functionality they provide over *raw* (normal) *pointers*.

A *SharedPtr<T>* (“shared pointer to a dynamically-allocated object of type T”) contains 2 data members (*SharedPtr.h*):

- *_element*, a pointer to the dynamically-allocated element (managed resource / owned object) (lines 12, 32)
- *_totalOwners*, a pointer to the *reference count* (the total number of *SharedPtr* objects pointing to (owning) the same *_element*) (lines 30, 33).

The default constructor (line 14) initializes the *_element* to null, and the *_totalOwners* to 1 (lines 36-42).

The *element* constructor (line 15) constructs a *SharedPtr* from the given *element* (raw pointer), initializing the *_totalOwners* to 1 (lines 44-50).

The copy constructor (line 16) constructs a *SharedPtr* pointing to the same element as that of the *source SharedPtr*, thereby increasing the number of *_totalOwners* by 1 (lines 52-58).

The destructor (line 18) destroys the *SharedPtr*; thereby decreasing the number of *_totalOwners* by 1 (line 63). If no other owners remain, it deletes the element and reference count (lines 65-69).

The *get* method (line 20) returns a raw pointer to the element (lines 72-76), and *use_count* (line 24) returns the reference count (lines 96-100).

The dereference operator (line 22) returns a reference to the element (lines 84-88), while the member access operator (line 23) returns a raw pointer to the element (lines 90-94).

The boolean operator (line 21) converts the *SharedPtr* to a *bool* value by simply returning the *_element* pointer, which is implicitly converted to a *bool* value (lines 78-82). This allows for a certain type of shorthand notation, which works with raw pointers, to also work with *SharedPtrs*. Given

```
Shape* p = nullptr;
Shape* q = new Circle();
```

for example, the expressions

```
if (p)
if (!p)
```

are equivalent to

```
if (p != nullptr)      // false
if (p == nullptr)      // true
```

and the expressions

```
if (q)
if (!q)
```

are equivalent to

```
if (q != nullptr)      // true
if (q == nullptr)      // false
```

The *reset* method (line 27) resets the *SharedPtr* to a default-constructed state (null *_element*, reference count of 1). We begin by decrementing the reference count (line 123).

- If there are no owners remaining, we delete the element, reset the reference count to 1 (lines 125-129), and set our own *_element* to null (line 135):

```
SharedPtr p ----> element
                    totalOwners(1)

p.reset();

SharedPtr p ----> nullptr
                    totalOwners(1)
```

- Otherwise, we leave the element and reference count intact for the remaining owners, create a new reference count of 1, and set our own *_element* to null (lines 130-135):

```

    SharedPtr q
    /_____
SharedPtr p -----> element <---- SharedPtr r
                           totalOwners(3)

p.reset();

    SharedPtr q
    /_____
           element <---- SharedPtr r
                           totalOwners(2)

SharedPtr p -----> nullptr
                           totalOwners(1)

```

The assignment operator (line 26) points the left-hand operand to the same element as the right-hand operand. We begin by incrementing the right-hand reference count and decrementing the left-hand reference count (lines 105-106).

- If there are no remaining owners of the left-hand element, we delete the left-hand element and reference count (lines 108-112), then point the left-hand operand to the right-hand element and reference count (lines 114-115).

```

SharedPtr a -> element      SharedPtr x -> element
                           totalOwners(1)          totalOwners(1)

a = x;

SharedPtr a ____ \
SharedPtr x -> element
                           totalOwners(2)

```

- Otherwise, we leave the left-hand element and reference count intact for the remaining owners, then point the left-hand operand to the right-hand element and reference count (lines 114-115).

```

SharedPtr a ____ \           SharedPtr c
                           /_____
SharedPtr b -> element <- SharedPtr d      SharedPtr x -> element
                           totalOwners(4)          totalOwners(1)

a = x;

                           /_____
                           SharedPtr c      SharedPtr a ____ \
SharedPtr b -> element <- SharedPtr d      SharedPtr x -> element
                           totalOwners(3)          totalOwners(2)

```

Our test program (*main.cpp*) begins by initializing a *SharedPtr a* to a new *Circle* (lines 10, 20), and calling *getDimensions* (line 21):

```
a --> Circle
    totalOwners(1)
```

We then create another *SharedPtr* *b* as a copy of *a* (line 23),

```
a --> Circle <-- b
    totalOwners(2)
```

and call the function *printUseCountAndArea* (lines 12, 24), which simply prints the *use_count* and *area* of *a* and *b* (lines 39-49). Because we're passing *a* and *b* by reference, no new *SharedPtrs* are created, leaving the *use_count* unchanged.

We then reassign *a* to take ownership of a new *Square*, *getDimensions* of the *Square*, and *printUseCountAndArea* (lines 26-28):

```
Circle <-- b      a --> Square
totalOwners(1)      totalOwners(1)
```

Lastly, we reassign *b* to take ownership of a new *Triangle*, *getDimensions* of the *Triangle*, and *printUseCountAndArea* (lines 30-32). The *Circle*, which no longer has any owners, is deleted:

```
a --> Square          b --> Triangle
totalOwners(1)          totalOwners(1)
```

At the end of *main*, *b*'s destructor deletes the *Triangle* and its reference count, then *a*'s destructor deletes the *Square* and its reference count.

One possible run generates the output

```
radius: 5
a.use_count() = 2
a->area() = 78.5397
// ShapePtr a(new Circle());
// a->getDimensions();
// ShapePtr b(a);
// Circle has 2 owners (a and b)

b.use_count() = 2
b->area() = 78.5397
// Circle has 2 owners (a and b)
length: 6
// a = new Square();
// a->getDimensions();

a.use_count() = 1
a->area() = 36
// Square has 1 owner (a)

b.use_count() = 1
b->area() = 78.5397
// Circle has 1 owner (b)
~Circle()           // b = new Triangle(); // deletes Circle
~Shape()

base: 7
height: 8
// b->getDimensions();
```

```

a.use_count() = 1           // Square has 1 owner (a)
a->area()      = 36

b.use_count() = 1           // Triangle has 1 owner (b)
b->area()      = 28

~Triangle()          // b's destructor (deletes Triangle)
~Shape()

~Square()            // a's destructor (deletes Square)
~Shape()

```

SharedPtrs can also be used with containers, as in

```

vector<SharedPtr<Shape>> v;

v.push_back(new Circle());    // v[0]
v.push_back(new Square());    // v[1]
v.push_back(new Triangle());  // v[2]

```

or with algorithms like *std::swap*:

```

swap(v.front(), v.back());    // v[0] now points to the Triangle,
                             // v[2] now points to the Circle

```

The Standard Library's shared pointer class, *std::shared_ptr*, is included in the *<memory>* header, and provides all the same functionality we implemented and more.

Part 10: Forward Lists

10.1: Introducing the *ForwardList* Class

Source files and folders

- *ForwardList/I*
- *ForwardList/common/ForwardListNode.h*
- *ForwardList/common/memberFunctions_I.h*

Chapter outline

- *Overview and terminology*
- *Using inheritance to implement nodes*
- *Implementing push_front*

The *List* class that we implemented in Volume 1 is a *doubly-linked list*, in which each node contains two links: one to the next node in the sequence and one to the previous node. In a *forward list*, more commonly known as a *singly-linked list*, each node contains a link to the next node only:

```
// Forward list containing the elements {0, 1, 2, 3, 4}

BH
next -> next -> next -> next -> next -> next -> null
    00      01      02      03      04
```

The node containing the first element (node 0 in the above diagram) is the *head*. *BH* stands for *before-head*, which is simply the node before the head. The before-head is unique in that unlike all the other nodes, it doesn't contain an element (only a *next* link to the head).

Forward lists are more efficient than doubly-linked lists in both space and time. The above list, for example, uses a total of 6 pointers, while its doubly-linked counterpart requires 12 (5 nodes x 2 links per node, plus the head and tail pointers). Having fewer pointers to update also allows for faster insertion and removal.

This efficiency comes at the cost of versatility, however, as we can't traverse the list backwards. Additionally, given a pointer to a node *n*:

- We can insert a node after *n*, but not before *n*. Given a pointer to node 2, for example, we can't insert a node before node 2 since that would require updating node 1's *next* link (to point to node 3) – and we can't access node 1 from node 2. We can only insert a node after node 2.
- We can erase the node after *n*, but not *n* itself. Given a pointer to node 3, for example, we

can't actually erase node 3 since that would require updating node 2's *next* link (to point to node 4) – and we can't access node 2 from node 3. We can only erase the node after node 3.

The first step to creating our forward list class is to implement the node class. As mentioned above, there are two types of nodes:

- Those that contain a *next* link only (the before-head)
- Those that contain a *next* link and an element (all other nodes)

We can express this relationship using inheritance. A *ForwardListNode* contains a link to the *next ForwardListNode*, default initialized to null (*ForwardListNode.h*, lines 6-11, 21-25). A *ForwardListElementNode* is a subtype of *ForwardListNode*: it has a *next* link (inherited from its base class) and an *element* of type *T* (lines 13-19). The constructor (line 16) automatically calls the base class constructor, after which we initialize the *element* according to the given argument (lines 27-32).

Within the *ForwardList* class, we'll shorten the names of the node classes to *Node* and *ElementNode* respectively (*ForwardList.h*, lines 33-34). The *_beforeHead* (line 42) is just a *Node*, while the nodes containing the elements are *ElementNodes*, managed by the allocator *_alloc* (line 43):

Node	BH												
	next	->	null										
ElementNode			00	01	02	03	04						

Because every *ElementNode* is a (type of) *Node*, we can use a *Node** (base class pointer) to access the *_beforeHead* as well as the *ElementNodes*. We'll use this technique to implement *ForwardList*'s iterators later on.

In the default constructor (*ForwardList.h*, line 23), we don't need to perform any explicit initialization (*memberFunctions_1.h*, lines 3-7), since *_beforeHead*'s default constructor initializes its *next* pointer to null.

empty (*ForwardList.h*, line 26) determines if the list is empty by checking for the existence of a head node (*memberFunctions_1.h*, lines 15-19).

To minimize the space requirement, *ForwardList* doesn't keep track of its own size (total number of elements), but we can (and will) add that functionality later.

_downcast (*ForwardList.h*, line 36) converts the given *Node* pointer *n* to an *ElementNode* pointer by performing a *static_cast* (*memberFunctions_1.h*, lines 42-47):

```
return static_cast<ElementNode*>(n);
```

This is called a *downcast* because we're casting a base class pointer (*Node**) down to a derived class pointer (*ElementNode**). Unlike *upcasts* (converting a derived class pointer up to a base class pointer, performed implicitly), downcasts must be performed explicitly as a safety measure:

```

void f(Circle* c)
{
    Shape* s = c;           // implicit upcast from Circle* to Shape* (ok):
                           // every Circle is guaranteed to be a Shape, so we can
                           // safely treat c as a Shape*

    s->getDimensions();   // Shape::getDimensions
}

void g(Shape* s)
{
    Circle* c = s;         // implicit downcast from Shape* to Circle* (error):
                           // not every Shape is guaranteed to be a Circle (s
                           // could actually point to a Square, Triangle, etc.),
                           // so we can't automatically treat s as a Circle*
}

void h(Shape* s)
{
    Circle* c = static_cast<Circle*>(s); // explicit downcast from Shape* to
                                             // Circle* (we're telling the
                                             // compiler "trust us, we know that
                                             // s really does point to a Circle,
                                             // so let us treat it like one")

    c->getDimensions();          // Circle::getDimensions
                                 // ok, as long as s really does point
                                 // to a Circle; but if s actually
                                 // points to some other derived type
                                 // (Square, Triangle, etc.), we're in
                                 // trouble (undefined behavior)
}

```

front (*ForwardList.h*, line 29) returns a reference to the head element. The expression (*memberFunctions_1.h*, line 30)

```
_downcast(_beforeHead.next) // static_cast<ElementNode*>(_beforeHead.next)
```

tells the compiler “Trust us, we know that *_beforeHead.next* (a *Node**) really does point to an *ElementNode*, so let us treat it like one.” We then use the returned pointer (*ElementNode**) to access the head node’s *element*. The *const* version of *front* (*ForwardList.h*, line 27) uses the *const_cast* technique to call the non-*const* version (*memberFunctions_1.h*, lines 21-25).

ForwardList doesn’t have a pointer to keep track of the tail node, so it doesn’t have a *back* or *push_back* method.

_createElementNode (*ForwardList.h*, line 38) creates a new *ElementNode* and returns a pointer to it (*memberFunctions_1.h*, lines 49-57).

_destroyElementNode (*ForwardList.h*, line 39) destroys the given *ElementNode n* and frees the associated memory block (*memberFunctions_1.h*, lines 59-64).

`_destroyAllElementNodes` (*ForwardList.h*, line 40) destroys all the *ElementNodes*. We begin at the head (*memberFunctions_1.h*, line 69), and in each iteration of the loop, we get a pointer to the next *ElementNode*, destroy the current *ElementNode* *n*, then proceed to the next *ElementNode* (lines 73-75). This is the exact same procedure that we used for *List* and *SkipList*.

The destructor (*ForwardList.h*, line 24) simply calls `_destroyAllElementNodes` (*memberFunctions_1.h*, lines 9-13). We don't need to manually destroy the `_beforeHead` since it wasn't dynamically-allocated.

`push_front` (*ForwardList.h*, line 30) inserts the *newElement* at the front of the list. We create a new *ElementNode*, attach its *next* link to the head, then attach the before-head's *next* link to the new node (*memberFunctions_1.h*, lines 33-40):

```

BH
next -> null

push_front(0);

newNode = _createElementNode(0); // create node 0;

newNode->next = _beforeHead.next; // (node 0)->next = null;
_beforeHead.next = newNode; // _beforeHead.next = (node 0);

BH      NN
next -> next -> null
      00


---


push_front(1);

newNode = _createElementNode(1); // create node 1;

newNode->next = _beforeHead.next; // (node 1)->next = (node 0);
_beforeHead.next = newNode; // _beforeHead.next = (node 1);

BH      NN
next -> next -> next -> null
      01      00


---


push_front(2);

newNode = _createElementNode(2); // create node 2;

newNode->next = _beforeHead.next; // (node 2)->next = (node 1);
_beforeHead.next = newNode; // _beforeHead.next = (node 2);

BH      NN
next -> next -> next -> next -> null
      02      01      00

```

Our test program (*main.cpp*) constructs a *ForwardList<Traceable<int>>* *f* (line 11). In each iteration

of the loop, we *push_front* a new element (0, 1, 2, 3, 4) and print the newly inserted (*front*) element (lines 13-17). The resultant output is

```
v 0          // i = 0
c 0
c 0
~ 0
~ 0
```

Inserted 0

```
v 1          // i = 1
c 1
c 1
~ 1
~ 1
```

Inserted 1

```
v 2          // i = 2
c 2
c 2
~ 2
~ 2
```

Inserted 2

```
v 3          // i = 3
c 3
c 3
~ 3
~ 3
```

Inserted 3

```
v 4          // i = 4
c 4
c 4
~ 4
~ 4
```

Inserted 4

```
~ 4          // destructor
~ 3
~ 2
~ 1
~ 0
```

All Traceables destroyed

10.2: Erasing the Front Element

Source files and folders

- *ForwardList/2*
- *ForwardList/common/memberFunctions_2.h*

Chapter outline

- *Implementing pop_front and clear*

Like *push_front*, *ForwardList*'s *pop_front* and *clear* methods (*ForwardList.h*, lines 31-32) are fairly simple. To remove the front element, we detach the head node (by attaching the before-head's *next* link to the head's right neighbor) and destroy it (*memberFunctions_2.h*, lines 6-9):

```

BH
next -> next -> next -> null
      01          00

pop_front();

head = _downcast(_beforeHead.next); // head = (node 1);
_beforeHead.next = head->next;    // _beforeHead.next = (node 0);

BH
next -> next -> null
      00

_destroyElementNode(head);           // destroy node 1


---


pop_front();

head = _downcast(_beforeHead.next); // head = (node 0);
_beforeHead.next = head->next;    // _beforeHead.next = null;

BH
next -> null

_destroyElementNode(head);           // destroy node 0

```

To *clear* the list, we simply *_destroyAllElementNodes* and set the before-head's *next* link to null (*memberFunctions_2.h*, lines 15-16).

Our test program (*main.cpp*) constructs two lists *f* and *g*, containing the elements {4, 3, 2, 1, 0} (lines 11-18). We then repeatedly call *pop_front* on *f* until the list is empty, printing the element we're about to erase each time (lines 22-27).

Once *f* is empty, we call *clear* on *g* (line 29). The resultant output is

```
Erasing 4...    // iteration 1
~ 4            // f.pop_front();

Erasing 3...    // iteration 2
~ 3            // f.pop_front();

Erasing 2...    // iteration 3
~ 2            // f.pop_front();

Erasing 1...    // iteration 4
~ 1            // f.pop_front();

Erasing 0...    // iteration 5
~ 0            // f.pop_front();

~ 4            // g.clear();
~ 3
~ 2
~ 1
~ 0

All Traceables destroyed
```

10.3: Implementing the Iterators

Source files and folders

- *ForwardList/3*
- *ForwardList/common/ForwardListIter.h*
- *ForwardList/common/memberFunctions_3.h*

Chapter outline

- *Implementing ForwardList::iterator and ForwardList::const_iterator*
- *Implementing the before_begin, begin, and end methods*

A *ForwardList* iterator contains a pointer to a *Node* *_n* (*ForwardListIter.h*, line 31, 36), initialized via the private constructor (lines 34, 92-97). This constructor will be used by *ForwardList* member functions such as *begin* and *end*, which is why *ForwardList* needs friend privileges (line 12). The default constructor (line 20) leaves *_n* uninitialized (lines 39-43).

The comparison operators (lines 22-23) determine whether the iterator points to the same element as the right operand by checking whether they point to the same node (lines 45-57).

The dereference / member access operators (lines 24-25) return a reference / pointer to the referent element by downcasting *_n* to an *ElementNode** and accessing its *element* (lines 32, 59-71).

The increment operators (lines 27-28) point to the next element by simply pointing *_n* to the next node in the sequence (lines 73-79). Since we can't traverse the list backwards, *ForwardListIter*'s *iterator_category* is *forward_iterator_tag* (line 18). As discussed in Chapter 5.1, a *forward iterator* is an iterator that can read / write its referent element more than once, and move forward one element at a time.

In order to access the private *typedefs* *ForwardList::Node* and *ForwardList::ElementNode* (*ForwardListIter.h*, lines 31-32), *ForwardListIter* needs friend access to *ForwardList* (*ForwardList.h*, line 18).

ForwardList::iterator is an alias of *ForwardListIter<ForwardList>* (line 23), while *ForwardList::const_iterator* is an alias of *ConstIter<ForwardList>* (line 20).

Among all the data structures we've developed so far, *ForwardList* is unique in that it has a *before_begin* method (lines 34, 39), which returns an iterator to the (nonexistent) before-the-first element. This is simply an iterator to *_beforeHead* (*memberFunctions_3.h*, lines 3-8, 22-26). It can't be dereferenced since *_beforeHead* is just a *Node* (not an *ElementNode*), but it can be incremented to point to the *front* element (head node). *before_begin* is only meant to be used with *ForwardList*'s insert and erase methods, which we'll implement in the next chapter.

The *begin* and *end* methods (*ForwardList.h*, lines 35-36, 40-41) return iterators to the front and one-past-the-last elements, respectively (*memberFunctions_3.h*, lines 10-20, 28-38).

Our test program (*main.cpp*) constructs a list *f*, containing the elements {4, 3, 2, 1, 0} (lines 14-17). We then increment the iterator returned by *before_begin* (to point to the front element) and print the corresponding value (line 19).

To test *ForwardList::const_iterator*, we pass *f* to *printContainer*, which calls the *const* versions of *begin* and *end* (line 21). To test *ForwardList::iterator*, we pass the non-*const* versions of *begin* and *end* to *printSequence* (line 22). The resultant output is

```
4           // * (++f.before_begin())
4 3 2 1 0   // printContainer(f);
4 3 2 1 0   // printSequence(f.begin(), f.end());
```

10.4: Inserting and Erasing Elements in the Middle

Source files and folders

- *ForwardList/4*
 - *ForwardList/common/memberFunctions_4.h*

Chapter outline

- *Implementing insert after and erase after*

Recall from Chapter 10.1 that given a pointer to node n ,

- We can insert a node after n (but not before n)
 - We can erase the node after n (but not n itself)

So instead of the standard *insert* and *erase* methods, *ForwardList* has *insert after* and *erase after*:

- *insert_after* (*ForwardList.h*, line 43) inserts the *newElement* after the *insertionPoint*, and returns an iterator to the newly inserted element
 - *erase_after* (line 44) removes the element after the *erasurePoint*, and returns an iterator to the next element in the list

To implement `insert_after`, we begin by creating a new node and retrieving the `Node` pointer from the `insertionPoint` (`memberFunctions_4.h`, lines 7-8). We then attach the new node to the insertion point's right neighbor, attach the insertion point to the new node, and return an iterator to the new node (lines 10-13):

```

BH      IP
next -> next -> null
02

insert_after(insertionPoint, 0);

newNode = _createElementNode(newElement); // create node 0
_insertionPoint = insertionPoint._i._n; // _insertionPoint = (node 2);

newNode->next = _insertionPoint->next; // (node 0)->next = null;
_insertionPoint->next = newNode; // (node 2)->next = (node 0);

BH      IP      NN
next -> next -> next -> null
02      00

```

```

BH      IP
next -> next -> next -> null
02      00

insert_after(insertionPoint, 1);

newNode = _createElementNode(newElement); // create node 1
_insertionPoint = insertionPoint._i._n; // _insertionPoint = (node 2);

newNode->next = _insertionPoint->next; // (node 1)->next = (node 0);
_insertionPoint->next = newNode; // (node 2)->next = (node 1);

BH      IP      NN
next -> next -> next -> next -> null
02      01      00

```

To implement *erase_after*, we begin by retrieving the *Node* pointer from the *erasurePoint* and downcasting its right neighbor to an *ElementNode* pointer named *trash* (*memberFunctions_4.h*, lines 20-21); this is the node that we want to erase. We then detach *trash* from the list (by attaching the erasure point to *trash*'s right neighbor), after which we destroy *trash* and return an iterator to the next node (lines 23-26):

```

BH          EP
next -> next -> next -> next -> next -> null // EP = erasurePoint
03          02          01          00

erase_after(erasurePoint);

_eraseasurePoint = erasurePoint._i._n;
_trash = _downcast(_erasurePoint->next); // trash = (node 1);

BH          EP          T
next -> next -> next -> next -> next -> null // T = trash
03          02          01          00

```

```

_erasurePoint->next = trash->next;           // (node 2)->next = (node 0);
_destroyElementNode(trash);                  // destroy node 1

    BH          EP
next -> next -> next -> next -> null
    03          02          00

return _erasurePoint->next;                  // return node 0;


---


    BH          EP
next -> next -> next -> next -> null
    03          02          00

erase_after(erasurePoint);

    _erasurePoint = erasurePoint._i._n;
    trash = _downcast(_erasurePoint->next); // trash = (node 0);

    BH          EP      T
next -> next -> next -> next -> null
    03          02          00

    _erasurePoint->next = trash->next;           // (node 2)->next = null;
    _destroyElementNode(trash);                  // destroy node 0

    BH          EP
next -> next -> next -> null
    03          02

return _erasurePoint->next;                  // return null;


---


    BH
next -> next -> next -> null
    EP      03      02

erase_after(before_begin());

    _erasurePoint = erasurePoint._i._n;
    trash = _downcast(_erasurePoint->next); // trash = (node 3);

    BH      T
next -> next -> next -> null
    EP      03      02

    _erasurePoint->next = trash->next;           // _beforeHead.next = (node 2);
    _destroyElementNode(trash);                  // destroy node 3

    BH
next -> next -> null
    EP      02

return _erasurePoint->next;                  // return node 2;

```

Our test program (*main.cpp*) constructs a list *f* containing the elements {4, 3, 2, 1, 0} (lines 14-17). We then demonstrate the following operations:

- insert 6 after *before_begin* (insert 6 at the front) (line 23)
- insert 5 after *begin* (insert 5 after the front element, 6) (line 28)
- erase the element after *begin* (erase the second element, 5) (line 33)
- erase the element after *before_begin* (erase the front element, 6) (line 38)

The resultant output is

```

4 3 2 1 0      // f

v 6            // f.insert_after(f.before_begin(), 6);
c 6
c 6
~ 6
~ 6

6 4 3 2 1 0

v 5            // f.insert_after(f.begin(), 5);
c 5
c 5
~ 5
~ 5

6 5 4 3 2 1 0

~ 5            // f.erase_after(f.begin());

6 4 3 2 1 0

~ 6            // f.erase_after(f.before_begin());

4 3 2 1 0

~ 4
~ 3
~ 2
~ 1
~ 0

All Traceables destroyed

```

10.5: Implementing Copy and Assignment

Source files and folders

- *ForwardList/5*
- *ForwardList/common/memberFunctions_5.h*

Chapter outline

- *Implementing ForwardList's copy constructor and assignment operator*

To implement the copy constructor and assignment operator, we'll first write a separate member function (*ForwardList.h*, line 57)

```
void _copyElementsFrom(const ForwardList& source);
```

which copies all elements from the *source* list to *this* list (the list on which the function was called). The copied elements are inserted after the before-head, so *this* list must be empty.

We begin by initializing a pointer, *leftNeighbor*, to the before-head of the destination list (*memberFunctions_5.h*, line 21); this node will become the left neighbor of the current copied node. We then traverse the *source* list one *element* at a time (line 23). In each iteration we create a new node containing the current *element* and attach it to its *leftNeighbor* (line 25), after which the *leftNeighbor* becomes the new node (line 26):

```

BH
source    next -> next -> next -> next -> null
          02      01      00

BH
this      next -> null

_copyElementsFrom(source);

leftNeighbor = &_beforeHead;

BH
this      next -> null                                // LN = leftNeighbor
          LN

// iteration 1 (element = 2)

leftNeighbor->next = _createElementNode(element);

BH
this      next -> next -> null
          LN      02

```

```

leftNeighbor = leftNeighbor->next;

      BH      LN
this     next -> next -> null
          02



---


// iteration 2 (element = 1)

leftNeighbor->next = _createElementNode(element);

      BH      LN
this     next -> next -> next -> null
          02      01

leftNeighbor = leftNeighbor->next;

      BH      LN
this     next -> next -> next -> null
          02      01



---


// iteration 3 (element = 0)

leftNeighbor->next = _createElementNode(element);

      BH      LN
this     next -> next -> next -> next -> null
          02      01      00

leftNeighbor = leftNeighbor->next;

      BH      LN
this     next -> next -> next -> next -> null
          02      01      00

```

The copy constructor (*ForwardList.h*, line 31) creates a copy of the *source* list by simply calling *_copyElementsFrom* (*memberFunctions_5.h*, line 6). The assignment operator (*ForwardList.h*, line 44) clears the left operand then copies the elements from the right operand (*memberFunctions_5.h*, lines 12-13).

Our test program (*main.cpp*) begins by constructing a list *f* containing the elements {4, 3, 2, 1, 0} (lines 14-19). We then copy construct a list *g* from *f* (line 21), and construct another list *h* containing the elements {6, 5} (lines 26-28). After assigning *f* to *h* (line 31), *h* contains the same elements as *f*, {4, 3, 2, 1, 0}. The resultant output is

```

// f contains {4, 3, 2, 1, 0}
// ForwardList g(f);
4 3 2 1 0 // printContainer(g);

// h contains {6, 5}
// h = f;

```

```
4 3 2 1 0      // printContainer(h);  
~ 4            // destroy h  
~ 3  
~ 2  
~ 1  
~ 0  
~ 4            // destroy g  
~ 3  
~ 2  
~ 1  
~ 0  
~ 4            // destroy f  
~ 3  
~ 2  
~ 1  
~ 0
```

All Traceables destroyed

The Standard Library's *forward_list* class provides the same functionality as *ForwardList*, and is available via the <*forward_list*> header.

10.6: Introducing the *ForwardListSize* Class

Source files and folders

- *ForwardListSize/I*
- *ForwardListSize/common*

Chapter outline

- *Tracking the size of a forward list*

As mentioned in Chapter 10.1, *ForwardList* doesn't have a *_size* member in order to minimize its memory footprint; the only way to know the number of elements is to measure the distance from *begin* to *end*, one element at a time.

To retain the benefits of *ForwardList* while keeping track of the size, we'll create a new class. A *ForwardListSize<T>* contains a *ForwardList<T>* and a *_size* member (*ForwardListSize.h*, lines 12, 45, 44).

The default constructor (line 24) initializes the *_size* to 0 (*memberFunctions_I.h*, lines 3-8). The *_list* is initialized via its own default constructor.

empty, *before_begin*, *begin*, *end*, and *front* (*ForwardListSize.h*, lines 26, 28-31, 33-36) simply wrap the corresponding method of the *_list* (*memberFunctions_I.h*, lines 10-14, 22-69).

size (*ForwardListSize.h*, line 27) returns the *_size* (*memberFunctions_I.h*, lines 16-20).

insert_after, *erase_after*, *push_front*, *pop_front*, and *clear* (*ForwardListSize.h*, lines 37-41) call the corresponding function on the *_list* and update the *_size* accordingly (*memberFunctions_I.h*, lines 71-108).

We'll use *ForwardListSize* to implement hash tables in Part 12.

Part 11: Bit Representation and Bitwise Operations

11.1: Binary Numbers, Characters, and Strings

Chapter outline

- Calculating the value of a binary number
- Bits and bytes
- Bit representation of characters and strings, using ASCII codes
- C-strings vs. `std::strings`
- `std::string's size, length, c_str, and data methods`

In the next section, we'll study a new type of data structure called a *hash table*. Like red-black trees, B-trees, and skip lists, hash tables provide efficient access to collections of key-mapped pairs. Our hash table implementation, however, will utilize some lower-level concepts that we've never discussed before: bit representation and bitwise operations.

The number system most commonly used in everyday life is the *decimal*, or *base-10* system. In the base-10 system, each digit of a given number has 1 of 10 possible values, $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The procedure for calculating the value of a base-10 number is:

- Count the number of digits d to the left of the *radix point* (decimal point)
- Multiply the leftmost digit by 10^{d-1} , the next digit by 10^{d-2} , the next digit by 10^{d-3} , etc.
- Take the sum of those values

The number

```
6735      // 4 digits to the left of the radix point (d = 4)
```

for example, denotes

$$\begin{array}{r} 6 \times 10^3 \\ + 7 \times 10^2 \\ + 3 \times 10^1 \\ + 5 \times 10^0 \end{array}$$

which becomes

$$\begin{array}{r} 6 \times 1000 \\ + 7 \times 100 \\ + 3 \times 10 \\ + 5 \times 1 \end{array}$$

and

$$\begin{array}{r}
 6000 \\
 + 700 \\
 + 30 \\
 + 5 \\
 \hline
 \end{array}$$

6735

The number

3.14159 // 1 digit to the left of the radix point ($d = 1$)

denotes

$$\begin{array}{rl}
 3 \times 10^0 & \\
 + 1 \times 10^{-1} & // 1 \times 1/10^1 = 1 \times 1/10 \\
 + 4 \times 10^{-2} & // 4 \times 1/10^2 = 4 \times 1/100 \\
 + 1 \times 10^{-3} & // 1 \times 1/10^3 = 1 \times 1/1000 \\
 + 5 \times 10^{-4} & // 5 \times 1/10^4 = 5 \times 1/10000 \\
 + 9 \times 10^{-5} & // 9 \times 1/10^5 = 9 \times 1/100000
 \end{array}$$

which becomes

$$\begin{array}{l}
 3 \times 1 \\
 + 1 \times 0.1 \\
 + 4 \times 0.01 \\
 + 1 \times 0.001 \\
 + 5 \times 0.0001 \\
 + 9 \times 0.00001
 \end{array}$$

and

$$\begin{array}{l}
 3 \\
 + 0.1 \\
 + 0.04 \\
 + 0.001 \\
 + 0.0005 \\
 + 0.00009
 \end{array}$$

3.14159

In the *binary*, or *base-2* system, each digit of a given number has 1 of 2 possible values, $\{0, 1\}$. To calculate the value of a base-2 number, the procedure is similar to that of a base-10 number:

- Count the number of digits d to the left of the radix point
- Multiply the leftmost digit by 2^{d-1} , the next digit by 2^{d-2} , the next digit by 2^{d-3} , etc.
- Take the sum of those values

The binary (base-2) number

```
1101 // 4 digits to the left of the radix point (d = 4)
```

for example, denotes

$$\begin{array}{r} 1 \times 2^3 \\ + 1 \times 2^2 \\ + 0 \times 2^1 \\ + 1 \times 2^0 \end{array}$$

which becomes

$$\begin{array}{r} 1 \times 8 \\ + 1 \times 4 \\ + 0 \times 2 \\ + 1 \times 1 \end{array}$$

and

$$\begin{array}{r} 8 \\ + 4 \\ + 0 \\ + 1 \\ \hline \end{array}$$

```
13 // Decimal (base-10) value of the binary (base-2) number 1101
```

Similarly, the binary number

```
10010110 // 8 digits to the left of the radix point (d = 8)
```

denotes

$$\begin{array}{r} 1 \times 2^7 \\ + 0 \times 2^6 \\ + 0 \times 2^5 \\ + 1 \times 2^4 \\ + 0 \times 2^3 \\ + 1 \times 2^2 \\ + 1 \times 2^1 \\ + 0 \times 2^0 \end{array}$$

which becomes

$$\begin{array}{r} 1 \times 128 \\ + 0 \times 64 \\ + 0 \times 32 \\ + 1 \times 16 \\ + 0 \times 8 \\ + 1 \times 4 \\ + 1 \times 2 \\ + 0 \times 1 \end{array}$$

and

$$\begin{array}{r}
 128 \\
 + 0 \\
 + 0 \\
 + 16 \\
 + 0 \\
 + 4 \\
 + 2 \\
 + 0 \\
 \hline
 \end{array}$$

150 // Decimal (base-10) value of the binary (base-2) number 10010110

The binary number

10.1101 // 2 digits to the left of the radix point (d = 2)

denotes

$$\begin{array}{rl}
 1 & \times 2^1 \\
 + 0 & \times 2^0 \\
 + 1 & \times 2^{-1} \quad // 1 \times 1/2^1 = 1 \times 1/2 \\
 + 1 & \times 2^{-2} \quad // 1 \times 1/2^2 = 1 \times 1/4 \\
 + 0 & \times 2^{-3} \quad // 1 \times 1/2^3 = 1 \times 1/8 \\
 + 1 & \times 2^{-4} \quad // 1 \times 1/2^4 = 1 \times 1/16
 \end{array}$$

which becomes

$$\begin{array}{r}
 1 \times 2 \\
 + 0 \times 1 \\
 + 1 \times 0.50 \\
 + 1 \times 0.25 \\
 + 0 \times 0.125 \\
 + 1 \times 0.0625
 \end{array}$$

and

$$\begin{array}{r}
 2 \\
 + 0 \\
 + 0.50 \\
 + 0.25 \\
 + 0 \\
 + 0.0625 \\
 \hline
 \end{array}$$

2.8125 // Decimal (base-10) value of the binary (base-2) number 10.1101

The smallest unit of information storable by a computer is a single binary digit known as a *bit*. A single bit, which has a value of either 0 or 1, can be physically represented by any device with two states:

- Early computers used paper cards, commonly known as *punch cards*. Punch cards were divided into rows and columns, where each cell represented a single bit. A punched hole in a given cell represented a value of 1, while the absence of a hole represented a value of 0.
- Punch cards were eventually replaced by magnetic tapes and hard disks, which use electrical polarity (+ / -) to represent the values of a bit (positive polarity = 1, negative polarity = 0).
- Optical discs (CDs, DVDs, etc.) represent bits using microscopic surface indentations called *pits*. A pit represents a value of 0, while the absence of a pit represents a value of 1.
- In random access memory (RAM), each bit is represented using a *capacitor* (a type of electrical component), which is either charged (1) or discharged (0).

The next smallest unit of information is a *byte*, also known as a *word*. In most computer architectures, a byte consists of 8 bits; a single 8-bit byte can therefore represent a total of 2^8 (256) nonnegative integers (0-255):

```

00000000 // 0
00000001 // 1
00000010 // 2
00000011 // 3
00000100 // 4
00000101 // 5
00000110 // 6
00000111 // 7
// ...
11111111 // 255

```

The common use of an 8-bit byte, also known as an *octet*, is rooted in character encoding. The *ASCII* (American Standard Code for Information Interchange) defines a set of 128 codes. Codes 32-126 represent alphanumeric characters and punctuation marks:

Code	Glyph (Symbol)	Code	Glyph	Code	Glyph
32	<whitespace>	48	0	58	:
33	!	49	1	59	;
34	"	50	2	60	<
35	#	51	3	61	=
38	&	52	4	62	>
39	'	53	5	63	?
40	(54	6	64	@
41)	55	7		
42	*	56	8		
43	+	57	9		
44	,				
45	-				
46	.				
47	/				

Code	Glyph	Code	Glyph	Code	Glyph	Code	Glyph
65	A	91	[97	a	123	{
66	B	92	\	98	b	124	
67	C	93]	99	c	125	}
68	D	94	^	100	d	126	~
69	E	95	-	101	e		
70	F	96	‘	102	f		
71	G			103	g		
72	H			104	h		
73	I			105	i		
74	J			106	j		
75	K			107	k		
76	L			108	l		
77	M			109	m		
78	N			110	n		
79	O			111	o		
80	P			112	p		
81	Q			113	q		
82	R			114	r		
83	S			115	s		
84	T			116	t		
85	U			117	u		
86	V			118	v		
87	W			119	w		
88	X			120	x		
89	Y			121	y		
90	Z			122	z		

Codes 0-31 and 127 represent *control codes* (line feed, tab, backspace, etc.):

Code	Special symbol / command	Code	Special symbol / command
0	Null character ('\0')	17	Device control 1
1	Start of header	18	Device control 2
2	Start of text	19	Device control 3
3	End of text	20	Device control 4
4	End of transmission	21	Negative acknowledgement
5	Enquiry	22	Synchronous idle
6	Acknowledgement	23	End of transmission block
7	Bell ('\a')	24	Cancel
8	Backspace ('\b')	25	End of medium
9	Horizontal tab ('\t')	26	Substitute
10	Line feed ('\n')	27	Escape ('\e')
11	Vertical tab ('\v')	28	File separator
12	Form feed ('\f')	29	Group separator
13	Carriage return ('\r')	30	Record separator
14	Shift out	31	Unit separator
15	Shift in	127	Delete
16	Data link escape		

An 8-bit byte is large enough to store the code of any ASCII character (0 to 127), or the code of a non-ASCII character defined by the system's architecture (we'll discuss non-ASCII codes a little bit more in the next chapter).

In all C++ implementations, the size of a *char* is exactly 1 byte. The sizes of all other types (*int*, *double*, etc.) are implementation-dependent, but they are guaranteed to be whole-number multiples of 1 byte.

A character string, at the lowest level, is an array of *chars*. All strings are *null-terminated*, meaning that a *null character* ('\0', ASCII code 0) is placed at the end to indicate the end of the string. A string of *n* characters therefore actually requires (*n* + 1) bytes of memory. The string

```
Saturn      // 6 visible chars: 'S', 'a', 't', 'u', 'r', 'n'
```

for example, is an array of 7 *chars* (7 bytes):

Bytes:							
01010011	01100001	01110100	01110101	01110010	01101110	00000000	
Base-10 values (ASCII codes):							
83	97	116	117	114	110	0	
Glyphs:							
S	a	t	u	r	n	\0	

Similarly, the string

```
Milky Way    // 9 visible chars: 'M', 'i', 'l', 'k', 'y', <space>,
//                                'W', 'a', 'y'
```

is an array of 10 *chars* (10 bytes):

Bytes:							
01001101	01101001	01101100	01101011	01111001	00100000	01010111	
01100001	01111001	00000000					
Base-10 values (ASCII codes):							
77	105	108	107	121	32	87	
97	121	0					
Glyphs:							
M	i	l	k	y	<space>	W	
a	y	\0					

These are known as *C-style strings*, or simply *C-strings*, because C++ inherited them from C. A C-string's official type is *char[]* (array of *chars*), and the name of a given C-string variable is actually a pointer (*char**) to the first element of the array:

```

char[] x = "Saturn";           // x is a C-string (array of 7 chars)
//   x[0] = 'S', x[1] = 'a', x[2] = 't', ...
//   x[6] = '\0'
// x is a pointer (char*) to the first element
//   *x = 'S', *(x + 1) = 'a', *(x + 2) = 't', ...
//   *(x + 6) = '\0'

char[] y = "Milky Way";       // y is a C-string (array of 10 chars)
//   y[0] = 'M', y[1] = 'i', y[2] = 'l', ...
//   y[10] = '\0'
// y is a pointer (char*) to the first element
//   *y = 'M', *(y + 1) = 'i', *(y + 2) = 'l', ...
//   *(y + 10) = '\0'

```

Because they're just primitive arrays, C-strings are relatively cumbersome to work with, even when they're the same length:

```

char[] z = "Gemini";          // z is the same size as x
// (7 chars)

x = z;                        // error: primitive arrays don't
// support assignment

for (size_t i = 0; i != sizeof(x); ++i)    // ok: set x to "Gemini", one
    x[i] = z[i];                  // element at a time
// sizeof(x) = 7

```

And the only way to have resizable C-strings is to use raw pointers (*char**) to dynamically-allocated arrays, which can be particularly error-prone:

```

char* w = new char[sizeof(z)];      // w is a dynamically-allocated
// array of 7 chars

for (size_t i = 0; i != sizeof(w); ++i)    // set w to "Gemini", one element
    w[i] = z[i];                      // at a time

delete[] w;                         // don't forget to deallocate the
// current array before creating a
// new one

w = new char[sizeof(y)];            // create a new dynamically-
// allocated array of 10 chars

for (size_t i = 0; i != sizeof(w); ++i)    // set w to "Milky Way", one
    w[i] = y[i];                      // element at a time

delete[] w;                         // don't forget to deallocate the
// array when we're done

```

The C++ Standard Library header *<cstring>* provides a set of useful functions (inherited from the C library) for working with C-strings. This chapter isn't meant to be an in-depth guide to working with C-strings; rather, our main goal is to discuss the binary / in-memory representation of strings, as well

as the relationship between C-strings and `std::strings`.

A `std::string` is a proper object, containing a pointer to a dynamically-allocated array of `char`s, and behaves much like a `std::vector<char>`:

The `string` class, in fact, supports many of the same operations as `vector`, including `push_back`, `pop_back`, `operator[]`, and `operator=`:

```
a.pop_back()           // a = "Mar"
a[0] = 'c';            // a.operator[](0) = 'c';
                      // a = "car"
a[2] = 's';            // a = "cas"
a.push_back('h');      // a = "cash"

a = b;                 // a.operator=(b);
                      // resizes a's array as needed, then assigns the
                      // value of each element (char), one at a time
                      // a = "Jupiter"

                      // when a and b go out of scope, their destructors
                      // free the arrays
```

`std::string` also provides the member functions

```
const char* c_str() const;      // c_str is short for "C-string"
const char* data() const;
```

which both return a pointer to the beginning (first *char*) of the array. These methods are useful for passing the value of a *std::string* to older functions that work with C-strings (*char**). It's worth noting, however, that the *std::string* member functions

```
size_type size() const;  
size_type length() const;
```

which both return the length of the string, do *not* include the null character at the end;

We'll use the *c_str* and *size* methods later on in this section, to print the binary representation of a *std::string*'s value. We'll also use them to implement a hash function in Part 12.

11.2: Integers

Chapter outline

- Bit representation of integers, using sign-magnitude and two's complement format

The most basic way of representing signed integers is *sign-magnitude representation*. Using this method, the first (leftmost) bit is the *sign bit*, which is 0 for non-negative numbers and 1 for negative numbers. The remaining bits contain the binary representation of the *magnitude* (the absolute value of the number). Using 4 bytes (32 bits), for example, the base-10 number 43 is

```
Sign bit (0 for non-negative)
|
00000000 00000000 00000000 00101011
|
Magnitude (43) ----->
```

while the number -43 is

```
Sign bit (1 for negative)
|
10000000 00000000 00000000 00101011
|
Magnitude (43) ----->
```

Although it's the most intuitive method, sign-magnitude representation has a few drawbacks, the first of which is that there are two ways to represent 0:

```
Sign bit (0 for non-negative)
|
00000000 00000000 00000000 00000000 // "Positive zero"
|
Magnitude (0) ----->

Sign bit (1 for negative)
|
10000000 00000000 00000000 00000000 // "Negative zero"
|
Magnitude (0) ----->
```

This complicates things a bit at the hardware level, as it must be designed to handle calculations involving both positive and negative zero. Addition, subtraction, and multiplication are also more complex because the procedures vary based on the signs of the operands.

Two's complement representation, which avoids these issues, is what most computers actually use. In two's complement format, non-negative numbers have the exact same representation as they do in sign-magnitude format. But unlike sign-magnitude format, two's complement allows one representation of zero:

```
// Assuming a 4-byte (32-bit) integer

00000000 00000000 00000000 00101011 // 43 (Same as sign-magnitude)
00000000 00000000 00000000 00000000 // 0
```

The main difference between two's complement and sign-magnitude format lies in the representation of negative numbers. If the first (leftmost) bit of a two's complement number is 1, then that number is negative (as it is in sign-magnitude format). The remaining bits, however, don't represent the absolute value of the number, but rather a *conversion* of the absolute value. The procedure for representing a negative number in two's complement format is:

- Start with the absolute value
- *Invert* each bit (0 becomes 1, 1 becomes 0)
- Add 1 to the result

To represent -43, for example:

```
// Start with the absolute value (43)

00000000 00000000 00000000 00101011

// Invert each bit

11111111 11111111 11111111 11010100

// Add 1 to the result

 11111111 11111111 11111111 11010100
+ 00000000 00000000 00000000 00000001
-----
11111111 11111111 11111111 11010101 // -43 (two's complement)
```

Given a negative number N in two's complement format, we can obtain its value by following a similar procedure:

- Invert each bit (0 becomes 1, 1 becomes 0)
- Add 1 to the result (this yields the absolute value of N)
- Take the *additive inverse* (negative)

Consider, for example, the negative two's complement number

```
11111111 11111111 11111111 11010101 // N = ?

// Invert each bit

00000000 00000000 00000000 00101010

// Add 1 to the result
```

```

00000000 00000000 00000000 00101010
+ 00000000 00000000 00000000 00000001
-----
00000000 00000000 00000000 00101011 // 43 (the absolute value of N)
// N = the additive inverse of 43 = -43

```

The two's complement method of representing negative numbers greatly simplifies addition and subtraction. We can solve any problem by performing simple addition, regardless of whether the operands are positive or negative:

37 + 19 = ?

```

00000000 00000000 00000000 00100101 // 37
+ 00000000 00000000 00000000 00010011 // 19
-----
0 // Carry the 1

```

```

00000000 00000000 00000000 00100101
+ 00000000 00000000 00000000 00010011
-----
```

```

00000000 00000000 00000000 00100101
+ 00000000 00000000 00000000 00010011
-----
```

```

00000000 00000000 00000000 00100101
+ 00000000 00000000 00000000 00010011
-----
```

```

00000000 00000000 00000000 00100101
+ 00000000 00000000 00000000 00010011
-----
```

```

00000000 00000000 00000000 00100101
+ 00000000 00000000 00000000 00010011
-----
```

111000

// Repeat until done...

00000000	00000000	00000000	00100101
+ 00000000	00000000	00000000	00010011

00000000	00000000	00000000	00111000	// 56
----------	----------	----------	----------	-------

37 - 19 = ?

= 37 + (-19)

// To represent -19, we start with the absolute value (19), invert the bits,
// then add 1 to the result

00000000	00000000	00000000	00100101	// 37
+ 11111111	11111111	11111111	11101101	// -19

0	// Carry the 1
---	----------------

00000000	00000000	00000000	00100101	1
+ 11111111	11111111	11111111	11101101	

10

00000000	00000000	00000000	00100101
+ 11111111	11111111	11111111	11101101

010	// Carry the 1
-----	----------------

00000000	00000000	00000000	00100101	1
+ 11111111	11111111	11111111	11101101	

0010	// Carry the 1
------	----------------

00000000	00000000	00000000	00100101	1
+ 11111111	11111111	11111111	11101101	

10010

00000000	00000000	00000000	00100101
+ 11111111	11111111	11111111	11101101

010010	// Carry the 1
--------	----------------

```

      1
00000000  00000000  00000000  00100101
+ 11111111  11111111  11111111  11101101
-----
                           0010010 // Carry the 1

      1
00000000  00000000  00000000  00100101
+ 11111111  11111111  11111111  11101101
-----
                           00010010 // Carry the 1

      1
00000000  00000000  00000000  00100101
+ 11111111  11111111  11111111  11101101
-----
                           0  00010010 // Carry the 1

// Repeat, carrying the 1 each time...

      1
00000000  00000000  00000000  00100101
+ 11111111  11111111  11111111  11101101
-----
                           00000000  00000000  00000000  00010010 // Carry the 1

// The final carried 1 overflows past our 32-bit capacity, so we're done.
// The result is 18.



---


19 - 37 = ?
= 19 + (-37)

// To represent -37, we start with the absolute value (37), invert the bits,
// then add 1 to the result

      00000000  00000000  00000000  00010011 // 19
+ 11111111  11111111  11111111  11011011 // -37
-----
                           11111111  11111111  11111111  11101110 // Result (R)

// The first (leftmost) bit is 1, so we know that R is a negative number

// To obtain the value of R, we invert the bits...

      00000000  00000000  00000000  00010001

// ...add 1...

```

```

00000000 00000000 00000000 00010001
+ 00000000 00000000 00000000 00000001
-----
00000000 00000000 00000000 00010010 // 18 (the absolute value of R)

// ...then take the additive inverse (R = -18)
-----
-37 - 19 = ?
= -37 + (-19)

11111111 11111111 11111111 11011011 // -37
+ 11111111 11111111 11111111 11101101 // -19
-----
11111111 11111111 11111111 11001000 // Result (R)

// The first (leftmost) bit is 1, so we know that R is a negative number

// To obtain the value of R, we invert the bits...

00000000 00000000 00000000 00110111

// ...add 1...
00000000 00000000 00000000 00110111
+ 00000000 00000000 00000000 00000001
-----
00000000 00000000 00000000 00111000 // 56 (the absolute value of R)

// ...then take the additive inverse (R = -56)
-----
```

Another difference between two's complement and sign-magnitude representation lies in the smallest value they can represent. Using N bits, the smallest possible value in sign-magnitude format is $(-2^{N-1} + 1)$, while the smallest possible value in two's complement format is -2^{N-1} .

Consider, for example, a system using 4-bit integers. Both formats can represent a total of 15 ($2^4 - 1$) values (excluding the negative zero in sign-magnitude format). Sign-magnitude, however, can represent the range $[-7, 7]$, while two's complement can represent the range $[-8, 7]$:

Integer value	Sign-magnitude format	Two's complement format
-8	n/a	1000
-7	1111	1001
-6	1110	1010
-5	1101	1011
-4	1100	1100
-3	1011	1101
-2	1010	1110
-1	1001	1111
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111

Recall from the previous chapter that an 8-bit byte is large enough to store the code of any ASCII character (0 to 127), or the code of a non-ASCII character defined by the system's architecture. The codes reserved for non-ASCII characters (which may include additional punctuation marks, currency symbols, etc.) depend upon the integer format used by the system:

- In unsigned integer format, an 8-bit byte can represent the range [0, 255] (non-ASCII codes = 128 to 255)
- In sign-magnitude format, an 8-bit byte can represent the range [-127, 127] (non-ASCII codes = -127 to -1)
- In two's complement format, an 8-bit byte can represent the range [-128, 127] (non-ASCII codes = -128 to -1)

The newer *Unicode standard* allows multiple 8-bit bytes per character, thereby exponentially increasing the number of available codes. Unicode supports over 100,000 characters, spanning most of the world's written languages.

This isn't meant to be an exhaustive guide to character encoding formats, however. For our purposes, the key points are:

- A *bit* (binary digit, 0 or 1) is the smallest unit of information storables by a computer, because it can be physically represented by any 2-state device.
- A *byte* (or *word*), the next smallest unit of information, consists of 8 bits in most computer architectures. The widespread use of an 8-bit byte arose from its ability to represent all 128 ASCII codes, as well as a set of additional (non-ASCII) codes.
- In all C++ implementations, the size of a *char* is guaranteed to be exactly 1 byte.
- The size of a non-*char* (*int*, *double**, *std::list<char>*, etc.) is implementation-dependent, but guaranteed to be a whole number of bytes.
- In most C++ implementations, the size of an *int* is 4 bytes.

- A *C-string*, the most basic type of character string, is a primitive null-terminated array of *chars*. The name of a C-string variable is a pointer (*char**) to the first element.
- A *std::string* is an object that manages a dynamically-allocated array of *chars*, similar to a *std::vector<char>*. The member functions *size* and *length* both return the length of the string, excluding the null character at the end. The *c_str* and *data* methods both return a pointer (*char**) to the first element of the array.
- The most common binary representation for integers is *two's complement format* because it's relatively easy to implement at the hardware level, though *sign-magnitude format* is more intuitive.

11.3: Floating-Point (Real) Numbers

Chapter outline

- Scientific notation
- Bit representation of floating-point (real) numbers, using the IEEE 754 Standard

The most commonly used format for representing floating-point numbers is the *IEEE* (Institute of Electrical and Electronics Engineers) 754 Standard for Floating-Point Arithmetic. But before we get into the specifics of IEEE 754, let's review scientific notation.

In *scientific notation*, a number N is represented in terms of a *mantissa* (M), *base* (B), and *exponent* (E), where

$$N = M \times B^E$$

and the mantissa (also known as the *coefficient* or *significand*) contains 1 nonzero digit to the left of the radix point. Consider, for example,

```
13.62510           // The base-10 number 13.625 (the subscript indicates the base)
```

To represent this number in scientific notation, we move the radix point 1 place to the left and use an exponent of 1:

$$13.625_{10} = 1.3625 \times 10^1$$

exponent (1)
 /
 | \\\
 mantissa base
 (1.3625) (10)

Similarly, to represent the number

```
-9473.18610
```

we move the radix point 3 places to the left and use an exponent of 3:

$$-9473.186_{10} = -9.473186 \times 10^3$$

exponent (3)
 /
 | \\\
 mantissa base
 (-9.473186) (10)

To represent the number

```
0.06809410
```

we move the radix point 2 places to the *right*, so the exponent is -2 (as opposed to 2):

$$0.068094_{10} = \begin{array}{c} \text{exponent } (-2) \\ / \\ 6.8094 \times 10^{-2} \\ | \quad \backslash \\ \text{mantissa} \quad \text{base} \\ (6.8094) \quad (10) \end{array}$$

All the above examples are expressed in base-10 scientific notation, but IEEE 754 uses base-2. Let's revisit the first example,

$$13.625_{10}$$

To express this number in base-2 scientific notation, we first obtain the binary value:

$$\begin{aligned} 13.625_{10} &= 13_{10} + 0.625_{10} \\ &= 1101_2 + 0.101_2 \quad // (8 + 4 + 0 + 1)_{10} + (1/2 + 0/4 + 1/8)_{10} \\ &= 1101.101_2 \end{aligned}$$

We then express the binary value (1101.101_2) in scientific notation by moving the radix point 3 places to the left and using an exponent of 3:

$$1101.101_2 = \begin{array}{c} \text{exponent } (3) \\ / \\ 1.101101_2 \times 2^3 \\ | \quad \backslash \\ \text{mantissa} \quad \text{base} \\ (1.101101) \quad (2) \end{array}$$

We can verify that this is correct by converting it back to base 10:

$$\begin{aligned} 1.101101_2 \times 2^3 &= (1 + 1/2 + 0/4 + 1/8 + 1/16 + 0/32 + 1/64)_{10} \times 2^3 \\ &= 1.703125_{10} \times 8 \\ &= 13.625_{10} \end{aligned}$$

In the above example, the fractional portion of the given quantity (13.625_{10}) had an exact binary representation (0.625_{10} is exactly 0.101_2). When this isn't the case, however, the binary representation will be an approximation. Consider, for example,

$$\begin{aligned} -3.3_{10} &\quad // \text{The fractional portion } (0.3_{10}) \text{ doesn't have an exact binary} \\ &\quad // \text{representation} \end{aligned}$$

The binary approximation is

$$\begin{aligned} -3.3_{10} &= -(3_{10} + 0.3_{10}) \\ &\approx -(11_2 + 0.01001_2) \quad // -((2 + 1)_{10} + \\ &\quad // (0/2 + 1/4 + 0/8 + 0/16 + 1/32)_{10}) \\ &\approx -11.01001_2 \end{aligned}$$

To represent -11.01001_2 in scientific notation, we move the radix point 1 place to the left and use an exponent of 1:

$$\begin{array}{r} \text{exponent (1)} \\ / \\ -11.01001_2 = -1.101001_2 \times 2^1 \\ | \quad \backslash \\ \text{mantissa} \quad \text{base} \\ (-1.101001) \quad (2) \end{array}$$

The base-10 equivalent of the binary approximation is

$$\begin{aligned} -1.101001_2 \times 2^1 &= -(1 + 1/2 + 0/4 + 1/8 + 0/16 + 0/32 + 1/64)_{10} \times 2^1 \\ &= -1.640625_{10} \times 2 \\ &= -3.28125_{10} \end{aligned}$$

The *error* is the difference between the exact quantity (-3.3_{10}) and the binary approximation (-3.28125_{10}), which amounts to 0.01875.

In the above example, we used 7 digits to represent the mantissa (-1.101001_2). By using more digits, we can obtain a more precise approximation. Suppose, for example, that we use 8 digits to represent the mantissa:

```
// Binary approximation

-3.310 = -(310 + 0.310)
≈ -(112 + 0.0100112)      // -((2 + 1)10 +
                                // (0/2 + 1/4 + 0/8 + 0/16 + 1/32 + 1/64)10)
≈ -11.0100112

// Scientific notation (Move the radix point 1 place to the left and use an
// exponent of 1)


$$\begin{array}{r} \text{exponent (1)} \\ / \\ -11.010011_2 = -1.1010011_2 \times 2^1 \\ | \quad \backslash \\ \text{mantissa (8 digits)} \quad \text{base} \\ (-1.1010011) \quad (2) \end{array}$$


// Base 10 equivalent of the binary approximation

-1.10100112 × 21 = -(1 + 1/2 + 0/4 + 1/8 + 0/16 + 0/32 + 1/64 + 1/128)10 × 21
= -1.648437510 × 2
= -3.29687510

Error = AbsValue(-3.310 - -3.29687510)
= 0.003125
```

The 8-digit mantissa incurs less error than the 7-digit mantissa (0.003125 vs. 0.01875), making it more accurate. The more digits we use, the greater the level of *precision* (the closer we can approximate the

exact quantity (-3.3_{10}), at the cost of increased storage space).

The IEEE 754 Standard defines 4 binary formats for representing floating-point numbers:

Format		Sign bits	Biased exponent bits (E)	Stored mantissa bits	Bias $(2^{E-1} - 1)$
binary16	(Half precision)	1	5	10	15
binary32	(Single precision)	1	8	23	127
binary64	(Double precision)	1	11	52	1023
binary128	(Quadruple precision)	1	15	112	16383

The *binary16* format, for example, uses 16 bits to represent each number: 1 for the sign (positive / negative), 5 for the exponent, and 10 for the mantissa. The base, which is assumed to be 2, is not stored explicitly.

The bit layout for representing a given number N is

```
<sign> <biased exponent> <stored mantissa>
```

where

```
sign = sign of the true mantissa (0/1 for positive/negative)
biased exponent = true exponent + bias
stored mantissa = fractional portion of the true mantissa
```

The *true exponent / true mantissa* are simply the exponent / mantissa of N , as expressed in base-2 scientific notation. The *biased exponent / stored mantissa* are the values actually stored by the computer. Consider, for example, the number

5.25_{10}

To store this in binary16 format, we first obtain the binary value:

$$\begin{aligned} 5.25_{10} &= 5_{10} + 0.25_{10} \\ &= 101_2 + 0.01_2 \quad // \quad (4 + 0 + 1)_{10} + (0/2 + 1/4)_{10} \\ &= 101.01_2 \end{aligned}$$

We then express the binary value in scientific notation by moving the radix point 2 places to the left and using an exponent of 2:

$$101.01_2 = 1.0101_2 \times 2^2$$

true exponent (2)
 /
 |
 true mantissa base
 (1.0101) (2)

We can then obtain the biased exponent and stored mantissa using the above table and formulas:

```

sign           = sign of the true mantissa (0/1 for positive/negative)
                = 0

biased exponent = true exponent + bias
                = 2 + 15
                = 1710
                = 100012           // 16 + 0 + 0 + 0 + 1

stored mantissa = fractional portion of the true mantissa
                = 0101
                = 0101000000          // 10-bit representation (6 trailing zeroes)

```

Using 2 8-bit bytes, the complete binary16 representation of 5.25₁₀ is thus

```

sign
(0)   stored mantissa (0101000000)
|     |
01000101 01000000
|
biased exponent (10001)

```

Omitting the first (leftmost) digit of the true mantissa is a space-saving optimization: as long as N is not zero, the first digit of the true mantissa is guaranteed to be 1 so we don't need to explicitly store it:

```

true exponent    = 2
biased exponent = 2 + 15 = 1710
                  = 100012
/
4.510 = 100.12 = 1.0012 x 22
|
true mantissa    = 1.001 (sign bit = 0)
stored mantissa = 001
                  = 0010000000 (7 trailing zeroes)

```

Complete binary16 representation:

```

sign   stored mantissa
|     |
01000100 10000000
|
biased exponent

```

```

true exponent    = -5
biased exponent = -5 + 15 = 1010
                  = 010102
/
-0.039062510 = -0.00001012 = -1.01 x 2-5
|
true mantissa    = -1.01 (sign bit = 1)
stored mantissa = 01
                  = 0100000000 (8 trailing zeroes)

```

Complete binary16 representation:

```
sign  stored mantissa
|      |
10101001 00000000
|
biased exponent
```

A special case exists for $N = 0$, which is represented by setting all the bits to 0:

Complete binary16 representation of 0:

```
sign  stored mantissa
|      |
00000000 00000000
|
biased exponent (00000, reserved for N = 0)
```

A biased exponent with all the bits set to 1 is also reserved, for representing infinity and *NaN* (“Not a Number,” used for undefined values such as the result of division by zero). To avoid conflicts with these reserved values, the biased exponent must lie in the range $[1, 2 \times bias]$. The true exponent (which is equal to the biased exponent minus the bias) must therefore lie in the range $[1 - bias, bias]$:

Format	Exponent bits (E)	Bias $(2^{E-1} - 1)$	Biased exponent range	True exponent range
binary16	5	15	[1, 30]	[-14, 15]
binary32	8	127	[1, 254]	[-126, 127]
binary64	11	1023	[1, 2046]	[-1022, 1023]
binary128	15	16383	[1, 32766]	[-16382, 16383]

In binary16, for example, the reserved biased exponents are 00000_2 (0_{10}) and 11111_2 (31_{10}). The biased exponent of any nonzero real number N that we wish to store must therefore be between 1 and 30 (inclusive). The bias is 15, so N 's true exponent must be between -14 and 15 (inclusive):

```
biased exponent = true exponent + bias
biased exponent = true exponent + 15

1   <= biased exponent    <= 30          // 0 and 32 are reserved
1   <= true exponent + 15 <= 30          // via substitution
-14 <= true exponent     <= 15
```

Similarly, in binary32 the reserved biased exponents are 00000000_2 (0_{10}) and 11111111_2 (255_{10}). To store a nonzero real number N , the biased exponent must be between 1 and 254 (inclusive). The bias is 127, so N 's true exponent must be between -126 and 127 (inclusive):

```
biased exponent = true exponent + bias
biased exponent = true exponent + 127
```

```

1     <= biased exponent      <= 254          // 0 and 255 are reserved
1     <= true exponent + 127 <= 254          // via substitution
-126 <= true exponent      <= 127

```

To convert a nonzero number from IEEE 754 to base-10, we first derive the true mantissa and true exponent. We can then convert the base 2 scientific notation to the base-10 representation. Consider, for example, the binary16 number

```

sign   stored mantissa
|       |
11011001 00011010
|
biased exponent

true mantissa = <sign: +/- for 0/1> <1.> <stored mantissa>
                = -1.01000110102

biased exponent = true exponent + bias
true exponent   = biased exponent - bias
                  = 101102 - 15
                  = (16 + 0 + 4 + 2 + 0)10 - 15
                  = 22 - 15
                  = 7

base-2 scientific notation = true mantissa x 2true exponent
                           = -1.01000110102 x 27

base-10 representation    = -1.01000110102 x 27
                           = -10100011.0102
                           = -101000112 + -0.0102
                           = -(128 + 0 + 32 + 0 + 0 + 0 + 2 + 1)10 +
                             -(0/2 + 1/4 + 0/8)10
                           = -163.2510

```

In most C++ implementations, the types *float* and *double* are represented using binary32 and binary64, respectively. We'll see how to obtain additional type information in the next chapter.

Before we move on, it's worth mentioning that in the IEEE 754 Standard and elsewhere, the *stored mantissa* is simply called the *mantissa*. Additionally, the *biased exponent* may simply be referred to as the *exponent*.

The other terms used in this chapter (*true exponent*, *true mantissa*, *stored mantissa*) were coined by me to make it easier to explain IEEE 754 and differentiate the various values. As a result, you probably won't see or hear them elsewhere.

11.4: Bitwise Operations

Source files and folders

- *bitwiseOperations*

Chapter outline

- *Obtaining type information*
- *Bitwise logical and shift operators*
- *Endianness*

Now that we have an understanding of binary representation, we're ready to begin working with bits and bytes in C++. As discussed in Chapters 11.1 and 11.2,

- The number of bits per byte is hardware-dependent, though usually 8.
- In all C++ implementations, the size of a *char* is guaranteed to be exactly 1 byte.
- The size of a non-*char* (*int*, *double**, etc.) is implementation-dependent, but guaranteed to be a whole number of bytes

We can obtain the number of bits per byte via the *CHAR_BIT* macro, defined in the Standard Library header *<climits>*. On my machine, for example, it's

```
#define CHAR_BIT 8      // 8 bits per byte (char)
```

This means that before compilation, the preprocessor will replace all instances of the text *CHAR_BIT* with 8. A statement like

```
cout << "On this machine, there are " << CHAR_BIT << " bits per byte\n";
```

for example, becomes

```
cout << "On this machine, there are " << 8 << " bits per byte\n";
```

As a sidenote, using macros to define constant values originated in C, which didn't have the *const* keyword. Using *const* is preferred, however, because it's less error-prone than the macro approach.

To obtain implementation-dependent traits of a given type *T*, we can use the *numeric_limits* class, provided in the Standard Library header *<limits>*. We'll use *numeric_limits* and *CHAR_BIT* in our own function (*printTypeTraits.h*, lines 9-10),

```
template <class T>
void printTypeTraits();
```

to print:

- T 's typename (*int*, *double*, etc.)
- The size of a single object of type T , in bytes and bits
- Whether T is a signed or unsigned type (1 if signed, 0 if unsigned)
- The number of:
 - Non-sign bits (if T is an *integral type*, i.e. a type that stores an integer value)
 - Mantissa bits (if T is a floating-point type)

The expression (line 17)

```
typeid(T)
```

returns an unnamed object of type *std::type_info*, on which we call the member function *name*. The *name* method then returns T 's typename. Note, however, that *name* returns a C-string (*const char**), not a *std::string*.

The expression (line 19)

```
sizeof(T)
```

returns the size of a single object of type T , in bytes (recall that we used *sizeof* in Volume 1, to implement the *Allocator* class). The expression (line 20)

```
sizeof(T) * CHAR_BIT // size of a T (in bytes) * number of bits per byte
```

therefore returns the size of a single object of type T , in bits.

In the expressions (lines 22-23)

```
numeric_limits<T>::is_signed
numeric_limits<T>::digits
```

is_signed and *digits* are public static data members of the class *numeric_limits*< T >:

- *is_signed* is either 1 or 0, depending on whether T is a signed or unsigned type
- *digits* is the number of non-sign bits or mantissa bits, depending on whether T is an integral or floating-point type

On my machine, the function calls (*main.cpp*, lines 12-17)

```
printTypeTraits<char>();
printTypeTraits<bool>();
printTypeTraits<unsigned int>();
printTypeTraits<int>();
printTypeTraits<float>();
printTypeTraits<double>();
```

generate the output

```

Type      = char
Size      = 1 byte(s) = 8 bits
Is signed = 1
Digits    = 7

Type      = bool
Size      = 1 byte(s) = 8 bits
Is signed = 0
Digits    = 1

Type      = unsigned int
Size      = 4 byte(s) = 32 bits
Is signed = 0
Digits    = 32

Type      = int
Size      = 4 byte(s) = 32 bits
Is signed = 1
Digits    = 31

Type      = float
Size      = 4 byte(s) = 32 bits
Is signed = 1
Digits    = 24

Type      = double
Size      = 8 byte(s) = 64 bits
Is signed = 1
Digits    = 53

```

As expected, the size of a *char* is a single 8-bit byte. The first thing we notice is that my system uses a signed *char* type. As we discussed in Chapter 11.2, this means that non-ASCII character codes lie in the range [-128, -1] (assuming that the codes are stored in two's complement format).

A *bool* is the same size as a *char* (1 byte). A *bool* can only have 2 possible values (1/0 for true/false), however, so this type only has 1 non-sign bit (*digits* = 1).

An *unsigned int* is the same size as a signed *int* (4 bytes), but an *unsigned int* can store larger positive values because it has 32 non-sign bits (as opposed to a signed *int*, which has 31 non-sign bits + 1 sign bit).

The *float* type has a 24-bit mantissa (*digits* = 24). This refers to the *true mantissa*, not the *stored mantissa* (which is 23 bits because it omits the first digit of the true mantissa). Similarly, the *double* type has a 53-bit true mantissa (*digits* = 53); the stored mantissa is 52 bits.

The next thing we'll do is write a function to print the binary representation of a given value. In order to do that, however, we'll need a basic understanding of *bitwise operations* (operations performed on individual bits). There are two types of bitwise operators: *logical operators* and *shift operators*.

Bitwise logical operators		Bitwise shift operators	
Name	Symbol	Name	Symbol
and	&	left shift	<<
or		right shift	>>
xor	^		
not	~		

The *and* (&), *or* (|), and *xor* (^) operators are used to compare the values of two bits *A* and *B*. The expression

```
A & B // Are A and B both true?
```

returns *true* if and only if *A* and *B* are both *true*:

A	B	A & B
true (1)	true (1)	true (1)
true (1)	false (0)	false (0)
false (0)	false (0)	false (0)
false (0)	true (1)	false (0)

The expression

```
A | B // Is either A or B true, or are A and B both true?
```

returns *true* if either *A* or *B* is *true*, or if *A* and *B* are both *true*:

A	B	A B
true (1)	true (1)	true (1)
true (1)	false (0)	true (1)
false (0)	false (0)	false (0)
false (0)	true (1)	true (1)

The *xor* (^) operator is pronounced “x-or,” in which the *x* stands for “exclusive.” The expression

```
A ^ B // Is A exclusively true, or is B exclusively true?
```

returns *true* if only *A* is *true*, or only *B* is *true*:

A	B	A ^ B
true (1)	true (1)	false (0)
true (1)	false (0)	true (1) // continued on next page

A	B	$A \wedge B$
false (0)	false (0)	false (0)
false (0)	true (1)	true (1)

The *not* operator (\sim) simply returns the inverse value of a given bit:

A	$\sim A$
true (1)	false (0)
false (0)	true (1)

We can perform bitwise comparisons between two objects of the same type. Given

```
int x = 1488009749;
int y = -892037412;
```

for example, the statement

```
int k = x & y;
```

performs an *and* comparison between each bit in *x* and the corresponding bit in *y*, then stores the result in the corresponding bit in *k*:

01011000 10110001 00111010 00010101	// x (1,488,009,749)
& 11001010 11010100 10010110 11011100	// y (-892,037,412)
<hr/>	
01001000 10010000 00010010 00010100	// k (1,217,401,364)

The leftmost bit in *k* (0) is the result of *and*-comparing the leftmost bits in *x* and *y* (0 and 1). The second-leftmost bit in *k* (1) is the result of *and*-comparing the second-leftmost bits in *x* and *y* (1 and 1), etc.

The *or* and *xor* operators work the same way:

int n = x y; // Perform an or comparison between each bit in x and // the corresponding bit in y, then store the result in the // corresponding bit in n	
01011000 10110001 00111010 00010101	// x (1,488,009,749)
11001010 11010100 10010110 11011100	// y (-892,037,412)
<hr/>	
11011010 11110101 10111110 11011101	// n (-621,429,027)

```

int p = x ^ y; // Perform an xor comparison between each bit in x and
                // the corresponding bit in y, then store the result in the
                // corresponding bit in p

01011000 10110001 00111010 00010101 // x (1,488,009,749)
^ 11001010 11010100 10010110 11011100 // y (-892,037,412)

-----
10010010 01100101 10101100 11001001 // p (-1,838,830,391)

```

C++ also provides compound assignment versions of the bitwise *and*, *or*, and *xor* operators. These are similar to the other compound assignment operators that we've been using all along. Just as

```

x += y;           // Addition assignment operator
x -= y;           // Subtraction assignment operator
x *= y;           // Multiplication assignment operator
x /= y;           // Division assignment operator
x %= y;           // Modulo assignment operator

```

are shorthand for

```

x = x + y;
x = x - y;
x = x * y;
x = x / y;
x = x % y;

```

the statements

```

x &= y;           // Bitwise and assignment operator
x |= y;           // Bitwise or assignment operator
x ^= y;           // Bitwise xor assignment operator

```

are shorthand for

```

x = x & y;         // Evaluate (x & y), then assign the result to x
x = x | y;         // Evaluate (x | y), then assign the result to x
x = x ^ y;         // Evaluate (x ^ y), then assign the result to x

```

Applying a bitwise *not* (\sim) to an object returns a new object in which the bits are inverted, leaving the original object unchanged. In the statement

```

int q = ~x;        // Take the inverse value of each bit in x and store it in
                  // the corresponding bit in q

```

for example, the expression

```
~x
```

returns an unnamed *int* containing the inverse bits of *x*. The unnamed *int* is then assigned to *q*, and *x* is unchanged:

```

01011000 10110001 00111010 00010101      //  x (1,488,009,749)
10100111 01001110 11000101 11101010      // ~x (-1,488,009,750)
10100111 01001110 11000101 11101010      //  q (-1,488,009,750)

```

To invert the bits in *x*, we can apply the *not* operator to *x* and assign the result back to *x* in a single statement:

```
x = ~x;      // Evaluate ~x, then assign the result back to x
```

Shifting an object by *N* bits moves the value of each bit *N* places to the left or right. There are several methods of shifting, but the most basic ones are the *logical left shift*, *logical right shift*, and *arithmetic right shift*.

To perform a 3-bit *logical left shift*, for example, we move the value of each bit 3 places to the left. This discards the 3 leftmost values, after which we fill the 3 rightmost values with 0.

```

sign bit                                // Shifted bits are underlined
|  

00000000 10110001 00111010 01011001      // Original value = 11,614,809  

|__|_____|  

|  

00000101 10001001 11010010 11001000      // New value      = 92,918,472

sign bit  

|  

11111000 10110001 00111010 01011001      // Original value = -122,602,919  

|__|_____|  

|  

11000101 10001001 11010010 11001000      // New value      = -980,823,352

```

In both of these examples, the new value is exactly 8 (2^3) times the original value. An *N*-bit logical left shift multiplies the original value by 2^N , but only if both of the following conditions are met:

- The *N* leftmost bits of the original value are all the same (either all 0 or all 1)
- The shift doesn't change the value of the sign bit

If the *N* leftmost bits of the original value aren't the same, or the shift changes the value of the sign bit, we'll end up with a condition known as *overflow*, in which we don't have enough space (bits) to store the new value. If overflow occurs, the multiplication property won't apply (the new value won't be exactly 2^N times the original value):

```

// 3-bit logical left shift resulting in overflow (the 3 leftmost bits,
// 0-1-0, aren't the same)

sign bit
|  

01000000 10110001 00111010 01011001      // Original value = 1,085,356,633  

|__|_____|  

|  

00000101 10001001 11010010 11001000      // New value      = 92,918,472

```

```
// 3-bit logical left shift that changes the sign bit (from 0 to 1)

sign bit
|
00010000 10110001 00111010 01011001      // Original value = 280,050,265
|__|           |__|
|   |           |
10000101 10001001 11010010 11001000      // New value      = -2,054,565,176
```

A *logical right shift* is the mirror image of a logical left shift. To perform a 3-bit logical right shift, for example, we move the value of each bit 3 places to the right (thus discarding the 3 rightmost values). We then fill the 3 leftmost values with 0:

```
sign bit
|
00000010 10110001 00111010 01011000      // Original value = 45,169,240
|__|           |__|
|   |           |
00000000 01010110 00100111 01001011      // New value      = 5,646,155
```

In this example, the new value is exactly $1/8$ ($1/2^3$) of the original value. An N -bit logical right shift divides the original value by 2^N , but only if both of the following conditions are met:

- The N rightmost bits of the original value are all 0
- The original value is a non-negative number

If the N rightmost bits of the original value aren't all 0, then overflow will occur and the division property won't apply (the new value won't be exactly $1/2^N$ of the original value):

```
// 5-bit logical right shift resulting in overflow (the 5 rightmost bits,
// 1-1-0-0-1, aren't all 0)

sign bit
|
00000000 00110100 01010010 10011001      // Original value = 3,429,017
|_____|           |_____|           |
|   |           |           |
00000000 00000001 10100010 10010100      // New value      = 107,156
```

If the original value is a negative number, then the shift will change the value of the sign bit (from 1 to 0), which also breaks the division property:

```
// 3-bit logical right shift of a negative number (changes the sign bit
// from 1 to 0)

sign bit
|
11111111 11110001 00111010 01011000      // Original value = -968,104
|__|           |__|
|   |           |
00011111 11111110 00100111 01001011      // New value      = 536,749,899
```

An *arithmetic right shift* solves this problem by filling the N leftmost values with the original sign bit, which preserves the division property for both positive and negative numbers. To perform a 3-bit arithmetic right shift, for example, we move the value of each bit 3 places to the right (thus discarding the 3 rightmost values). We then fill the 3 leftmost values with the original sign bit:

```

sign bit
|
01010110 10110001 00111010 10001000 // Original value = 1,454,455,432
|__|_____|_____|_____
|   |   |   |
00001010 11010110 00100111 01010001 // New value      = 181,806,929
|
The 3 leftmost values are filled
with the original sign bit (0)

```

```

sign bit
|
10010110 10110001 00111010 01010000 // Original value = -1,766,770,096
|__|_____|_____|_____
|   |   |   |
11110010 11010110 00100111 01001010 // New value      = -220,846,262
|
The 3 leftmost values are filled
with the original sign bit (1)

```

In both of these examples, the new value is exactly $1/8$ ($1/2^3$) of the original value. An N -bit arithmetic right shift divides the original value by 2^N , as long as the N rightmost bits of the original value are all 0.

The following table summarizes the differences between the 3 shift types:

Type	Fill value	Mathematical property of an N-bit shift	Mathematical property applies only if
Logical left shift	0	Multiplication by 2^N	the N leftmost bits are the same, and the sign bit doesn't change
Logical right shift	0	Division by 2^N	the N rightmost bits are all 0, and the number is non-negative
Arithmetic right shift	Original sign bit	Division by 2^N	the N rightmost bits are all 0

In C++, the bitwise shift operators (like the logical operators) return a new object, leaving the original object unchanged. Given

```
unsigned int x = 360;
```

for example, the expression

```
x << 3
```

returns a new *unsigned int* containing the bit pattern of *x*, logically shifted 3 bits to the left. The statement

```
unsigned int y = (x << 3);
```

therefore assigns *y* a value of 2,880 (360×2^3), leaving *x* unchanged:

<u>00000000</u>	<u>00000000</u>	<u>00000001</u>	<u>01101000</u>	// x (360)
<u>00000000</u>	<u>00000000</u>	<u>00001011</u>	<u>01000000</u>	// x << 3 (2,880)
00000000	00000000	00001011	01000000	// y (2,880)

Conversely, the expression

```
x >> 3
```

returns a new *unsigned int* containing the bit pattern of *x*, logically shifted 3 bits to the right. The statement

```
unsigned int z = (x >> 3);
```

therefore assigns *z* a value of 45 ($360 / 2^3$), leaving *x* unchanged:

<u>00000000</u>	<u>00000000</u>	<u>00000001</u>	<u>01101000</u>	// x (360)
<u>00000000</u>	<u>00000000</u>	<u>00000000</u>	<u>00101101</u>	// x >> 3 (45)
00000000	00000000	00000000	00101101	// z (45)

In C++, the bitwise shift operators perform a logical left / right shift, but only when applied to an object of *unsigned integral type* (a type that stores an unsigned integer value, such as *unsigned int* and *unsigned char*). We can, in fact, directly assign integer values to *chars*. The statement

```
unsigned char c = 40;
```

for example, is equivalent to

```
unsigned char c = '(';
```

because 40 is the ASCII code for the left parenthesis character. The expressions

```
c << 1
c >> 1
```

would then return new *unsigned chars* containing the bit pattern of *c*, logically shifted 1 place to the left / right:

```
00101000    // c      (40          = ASCII code for '(')
01010000    // c << 1 (40 x 21 = 80 = ASCII code for 'P')

00101000    // c      (40          = ASCII code for '(')
00010100    // c >> 1 (40 / 21 = 20 = ASCII code for "Device control 4")
```

When applied to an object of *signed integral type* (a type that stores a signed integer value), the left shift operator (`<<`) performs a logical left shift, but the right shift operator (`>>`) is implementation-dependent. The following table summarizes the differences:

Operator	Object type	Shift type
Left shift (<code><<</code>)	Unsigned integral	Logical left
Right shift (<code>>></code>)	Unsigned integral	Logical right
Left shift (<code><<</code>)	Signed integral	Logical left
Right shift (<code>>></code>)	Signed integral	Implementation-dependent

We now have everything we need to write the function (*printBinaryRep.h*, line 14)

```
void printByte(unsigned char byte);
```

which prints the value of each bit in the given *byte*. We're using an *unsigned char* parameter because we need to perform logical right shifts to get the value of each bit. Suppose, for example, that the given *byte* is

```
B
01001101    // byte
```

We start with the leftmost bit. To get the value of the current bit (*B*), we logically shift it to the rightmost position, then perform an *and* comparison with the number 1 (00000001). The resulting byte (*char*) is guaranteed to be either 00000000 (if *B* is 0) or 00000001 (if *B* is 1). We can then store that value as an *int* and print it:

```
B
01001101    // byte

B
00000000    // byte >> 7 (Logically shift B to the rightmost position)
& 00000001    // 1

_____
00000000    // char containing the value of B (0)
```

To get the value of the next bit we repeat the process, using a 6-bit logical right shift:

B
01001101 // byte

B
00000001 // byte >> 6
& 00000001 // 1

00000001 // B (1)

B
01001101 // byte

B
00000010 // byte >> 5
& 00000001 // 1

00000000 // B (0)

B
01001101 // byte

B
00000100 // byte >> 4
& 00000001 // 1

00000000 // B (0)

B
01001101 // byte

B
00001001 // byte >> 3
& 00000001 // 1

00000001 // B (1)

B
01001101 // byte

B
00010011 // byte >> 2
& 00000001 // 1

00000001 // B (1)

```

      B
01001101    // byte

      B
00100110    // byte >> 1
& 00000001    // 1



---


      00000000    // B (0)

```

```

      B
01001101    // byte

      B
01001101    // byte >> 0
& 00000001    // 1



---


      00000001    // B (1)

```

printByte is defined in lines 65-72 (*printBinaryRep.h*). The loop performs a total of *CHAR_BIT* iterations, one for each bit in the *byte*. *shiftLength* is the number of bits that we're shifting in the current iteration ({7, 6, 5, 4, 3, 2, 1, 0} in the above example).

Now that we can easily print the bits in a single byte, printing a sequence of bytes is relatively simple. To do that, we'll write another function (line 13),

```
void printByteSequence(const unsigned char* first, std::size_t length);
```

where *first* is a pointer to the first byte in the sequence, and *length* is total number of bytes in the sequence.

We'll print the bytes in both forward and reverse order, for reasons that we'll discuss shortly. We start with the first byte and simply traverse the sequence, printing the *currentByte* in each iteration (lines 46-53). When the loop terminates, *currentByte* points to the one-past-the-last byte, so we back up to the last byte (line 55), then print the sequence in reverse order (lines 57-62).

We can now write a function to print the binary representation of any object of type *T* (lines 10-11):

```
template <class T>
void printBinaryRep(const T& value);
```

The function is defined in lines 16-26. Line 21 simply prints the object's typename and value. Line 23 then calls *printByteSequence*, passing a pointer to the object's first byte,

```
reinterpret_cast<const unsigned char*>(&value)
```

along with the total number of bytes occupied by the object,

```
sizeof(T)
```

The address of the object (`&value`) is the address of the first byte, but `&value` returns a `T*` (pointer to a `T`). The `reinterpret_cast` explicitly converts that `T*` to a `const unsigned char*` (pointer to a byte), which then gets passed to `printByteSequence`.

This version of `printBinaryRep` will work fine for basic types like `int` and `double`: it'll print all 4 bytes of an `int` and all 8 bytes of a `double`, which directly correspond to the object's "value."

But what about a `std::string`? As we discussed in Chapter 11.1, a `std::string` internally resembles a `std::vector<char>`. It doesn't directly contain the `chars` that correspond to its "value," but rather a `pointer` to a dynamically-allocated array of `chars`. It also contains other data members for storing the size, capacity, etc. `printBinaryRep` would therefore print the binary representation of the string's *data members* as opposed to its *value*, which isn't exactly the behavior that we want.

To print the binary representation of a `std::string`'s *value* (i.e. the its actual characters), we need to write a specialization of `printBinaryRep` (lines 28-40). This version also calls `printByteSequence`, but instead of passing the address and size of the whole `std::string` object, it passes the address and size of the character array only (lines 36-37),

```
s.c_str()      // The address of the first character in the array,
               // as opposed to &s (the address of the first byte of the
               // whole std::string object)

s.size()       // The number of characters in the array,
               // as opposed to sizeof(std::string) (the number of bytes
               // occupied by the whole std::string object)
```

We'll apply this concept again in the next section, when implementing a hash function.

We can now try `printBinaryRep` on a few different types. Lines 19-21 (*main.cpp*),

```
printBinaryRep('x');      // char
printBinaryRep(true);    // bool
printBinaryRep(false);   // bool
```

generate the output

```
char x
Bytes (forward):
01111000                  // 120 (ASCII code for 'x')
Bytes (reverse):
01111000

bool 1
Bytes (forward):
00000001
Bytes (reverse):
00000001
```

```

bool 0
Bytes (forward):
00000000
Bytes (reverse):
00000000

```

When we try it on an *int* (line 22),

```
printBinaryRep(137494);
```

the bytes actually appear in the opposite order of what we would expect:

```

int 137494
Bytes (forward):
00010110 00011001 00000010 00000000      // In-memory representation
Bytes (reverse):
00000000 00000010 00011001 00010110      // Standard binary format
                                                //  $2^{17} + 2^{12} + 2^{11} + 2^8 + 2^4 + 2^2 + 2^1$ 

```

This is due to a concept known as *endianness*, which refers to the in-memory ordering of bytes. There are two types of systems:

- *Big-endian systems*, which store the *most* significant byte first (at the lowest address)
- *Little-endian systems*, which store the *least* significant byte first (at the lowest address)

The *most significant byte (MSB)* is the byte containing the largest power of 2; the *least significant byte (LSB)* is the byte containing the smallest power of 2. In the 4-byte representation of 137,494, the MSB is 00000000 and the LSB is 00010100:

$ \begin{array}{ccccccc} & & 2^{11} & & 2^2 & 2^0 \\ \text{MSB} & & & \text{LSB} & & \\ 00000000 & 00000010 & 00011001 & 00010110 & & & \\ & & & & & \\ 2^{30} & 2^{17} & 2^{12} & 2^8 & 2^4 & 2^1 \end{array} $ Lowest -----> Highest address address	$ \begin{array}{c} 2^{11} \quad 2^2 \quad 2^0 \\ \quad \quad \\ \text{MSB} \\ // Big-endian format \\ // MSB is stored first (at the \\ // lowest address) \end{array} $ $ \begin{array}{c} 2^2 \quad 2^0 \quad 2^{11} \\ \text{LSB} \quad / \quad \text{MSB} \\ 00010110 \quad 00011001 \quad 00000010 \quad 00000000 \\ \quad \quad \quad \quad \quad \\ 2^4 \quad 2^1 \quad 2^{12} \quad 2^8 \quad 2^{17} \quad 2^{30} \end{array} $ $ \begin{array}{c} // Little-endian format \\ // LSB is stored first (at the \\ // lowest address) \end{array} $
--	---

Note how the only difference between the two formats is the order of *bytes*; within each byte, the order of *bits* is identical. Line 23,

```
printBinaryRep(-43);
```

generates the output

```
int -43
Bytes (forward):
11010101 11111111 11111111 11111111
Bytes (reverse):
11111111 11111111 11111111 11010101
```

My system is little-endian, so if we take the reverse byte sequence,

```
11111111 11111111 11111111 11010101
```

invert the bits, and add 1, we end up with the binary representation of 43:

```
00000000 00000000 00000000 00101010
+ 00000000 00000000 00000000 00000001
-----
00000000 00000000 00000000 00101011 // 32 + 8 + 2 + 1 = 43
```

This confirms that my system uses two's complement format. We then then try a *double* (line 24),

```
printBinaryRep(-2.875);
```

which generates the output

```
double -2.875
Bytes (forward):
00000000 00000000 00000000 00000000 00000000 00000000 00000111 11000000
Bytes (reverse):
11000000 00000111 00000000 00000000 00000000 00000000 00000000 00000000
```

In the context of floating-point types, the MSB is the byte containing the sign bit (in IEEE 754 format). If we take the reverse byte sequence (again, because my system is little-endian),

```
11000000 00000111 00000000 00000000 00000000 00000000 00000000 00000000
```

we can confirm that my system uses binary64 format:

```
11000000 00000111 00000000 00000000 00000000 00000000 00000000 00000000
| \ / |
| biased |
| exponent |
| |
sign |---- stored mantissa -----
sign = 1 (negative)

true mantissa = <sign: +/- for 0/1> <1.> <stored mantissa>
                = -1.01112
```

```

biased exponent = true exponent + bias
true exponent   = biased exponent - bias
                  =  $1000000000_2 - 1023$ 
                  =  $1024_{10} - 1023$ 
                  = 1

base-2 scientific notation = true mantissa  $\times 2^{\text{true exponent}}$ 
                           =  $-1.0111_2 \times 2^1$ 

base-10 representation      =  $-1.0111_2 \times 2^1$ 
                           =  $-10.111_2$ 
                           =  $-10_2 + -0.111_2$ 
                           =  $-(2 + 0)_{10} + -(1/2 + 1/4 + 1/8)_{10}$ 
                           =  $-2.875_{10}$ 

```

Lines 25-26,

```

printBinaryRep("umbra");           // C-string argument
printBinaryRep(string("umbra"));    // std::string argument

```

call the default version and *std::string* specialization of *printBinaryRep*, generating the output

```

char [6] umbra
Bytes (forward):
01110101 01101101 01100010 01110010 01100001 00000000
Bytes (reverse):
00000000 01100001 01110010 01100010 01101101 01110101

std::string umbra
Bytes (forward):
01110101 01101101 01100010 01110010 01100001
Bytes (reverse):
01100001 01110010 01100010 01101101 01110101

```

Endianness doesn't apply here, so the forward byte sequences are in the order that we expect. The *std::string* specialization doesn't print the null character at the end because it gets the total number of characters via *std::string*'s *size* method (which, as we discussed in Chapter 11.1, doesn't include the null-terminating character):

```

printBinaryRep("umbra");


---


Byte          01110101 01101101 01100010 01110010 01100001 00000000
Base 10 (ASCII code) 117      109      98       114      97       0
Glyph        u         m         b         r         a         \0

printBinaryRep(std::string("umbra"));


---


Byte          01110101 01101101 01100010 01110010 01100001
Base 10 (ASCII code) 117      109      98       114      97
Glyph        u         m         b         r         a

```


Part 12: Hash Tables

12.1: The FNV Hash Function

Source files and folders

- *FnvHash*

Chapter outline

- *Hash tables vs. linked data structures*
- *Collision resolution via bucket addressing*
- *Implementing the FNV hash function*

Although red-black trees, B-trees, and skip lists operate in logarithmic time on average, their linked structure guarantees that certain elements will take longer to find than others. A *hash table* addresses this problem by using a *hash function* to convert key values to array indexes, and storing its elements (key-mapped pairs) in an array, where they can be accessed in constant time.

Depending on the number of unused cells in the array, however, a hash table may be less memory-efficient than a linked structure. The other downside to this approach is that the elements aren't sorted by their key values.

The process of converting a key value to an array index is called *hashing*, and the array index derived from a given key value is called the *hash* of that key. Suppose, for example, that we're storing a set of country-capital (*string-string*) pairs in an array with a 10-element capacity. One possible hash function is

```
Hash(key) = (ASCII code of first letter + ASCII code of last letter) % 10
```

The modulus division by 10 guarantees that the hash value will be between 0 and 9 inclusive (and thus be a valid array index). The pairs

```
(China, Beijing)
(Peru, Lima)
(Finland, Helsinki)
```

for example, would be stored at indexes 4, 7, and 0:

```
Hash("China")      = (ASCII code of 'C' + ASCII code of 'a') % 10
                      = (67 + 97) % 10
                      = 4
```

```

Hash("Peru")      = (ASCII code of 'P' + ASCII code of 'u') % 10
                  = (80 + 117) % 10
                  = 7

```

```

Hash("Finland")   = (ASCII code of 'F' + ASCII code of 'd') % 10
                  = (70 + 100) % 10
                  = 0

```

Index Element

0	(Finland, Helsinki)
1	
2	
3	
4	(China, Beijing)
5	
6	
7	(Peru, Lima)
8	
9	

When a hash function generates the same value for two or more different keys, a *collision* is said to occur. Suppose, for example, that we try to insert the pair (Japan, Tokyo):

```

Hash("Japan")     = (ASCII code of 'J' + ASCII code of 'n') % 10
                  = (74 + 100) % 10
                  = 4                      // Collision: same hash as "China"

```

To avoid this collision, we could increase the size of the array to say, 100, and modify the hash function accordingly:

```

Hash(key)         = (ASCII code of first letter +
                     ASCII code of last letter) % 100

Hash("China")     = (67 + 97) % 100 = 64
Hash("Peru")       = (80 + 117) % 100 = 97
Hash("Finland")   = (70 + 100) % 100 = 70
Hash("Japan")      = (74 + 100) % 100 = 74

```

Alternatively, we could keep the array size at 10 and incorporate the middle letter of the key value:

```

Hash(key)         = (ASCII code of first letter +
                     ASCII code of middle* letter
                     ASCII code of last letter) % 10

// *If there is no middle letter (because the key has an even number of
// letters), use the first letter of the right half (so for "Peru," we
// take the right half, "ru," and use the first letter, 'r')

Hash("China")     = (67 + 105 + 97) % 10 = 9
Hash("Peru")       = (80 + 114 + 117) % 10 = 1

```

```
Hash("Finland") = (70 + 108 + 100) % 10 = 8
Hash("Japan")   = (74 + 112 + 100) % 10 = 6
```

If we know all of the keys in advance, we can create a *perfect* (collision-free) *hash function*. A perfect hash function that also uses every cell of the array is called a *minimal perfect hash function*, because it requires the smallest possible array size.

In a general-purpose hash table, where we don't know the keys in advance, collisions are inevitable so we need a way to resolve them. We'll use a method called *bucket addressing*, in which each cell of the array contains a *bucket* (expandable list) of elements. Each bucket can accommodate as many colliding elements as necessary:

```
Hash(key)      = (ASCII code of first letter +
                  ASCII code of last letter) % 10
```

```
Hash("China")   = 4
Hash("Peru")    = 7
Hash("Finland") = 0
Hash("Japan")   = 4      // Collision
Hash("Egypt")   = 5
Hash("Bulgaria")= 3
Hash("Chile")   = 8
Hash("Cameroon")= 7      // Collision
Hash("Barbados")= 1
Hash("Canada")  = 4      // Collision
```

Index	Bucket
0	(Finland, Helsinki)
1	(Barbados, Bridgetown)
2	
3	(Bulgaria, Sofia)
4	(China, Beijing) (Japan, Tokyo) (Canada, Ottawa)
5	(Egypt, Cairo)
6	
7	(Peru, Lima) (Cameroon, Yaounde)
8	(Chile, Santiago)
9	

0	(Finland, Helsinki)
1	(Barbados, Bridgetown)
2	
3	(Bulgaria, Sofia)
4	(China, Beijing) (Japan, Tokyo) (Canada, Ottawa)
5	(Egypt, Cairo)
6	
7	(Peru, Lima) (Cameroon, Yaounde)
8	(Chile, Santiago)
9	

To locate a key, we calculate the hash value (index) and search the corresponding bucket. Larger buckets take longer to search, so the fewer collisions the better (in a collision-free table, each bucket contains at most 1 element).

Our hash table implementation will use the *FNV hash function*, named for its inventors Glenn Fowler, Landon Curt Noll, and Phong Vo. In addition to providing a good balance of speed and *dispersion* (collision avoidance), FNV is also a general-purpose hash function, applicable to any datatype.

FNV generates a hash value through a combination of *xor*-manipulation and prime number multiplication. The procedure is

```

offsetBasis = <constant value>;
fnvPrime = <constant value>

hash = offsetBasis;

for each byte in the given key value
{
    hash ^= byte;                                // hash = hash ^ byte;
    hash *= fnvPrime;                            // hash = hash * fnvPrime;
}

return hash;

```

The exact values of *offsetBasis* and *fnvPrime* depend on the size of the hash to be generated:

Hash size	offsetBasis	fnvPrime
32-bit	2166136261	16777619
64-bit	14695981039346656037	1099511628211

Source:

<http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-param>

Also includes the recommended *offsetBasis* and *fnvPrime* for generating 128, 256, 512, and 1024-bit hashes

Suppose, for example, that we're generating the 32-bit hash of the 5-byte key value "Egypt":

Glyph	E	g	y	p	t
ASCII code	01000101	01100111	01111001	01110000	01110100

```

offsetBasis = 2166136261;
fnvPrime = 16777619;

hash = offsetBasis;                                // hash = 2166136261

Iteration 1:

hash ^= byte;
  11000101 10011101 00011100 10000001      // hash (2166136261)
  ^ 01000101                                // byte ('E')

  10000000 10011101 00011100 10000001      // hash = 2166136192

hash *= fnvPrime;                                // hash = 3222007936

```

Iteration 2:

```

hash ^= byte;

 10000000 11110000 00001011 11000000
^ 01100111
-----
 11100111 11110000 00001011 11000000 // hash = 3222008039

hash *= fnvPrime; // hash = 969685925

```

Iteration 3:

```

hash ^= byte;

 10100101 00111011 11001100 00111001
^ 01111001
-----
 11011100 00111011 11001100 00111001 // hash = 969685980

hash *= fnvPrime; // hash = 3632413524

```

Iteration 4:

```

hash ^= byte;

 01010100 00111011 10000010 11011000
^ 01110000
-----
 00100100 00111011 10000010 11011000 // hash = 3632413476

hash *= fnvPrime; // hash = 4177729964

```

Iteration 5:

```

hash ^= byte;

 10101100 00011001 00000011 11111001
^ 01110100
-----
 11011000 00011001 00000011 11111001 // hash = 4177730008

hash *= fnvPrime; // hash = 3621891848

```

To convert the hash value (3,621,891,848) to an array index, we modulus divide it by the array size:

Array size	Array index = Hash value % Array size
10	8
100	48
1000	848
10000	1848

The function (*FnvHash.h*, line 27)

```
std::size_t fnvHash(const unsigned char* first, std::size_t length);
```

generates an FNV hash, where *first* points to the first byte of the key value and *length* is the length (in bytes) of the key value. The implementation (lines 43-59) is nearly identical to the above pseudocode.

On my system, *size_t* (the type we're using to represent hash values) is 32 bits, so I've used the corresponding values of *offsetBasis* and *fnvPrime* (lines 47-48). If *size_t* on your system is a different size, you should change *offsetBasis* and *fnvPrime* accordingly. You can obtain the size of *size_t* (in bits) via the expression

```
sizeof(std::size_t) * CHAR_BIT
```

where *sizeof(std::size_t)* returns the number of bytes per *size_t* and *CHAR_BIT* returns the number of bits per byte (which, on my system, is 4 bytes x 8 bits per byte = 32 bits).

We can now create the *FnvHash* function object (lines 9-16). The template parameter *Key* (line 9) is the type of key value to be hashed. *argument_type* (line 12) is the type of argument taken by the function object, and *result_type* (line 13) is the return type.

The function call operator (line 15)

```
std::size_t operator()(const Key& key) const;
```

returns the hash of the given key value by simply calling *fnvHash* (lines 29-33). Just like in Chapter 11.4, we need need to *reinterpret_cast* the address of the key (*&key*, of type *const Key**) to a pointer to a byte (*const unsigned char**).

The other concept that we're reapplying from Chapter 11.4 is *std::string* specialization. To see why the default version of *FnvHash* won't work for *std::strings*, consider the following example:

```
FnvHash<string> hash;
string s = "Mercury";
string t = "Mercury";
```

Although *s* and *t* have the exact same value ("Mercury"), the expressions

```
hash(s)
```

```
hash(t)
```

would return *different* values. This is because the default version of *FnvHash* wouldn't hash the *values* of *s* and *t* ("Mercury"), but rather the *objects* *s* and *t*. And because *s* and *t* store their elements (*chars*) in different arrays, their internal pointers have different values. Hashing the entire objects (which include those pointers) would therefore generate different hash values.

The *std::string* specialization of *FnvHash* is defined in lines 18-25. The only difference from the default version lies in how we call *fnvHash* (lines 35-41): instead of passing the *std::string* object, we pass the characters only. This is the exact same technique that we used in Chapter 11.4, when implementing the *std::string* specialization of *printBinaryRep*.

Our test program (*main.cpp*) constructs *FnvHash* objects of type *int*, *double*, and *string* (lines 11-13). Lines 15-17,

```
cout << hashInt(67) << endl;
cout << hashDouble(3.14159) << endl;
cout << hashString("perihelion") << endl;
```

generate the output

```
2427613158      // 4 iterations (assuming sizeof(int) = 4)
2064320989      // 8 iterations (assuming sizeof(double) = 8)
988373770       // 10 iterations (1 for each character in "perihelion")
```


12.2: Introducing the *HashTable* Class

Source files and folders

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

Chapter outline

- *Implementing the constructor, destructor, basic accessor methods, and insertion*

Now that we have a generic hash function, we're ready to begin implementing the *HashTable* class. In the previous chapter we introduced the concept of bucket addressing, wherein each index of the hash table's internal array refers to an expandable list (bucket) of colliding elements. Our implementation, however, will utilize an extra layer of indirection, allowing us to more easily support traversal and erasure.

A *HashTable* has 4 template parameters (*HashTable.h*, lines 15-18):

- *Key*, the type of key values
- *Mapped*, the type of mapped values
- *Hash*, the type of function object used to hash key values (the default is *FnvHash<Key>*)
- *Predicate*, the type of function object used to compare key values for equality, while searching within buckets (the default is *std::equal_to<Key>*)

_hash is the stored *Hash* function, and *_isEqual* is the stored *Predicate* (lines 71-72). The *Hash* and *Predicate* types also go by the names *hasher* and *key_equal* (lines 25-26).

_elements (lines 24, 27, 73) is a doubly-linked list containing the elements (*ElementList*). *HashTable*'s standard member types (*size_type*, *iterator*, etc.) all come from *ElementList* (lines 28-35).

Each *Bucket* (line 57) is a singly-linked list of iterators to *_elements*. *_buckets* (lines 58, 74) is a *vector* of pointers to buckets (*BucketArray*). *_alloc* is the *allocator* used to create and destroy *Buckets* (line 75).

Consider, for example, the *HashTable*

```
    _elements
    ElementList
    (list<pair<const string, string>>)
```

```
(China, Beijing)
(Barbados, Bridgetown)
(Egypt, Cairo)
(Chile, Santiago)
```

```

(Bulgaria, Sofia)
(Peru, Lima)
(Japan, Tokyo)
(Finland, Helsinki)
(Cameroon, Yaounde)

    _buckets
BucketArray           Bucket
(vector<Bucket*>)   ForwardListSize<ElementList::iterator>


---


0      [iter(Finland, Helsinki)]
1      [iter(Barbados, Bridgetown)]
2      []
3      [iter(Bulgaria, Sofia)]
4      [iter(China, Beijing), iter(Japan, Tokyo)]
5      [iter(Egypt, Cairo)]
6      []
7      [iter(Peru, Lima), iter(Cameroon, Yaounde)]
8      [iter(Chile, Santiago)]
9      []

```

The elements (*pair<const string, string>*) themselves are stored in the *ElementList*. We can traverse the entire *HashTable* by simply traversing *_elements* from *begin* to *end* [(China, Beijing), (Barbados, Bridgetown), (Egypt, Cairo), (Chile, Santiago)...].

Each element of *_buckets* is a pointer to a dynamically-allocated *Bucket* (*ForwardListSize* of iterators to *_elements*). *_buckets[4]*, for example, is a pointer to a *ForwardListSize* containing two iterators, one to (China, Beijing), and one to (Japan, Tokyo). *_buckets[2]*, *_buckets[6]*, and *_buckets[9]* point to empty *Buckets*.

To find the element with the key value “Cameroon,” we use the *_hash* function to obtain the array index (7), then search the corresponding *Bucket* until we find the element whose key value *_isEqual* to “Cameroon.”

The *load factor* of a hash table is the average bucket size (average number of elements per bucket), calculated using the formula

$$\text{load factor} = \text{total elements} / \text{bucket count}$$

The *bucket count* (total number of buckets) includes *all* buckets, empty and non-empty. The table in the above diagram, for example, has a load factor of 0.9 (9 elements / 10 buckets); each bucket contains 0.9 elements, on average. A table with 48 elements and 20 buckets has a load factor of 2.4 (48 elements / 20 buckets); each bucket contains 2.4 elements, on average.

The load factor represents a trade-off between search time and memory usage, as summarized by the following table:

load factor	average bucket size	average search time	memory usage
lower	smaller (fewer elements per bucket)	faster	higher (larger BucketArray, more buckets)
higher	larger (more elements per bucket)	slower	lower (smaller BucketArray, fewer buckets)

The *max load factor* is the maximum allowable load factor, set by the user. This value is stored in the data member `_maxLoadFactor` (line 70).

The *capacity* of a hash table is the maximum number of elements it can store without exceeding the max load factor, calculated using the formula

$$\text{capacity} = \text{bucket count} \times \text{max load factor}$$

Suppose, for example, that a table contains 10 buckets with a `_maxLoadFactor` of 2.5:

$$\begin{aligned}\text{capacity} &= \text{bucket count} \times \text{max load factor} \\ &= 10 \text{ buckets} \times 2.5 \text{ elements / bucket} \\ &= 25 \text{ elements}\end{aligned}$$

We can store a total of 25 elements, at which point the table will become *full* (its load factor will be equal to its max load factor):

$$\begin{aligned}\text{load factor} &= \text{total elements} / \text{total buckets} \\ &= 25 \text{ elements} / 10 \text{ buckets} \\ &= 2.5 \text{ elements / bucket} \\ &= \text{max load factor}\end{aligned}$$

According to the formula, we can increase the capacity by increasing the bucket count, max load factor, or both. Increasing the capacity via the max load factor alone is easy since all we have to do is adjust the value (`_maxLoadFactor`). Increasing the bucket count, however, invalidates the index values of all the stored keys. The process of recalculating these index values (using the new bucket count) and updating the buckets accordingly is called a *rehash*. Before implementing `HashTable`'s rehash and insertion methods, let's write some of the simpler member functions.

`_createBucket` (`HashTable.h`, line 67) allocates memory for and constructs a new `Bucket`, then returns a pointer to the new `Bucket` (`memberFunctions_1.h`, lines 188-196).

`_destroyBucket` (`HashTable.h`, line 68) destroys the given `bucket` and deallocates the associated memory (`memberFunctions_1.h`, lines 198-204).

The default constructor (`HashTable.h`, line 37) initializes the `_maxLoadFactor` to 1.0 and creates 10 `Buckets` (`memberFunctions_1.h`, lines 5-12). The range-based `for` loop (lines 10-11) traverses each pointer (`bucket`) in `_buckets`, and is equivalent to

```

for (BucketArray::iterator bucket = _buckets.begin();
    bucket != _buckets.end();
    ++bucket)
{
    *bucket = _createBucket();
}

```

where `*bucket` is the current element (`Bucket*`). The other data members (`_hash`, `_isEqual`, `_elements`, and `_alloc`) are implicitly initialized via their own default constructors.

The destructor (*HashTable.h*, line 38) destroys all buckets by passing each of the pointers in the `BucketArray` to `_destroyBucket` (*memberFunctions_1.h*, lines 14-19). The elements (*pair<const Key, Mapped>*) themselves are destroyed by the *ElementList*'s destructor.

`empty` and `size` (*HashTable.h*, lines 40-41) simply call the corresponding methods on the *ElementList* (*memberFunctions_1.h*, lines 21-32).

`bucket_count` (*HashTable.h*, line 43) returns the total number of *Buckets*, empty and non-empty (*memberFunctions_1.h*, lines 41-46).

The private member function `_index` (*HashTable.h*, lines 60, 63) returns the *BucketArray* index of the given *key* value by taking the hash and modulus dividing it by the `bucket_count` (*memberFunctions_1.h*, lines 147-152).

The public member function `bucket` (*HashTable.h*, line 42) returns the *BucketArray* index of the given *key* value by simply calling `_index` (*memberFunctions_1.h*, lines 34-39).

`bucket_size` (*HashTable.h*, line 44) returns the size of the *Bucket* at the given *index* (*memberFunctions_1.h*, lines 48-53).

`begin` and `end` (*HashTable.h*, lines 45-46, 52-53) return *iterators / const_iterators* to the first and one-past-the-last elements (*memberFunctions_1.h*, lines 55-67, 95-107).

`hash_function` and `key_eq` (*HashTable.h*, lines 47-48) return the *Hash* function and *Predicate* (*memberFunctions_1.h*, lines 69-81).

`load_factor` and `max_load_factor` (*HashTable.h*, lines 49-50) return the current load factor (total elements / total buckets) and `_maxLoadFactor` (*memberFunctions_1.h*, lines 83-93).

The private member function `_capacity` (*HashTable.h*, line 62) returns the capacity (`bucket_count` x `_maxLoadFactor`) (*memberFunctions_1.h*, lines 140-145).

`insert` (*HashTable.h*, line 54) inserts the `newElement` and returns a *pair<iterator, bool>*:

- If the new element was inserted, *second* is *true* and *first* refers to the newly inserted element
- If the new element wasn't inserted (because the table already contains an element with the

same key value), *second* is *false* and *first* refers to the preexisting element

The procedure is

```
pair<iterator, bool> insert(newElement)
{
    calculate the index of the newElement's key value;

    search the corresponding bucket for a preexisting element
    with the same key;

    if a preexisting element with the same key was found,
        return pair(iterator to preexisting element, false);

    if the table is full,
        increase the capacity (rehash the table);

    insert the newElement in the ElementList;
    insert an iterator to the new element in the corresponding bucket;

    return pair(iterator to new element, true);
}
```

Before beginning the implementation, let's write a separate function to handle the second step (*HashTable.h*, lines 59, 65):

```
BucketIter _findIter(Bucket* bucket, const key_type& key);
```

_findIter searches the given *bucket* for the desired *key* value. If the desired *key* isn't found, the function returns *bucket->end()*. It's worth emphasizing that the return value (*BucketIter* / *Bucket::iterator*) is actually an iterator to an iterator (to a *pair*), not just an iterator to a *pair*. Consider, for example, the table

<pre>elements ElementList (list<pair<const string, string>>)</pre> <hr/> <pre>(China, Beijing) (Barbados, Bridgetown) (Japan, Tokyo) (Bulgaria, Sofia) (Finland, Helsinki)</pre> <hr/> <pre>buckets BucketArray (vector<Bucket*>) Bucket ForwardListSize<ElementList::iterator></pre> <hr/>	<pre>0 [iter(Finland, Helsinki)] 1 [iter(Barbados, Bridgetown)] 2 []</pre>
---	--

```

3           [iter(Bulgaria, Sofia)]
4           [iter(China, Beijing), iter(Japan, Tokyo)]  
  

string key = "Japan";
Index index = _index(key);                                // index = 4  
  

BucketIter bucketIter = _findIter(_buckets[index], key);   // search Bucket 4
                                                               // for "Japan"

```

bucketIter is an iterator to the second element of *Bucket 4*,

```
iter(Japan, Tokyo)
```

which is an iterator to the third element of the *ElementList*. The expression

```
*bucketIter
```

therefore accesses the *iterator* to the pair (Japan, Tokyo), while the expression

```
**bucketIter
```

accesses the pair (Japan, Tokyo) itself.

To implement *_findIter*, we traverse the given *bucket* (*memberFunctions_I.h*, lines 159-167). If the current *pair*'s key value (line 163),

```
(**i).first
```

matches the desired *key*, we return the current *BucketIter*. If we reach the end without finding a match, we return *bucket->end()*.

We can now implement the first few steps of *insert*:

- Calculate the *BucketArray* index of the *newElement*'s key value (line 113)
- Search the corresponding *Bucket* (*_buckets[index]*) for a *preexistingElement* with the same key value as the *newElement* (line 115)
- If we found a preexisting element with the same key as the *newElement*, return a *pair* containing an iterator to the preexisting element and *false* (lines 117-118)

If the new key doesn't already exist, we then check whether the table is full (line 120), and if so, increase the capacity. The formula we'll use to determine the new capacity is

```
new capacity = 1.5 x (current capacity + 2)
```

As discussed earlier, the formula for calculating the current capacity is

```
capacity = bucket count x max load factor
```

Using the initial bucket count (`_bucketArray.size()`) of 10 and `_maxLoadFactor` of 1.0, the initial capacity is 10 elements:

$$\begin{aligned}\text{capacity} &= 10 \text{ buckets} \times 1.0 \text{ elements / bucket} \\ &= 10 \text{ elements}\end{aligned}$$

The next 5 capacity levels are

$$\begin{aligned}1.5 \times (10 + 2) &= 18 \text{ elements} \\1.5 \times (18 + 2) &= 30 \text{ elements} \\1.5 \times (30 + 2) &= 48 \text{ elements} \\1.5 \times (48 + 2) &= 75 \text{ elements} \\1.5 \times (75 + 2) &= 115 \text{ elements}\end{aligned}$$

Once we know the `newCapacity` (line 122), we can use that value to calculate the `newBucketCount` required to attain the `newCapacity` (lines 124-125). Solving the equation

$$\text{new capacity} = \text{new bucket count} \times \text{max load factor}$$

for bucket count (by dividing both sides by the max load factor) yields the formula

$$\text{new bucket count} = \text{new capacity} / \text{max load factor}$$

Suppose, for example, that the current capacity is 10 elements and the max load factor is 1.0:

$$\begin{aligned}\text{new capacity} &= 1.5 \times (\text{current capacity} + 2) \\&= 1.5 \times (10 + 2) \\&= 18 \text{ elements}\end{aligned}$$

$$\begin{aligned}\text{new bucket count} &= \text{new capacity} / \text{max load factor} \\&= 18 / 1.0 \\&= 18 \text{ buckets required to attain the new capacity}\end{aligned}$$

Similarly, if the current capacity is 48 elements and the max load factor is 3.0,

$$\begin{aligned}\text{new capacity} &= 1.5 \times (\text{current capacity} + 2) \\&= 1.5 \times (48 + 2) \\&= 75 \text{ elements}\end{aligned}$$

$$\begin{aligned}\text{new bucket count} &= \text{new capacity} / \text{max load factor} \\&= 75 / 3.0 \\&= 25 \text{ buckets required to attain the new capacity}\end{aligned}$$

If the `newBucketCount` is a fractional value, we need to take the ceiling since the bucket count must be a whole number, and taking the ceiling ensures that the `newBucketCount` will yield the `newCapacity`:

$$\begin{aligned}\text{current capacity} &= 30 \text{ elements} \\ \text{max load factor} &= 2.75\end{aligned}$$

$$\begin{aligned}\text{new capacity} &= 1.5 \times (\text{current capacity} + 2) \\&= 1.5 \times (30 + 2)\end{aligned}$$

```

        = 48 elements

new bucket count = new capacity / max load factor
                  = 48 / 2.75
                  = 17.45 buckets

ceiling(17.45)   = 18 buckets

new capacity     = new bucket count x max load factor
                  = 18 x 2.75
                  = 49.5 elements
                  = 49 elements (ok: meets or exceeds desired capacity of 48)

floor(17.45)    = 17 buckets

new capacity     = new bucket count x max load factor
                  = 17 x 2.75
                  = 46.75 elements
                  = 46 elements (does not meet desired capacity of 48)

```

We can now use the *newBucketCount* to perform a rehash (*memberFunctions_1.h*, line 127), using the separate function (*HashTable.h*, line 66)

```
void _rehash(size_type newBucketCount);
```

This function increases the bucket count to the desired *newBucketCount*, calculates the new index values of all existing keys, and updates the buckets accordingly.

We begin by removing all *ElementList::iterators* from the existing *Buckets* (*memberFunctions_1.h*, lines 175-176), then create enough new *Buckets* to reach the *newBucketCount* (lines 178-179). We then traverse the *ElementList* (*_elements*). In each iteration, we calculate the *newIndex* of the current key then place an iterator to the current element (*pair*) in the corresponding *Bucket* (lines 181-185).

Consider, for example, the table

```

    _elements
    ElementList
    (list<pair<const string, string>>)

```

```

(China, Beijing)
(Barbados, Bridgetown)
(Japan, Tokyo)
(Bulgaria, Sofia)
(Finland, Helsinki)

```

```
// continued on next page
```

<u>buckets</u>	<u>Bucket</u>
<u>BucketArray</u>	<u>ForwardListSize<ElementList::iterator></u>
0	[iter(Finland, Helsinki)]
1	[iter(Barbados, Bridgetown)]
2	[]
3	[iter(Bulgaria, Sofia)]
4	[iter(China, Beijing), iter(Japan, Tokyo)]

using the hash function

```
Hash(key)      = (ASCII code of first letter +
                  ASCII code of last letter) % bucket count
```

Increasing the bucket count to say, 8, invalidates the hash (index) values of all the keys, so we clear out the existing buckets, add 4 new buckets, then rehash the keys (in the *ElementList*) using the new bucket count:

```
Hash("China")    = (67 + 97) % 8 = 4
Hash("Barbados") = (66 + 115) % 8 = 5
Hash("Japan")    = (74 + 110) % 8 = 0
Hash("Bulgaria") = (66 + 97) % 8 = 3
Hash("Finland")  = (70 + 100) % 8 = 2
```

<u>buckets</u>	<u>Bucket</u>
<u>BucketArray</u>	<u>ForwardListSize<ElementList::iterator></u>
0	[iter(Japan, Tokyo)]
1	[]
2	[iter(Finland, Helsinki)]
3	[iter(Bulgaria, Sofia)]
4	[iter(China, Beijing)]
5	[iter(Barbados, Bridgetown)]
6	[]
7	[]
8	[]

Now that we've implemented *_rehash*, we can complete *insert*. After increasing the capacity via the bucket count (*memberFunctions_1.h*, lines 122-127), we need to recalculate the *index* of the *newElement*'s key value (line 129) because changing the bucket count invalidated the original *index* value (line 113).

The table is now ready for the *newElement*, so we insert it at the front of the *ElementList* (line 132), insert an iterator to the new element in the corresponding *Bucket* (lines 134-135), then return a *pair* containing an iterator to the new element and *true* (line 137).

Putting it all together, our test program (*main.cpp*) begins by constructing a *HashTable h* and a

pair<HashTable::iterator, bool> *p* (lines 15-18). We don't need to use *Traceable* key values since all the *HashTable*'s elements (key-mapped pairs) are handled by the *ElementList*.

In lines 20-23, we insert (Shanghai, China) and use the returned *pair* to verify that the new element was inserted. We then try to insert an element with a duplicate key, (Shanghai, Virginia), and use the returned *pair* to verify that the new element wasn't inserted (lines 25-28).

In lines 30-41 we fill *h* to capacity then print the size, bucket count, and elements, generating the output

```
10 elements (10 buckets):
```

```
(Jakarta,Indonesia) (Sao Paulo,Brazil) (Moscow,Russia) (Seoul,South Korea)  
(Tokyo,Japan) (Istanbul,Turkey) (Delhi,India) (Lagos,Nigeria)  
(Karachi,Pakistan) (Shanghai,China)
```

We then insert 1 more element (line 44), which triggers a rehash, increasing the bucket count to 18. Once again we print the size, bucket count, and elements (line 46), generating the output

```
11 elements (18 buckets):
```

```
(Kinshasa,Congo) (Jakarta,Indonesia) (Sao Paulo,Brazil) (Moscow,Russia)  
(Seoul,South Korea) (Tokyo,Japan) (Istanbul,Turkey) (Delhi,India)  
(Lagos,Nigeria) (Karachi,Pakistan) (Shanghai,China)
```

12.3: Implementing the Local Iterators

Source files and folders

- *HashTable/2*
- *HashTable/common/LocalIter.h*
- *HashTable/common/memberFunctions_2.h*
- *printBuckets*

Chapter outline

- *Implementing local_iterator, const_local_iterator, and find*
- *Printing the contents of each bucket*

A *local iterator* traverses the elements in a single bucket. The *LocalIter* class (*LocalIter.h*, lines 6-7) has two template parameters, *HashTable* (the type of host container) and *Element* (the type of referent element (key-mapped pair)).

Each *LocalIter* contains a *_bucketIter* (lines 16, 36), initialized via the private constructor (lines 34, 109-114). This constructor will be used by *HashTable*'s member functions, so *HashTable* needs friend privileges (line 10). The default constructor (line 19) doesn't perform any explicit initialization (lines 39-43).

The member function *base* (line 24) returns (a copy of) the *_bucketIter* (lines 54-59).

The comparison operators (lines 25-26) compare the left and right operands by simply comparing their *_bucketIter*s (lines 61-73).

The dereference operator (line 28) accesses the referent element (key-mapped pair) by dereferencing the *_bucketIter* twice: the first dereference accesses an iterator to the key-mapped pair (*ElementList::iterator*), and the second dereference accesses the key-mapped pair itself (lines 82-87). The member access operator (line 27) does the same, but returns a pointer to the key-mapped pair instead (lines 75-80).

The increment operators (lines 30-31) point to the next element by advancing the *_bucketIter* (lines 89-107). There are no decrement operators because the *_bucketIter* is a forward iterator (line 17).

Before discussing the template constructor (*LocalIter.h*, lines 21-22), let's take a look at the updated *HashTable* class (*HashTable.h*, lines 39-40):

Type	is an alias of
<code>HashTable::const_local_iterator</code>	<code>LocalIter<HashTable, const value_type></code>
<code>HashTable::local_iterator</code>	<code>LocalIter<HashTable, value_type></code>

The difference between `const_local_iterator` and `local_iterator` lies in their `LocalIter`'s `Element` type:

Type	<code>LocalIter</code> Element type
<code>HashTable::const_local_iterator</code>	<code>const pair<const Key, Mapped></code>
<code>HashTable::local_iterator</code>	<code>pair<const Key, Mapped></code>
<hr/>	
<code>Type</code>	<code>LocalIter::pointer type</code> <code>(LocalIter::operator-> return type)</code>
<code>HashTable::const_local_iterator</code>	<code>const pair<const Key, Mapped>*</code>
<code>HashTable::local_iterator</code>	<code>pair<const Key, Mapped>*</code>
<hr/>	
<code>Type</code>	<code>LocalIter::reference type</code> <code>(LocalIter::operator* return type)</code>
<code>HashTable::const_local_iterator</code>	<code>const pair<const Key, Mapped>&</code>
<code>HashTable::local_iterator</code>	<code>pair<const Key, Mapped>&</code>

A `const_local_iterator` therefore cannot modify the `Mapped` value (because the whole `pair` is `const`), while a `local_iterator` can (because only the `Key` is `const`).

`HashTable`'s member function `begin` (`HashTable.h`, lines 50, 60) returns a local iterator to the first element in the bucket at the given `index` (`memberFunctions_2.h`, lines 3-8, 24-29). The return value (local iterator) is created via `LocalIter`'s private constructor (`LocalIter.h`, line 34).

`end` (`HashTable.h`, lines 51, 61) returns a local iterator to the one-past-the-last element in the bucket at the given `index` (`memberFunctions_2.h`, lines 10-15, 31-36).

`LocalIter`'s template constructor (`LocalIter.h`, lines 21-22) creates a `LocalIter` of one `Element` type (`LocalIter<HashTable, Element>`) from a `LocalIter` of a different `Element` type (`LocalIter<HashTable, OtherElement>`) (lines 45-52). This lets us construct `const_local_iterators` from `local_iterators` and perform implicit conversions:

```
typedef HashTable<string, int> HashTable;
HashTable h;
// insert some pairs, assume there are no empty buckets...
HashTable::local_iterator i = h.begin(7);
```

```

HashTable::const_local_iterator ci(i);      // construct const_local_iterator
                                           // from local_iterator

ci = i;                                     // implicitly convert local_iterator
                                           // to const_local_iterator

```

HashTable's *find* method (*HashTable.h*, lines 54, 64) returns an iterator to the element with the given *key* value, or *end* if the *key* wasn't found. We begin by calculating the index of the key value and searching the corresponding bucket (*memberFunctions_2.h*, lines 42-43). If the key was found, we return an iterator to the corresponding *pair* by dereferencing the *BucketIter* (lines 45-46); otherwise, we return *end* (lines 47-48).

To test the local iterators and view the exact layout of a *HashTable*, we'll write a separate non-member function (*printBuckets.h*, lines 7-8)

```

template <class HashTable>
void printBuckets(const HashTable& h);

```

This function prints the *bucket_count* of the given table *h*, followed by the index number and contents of each non-empty bucket. In each iteration of the loop, we first check whether the current bucket is empty, and if it is, we begin the next iteration without printing anything (lines 19-20). If the current bucket isn't empty, we print the index number and contents, using local iterators (lines 22-23).

Our test program (*main.cpp*) begins by constructing a table *h* and filling it to capacity (lines 16-29). We then *printBuckets* (line 31), generating the output

Non-empty buckets (out of 10 total):

```

1: (Jakarta,Indonesia)
3: (Delhi,India)
4: (Karachi,Pakistan) (Shanghai,China)
5: (Lagos,Nigeria)
7: (Sao Paulo,Brazil) (Seoul,South Korea) (Tokyo,Japan)
9: (Moscow,Russia) (Istanbul,Turkey)

```

Inserting another element, (Kinshasa, Congo), automatically increases the bucket count to 18 and triggers a rehash (line 33). Calling *printBuckets* again (line 35) generates the output

Non-empty buckets (out of 18 total):

```

1: (Seoul,South Korea)
2: (Shanghai,China)
7: (Lagos,Nigeria) (Istanbul,Turkey)
8: (Karachi,Pakistan)
9: (Moscow,Russia) (Sao Paulo,Brazil) (Jakarta,Indonesia)
11: (Tokyo,Japan)
13: (Kinshasa,Congo) (Delhi,India)

```

We then search for the key value "Tokyo" and confirm that it was found by printing the whole *pair* (lines 37-40), generating the output

Found (Tokyo, Japan)

When searching for the nonexistent key value “Beijing” (line 42), *find* returns *end*, so lines 44-45,

```
if (ci == h.end())
    cout << "\nBeijing not found\n\n";
```

generate the output

```
Beijing not found
```

Lastly, we *find* the element (Shanghai, China) and change its mapped value to “Virginia” (lines 47-50). Calling *printContainer* (line 52) shows the new value, generating the output

```
(Kinshasa,Congo) (Jakarta,Indonesia) (Sao Paulo,Brazil) (Moscow,Russia)
(Seoul,South Korea) (Tokyo,Japan) (Istanbul,Turkey) (Delhi,India)
(Lagos,Nigeria) (Karachi,Pakistan) (Shanghai,Virginia)
```

12.4: Erasing Elements

Source files and folders

- *HashTable/3*
- *HashTable/common/memberFunctions_3.h*

Chapter outline

- *Implementing HashTable's erase and clear methods*

To implement *HashTable*'s *clear* method (*HashTable.h*, line 67), we simply *clear* each *Bucket* along with the *ElementList* (*memberFunctions_3.h*, lines 18-21).

HashTable's *erase* method (*HashTable.h*, line 66) removes the given *element* and returns an iterator to the next element in the table. The procedure isn't particularly difficult: we calculate the index of the element's key value, remove the iterator from the corresponding *Bucket*, then remove the element from the *ElementList*:

```

    _elements
    ElementList
    (list<pair<const string, string>>)

    _____

    (China, Beijing)
    (Barbados, Bridgetown)
    (Japan, Tokyo)
    (Bulgaria, Sofia)
    (Finland, Helsinki)

    _____

    _buckets
    BucketArray
    (vector<Bucket*>)           Bucket
                                ForwardListSize<ElementList::iterator>

    _____

    0          [iter(Finland, Helsinki)]
    1          [iter(Barbados, Bridgetown)]
    2          []
    3          [iter(Bulgaria, Sofia)]
    4          [iter(China, Beijing), iter(Japan, Tokyo)]

```

```
Hash(key) = (ASCII code of first letter + ASCII code of last letter) %  
bucket count
```

To *erase* (Japan, Tokyo) :

- $\text{Hash("Japan")} = (74 + 110) \% 5 = 4$
- Remove *iter(Japan, Tokyo)* from Bucket 4
- Remove (Japan, Tokyo) from *ElementList*

Unfortunately, however, we can't use `_findIter` to search the bucket for "Japan"; we actually need to find the element *before* "Japan" so that we can `erase_after` it. To perform this search, let's write a separate function (`HashTable.h`, line 77),

```
BucketIter _findIterBefore(Bucket* bucket, const key_type& key);
```

This function searches the given `bucket` for the element (`ElementList::iterator`) before the element with the desired `key` value. The implementation is nearly identical to `_findIter`, but we use two iterators, one to the *current* element and one to the element *before* the current element (`memberFunctions_3.h`, lines 29-30).

In each iteration of traversing the bucket, we check if the *current* key matches the desired key, and if so, return the element *before* the current one (lines 34-35, 41); otherwise, we advance both iterators for the next iteration (lines 37-38). If we reach the end without finding the desired key, the return value (*before*) will point to the last element in the bucket (the one before *end*) (line 41).

We can now easily implement `erase`. After calculating the *index* of the "trash" (unwanted) element's key value (line 7), we use `_findIterBefore` to locate the `ElementList::iterator beforeTrash` (line 8). We then `erase_after beforeTrash` (line 10), thereby removing the iterator to the trash element from the bucket. Lastly, we use the supplied iterator (`element`) to remove the trash element from the `ElementList` (line 12), and return an iterator to the next element.

Our test program (`main.cpp`) begins by constructing a table `h` containing 5 elements (lines 17-29):

```
Non-empty buckets (out of 10 total):
```

```
1: (Jakarta,Indonesia)
3: (Delhi,India)
4: (Shanghai,China) (Karachi,Pakistan)
5: (Lagos,Nigeria)
```

```
(Jakarta,Indonesia) (Delhi,India) (Shanghai,China) (Lagos,Nigeria)
(Karachi,Pakistan)
```

We then find and erase (Shanghai, China), print the next element, and print the table (lines 31-43), generating the output

```
Erasing (Shanghai,China)...
```

```
Next element = (Lagos,Nigeria)
```

```
Non-empty buckets (out of 10 total):
```

```
1: (Jakarta,Indonesia)
3: (Delhi,India)
4: (Karachi,Pakistan)
5: (Lagos,Nigeria)
```

```
(Jakarta,Indonesia) (Delhi,India) (Lagos,Nigeria) (Karachi,Pakistan)
```

Lastly, we clear and print the table (lines 45-49), generating the output

```
Clearing table...
```

```
Non-empty buckets (out of 10 total) :
```


12.5: Implementing Copy and Assignment

Source files and folders

- *HashTable/4*
- *HashTable/common/memberFunctions_4.h*
- *Traceable/2*

Chapter outline

- *Implementing HashTable's copy constructor, assignment operator, and reserve method*
- *Creating a specialization of FnvHash for Traceable keys*

To implement *HashTable*'s copy constructor (*HashTable.h*, line 43), we initialize each data member in the new table as a copy of its counterpart in the *source* table, except for the *BucketArray* (*memberFunctions_4.h*, lines 7-10, 12).

The new table's *BucketArray* is initialized to the same size as the *source* array, but contains null pointers (line 11); we can't simply copy the *source* buckets because the iterators in those buckets refer to the *source* table's *ElementList*.

After creating a set of empty buckets for the new table (lines 14-15), we need to populate them with iterators to the new elements. To do this we traverse the new *ElementList* (line 17), and in each iteration, calculate the *index* of the current key and place an iterator to the current element in the corresponding bucket (lines 19-20):

```

source._elements
    ElementList
    (list<pair<const string, string>>)



---


(China, Beijing)
(Barbados, Bridgetown)
(Japan, Tokyo)
(Bulgaria, Sofia)
(Finland, Helsinki)

source._buckets
    BucketArray
    (vector<Bucket*>)           Bucket
                                ForwardListSize<ElementList::iterator>



---


0          [iter(Finland, Helsinki)]
1          [iter(Barbados, Bridgetown)]
2          []
3          [iter(Bulgaria, Sofia)]
4          [iter(China, Beijing), iter(Japan, Tokyo)]

```

```

// Copy all data members from the source table, but populate the new
// BucketArray with an equal number of empty buckets

    _elements
    ElementList
(list<pair<const string, string>>)



---


(China, Beijing)
(Barbados, Bridgetown)
(Japan, Tokyo)
(Bulgaria, Sofia)
(Finland, Helsinki)

    _buckets
    BucketArray
(vector<Bucket*>)      Bucket
                           ForwardListSize<ElementList::iterator>



---


0          []
1          []
2          []
3          []
4          []

// Traverse the new ElementList (_elements), populating the new buckets
// with iterators to the new elements

    _index("China")     = 4
    _index("Barbados")  = 1
    _index("Japan")     = 4
    _index("Bulgaria")  = 3
    _index("Finland")   = 0

    _buckets
    BucketArray
(vector<Bucket*>)      Bucket
                           ForwardListSize<ElementList::iterator>



---


0          [iter(Finland, Helsinki)]
1          [iter(Barbados, Bridgetown)]
2          []
3          [iter(Bulgaria, Sofia)]
4          [iter(Japan, Tokyo), iter(China, Beijing)]

```

Before implementing *HashTable*'s assignment operator, let's write the *reserve* method (*HashTable.h*, line 67),

```
void reserve(size_type newCapacity);
```

This function, like *vector::reserve*, increases the capacity of the table to the *newCapacity*. If the *newCapacity* exceeds the current capacity (*memberFunctions_4.h*, line 27), we determine the *newBucketCount* required to achieve the *newCapacity* (lines 29-30), using the formula from Chapter

12.1. We then `_rehash` the table using the `newBucketCount` (line 32).

We can now use `reserve` to optimize the assignment operator (*HashTable.h*, line 66). After clearing the left operand (*memberFunctions_4.h*, line 40), we `reserve` enough capacity to avoid unnecessary rehashes when copying the elements from the right operand. We then traverse the right-hand *ElementList* (line 43), and in each iteration, calculate the *index* of the current *element*, copy the *element* to the left-hand *ElementList*, and place an iterator to the new element in the corresponding bucket (lines 45-48).

To test these new methods, we'll construct a table of *Traceable* keys (*HashTable<Traceable<string>, string>*). The default version of *FnvHash*, however, won't work with *Traceable<string>* keys because it will hash the entire *Traceable object* as opposed to the string *value*. Given

```
FnvHash<Traceable<string>> hash;
Traceable<string> x = "Mercury";
Traceable<string> y = "Mercury";
```

for example, the expressions

```
hash(x)
hash(y)
```

would return *different* values because the default version of *FnvHash* wouldn't hash the *values* of *x* and *y* ("Mercury"), but rather the *objects* *x* and *y*. Similarly, given

```
FnvHash<Traceable<int>> hash;
Traceable<int> a = 7;
Traceable<int> b = 7;
```

the expressions

```
hash(a)
hash(b)
```

would return different values because the default version of *FnvHash* wouldn't hash the *values* of *a* and *b* (7), but rather the *objects* *a* and *b*.

To get the behavior we need, we'll create a specialization of *FnvHash* for *Traceable<string>*, similar to how we specialized *FnvHash* for *std::string* in Chapter 12.1. The only difference from the default version lies in how we call *fnvHash* (*Traceable.h*, lines 36-42): instead of passing the entire *Traceable<string>* object (*key*), we pass the characters only (*key.value().c_str()*).

The other specialization of *FnvHash* applies to all other *Traceable* types, such as *Traceable<int>*: instead of passing the entire *Traceable* object (*key*), we pass the embedded value only (*key.value()*) (lines 27-34).

Our test program (*main.cpp*) begins by constructing a table *h* and inserting 11 elements, distributed among 18 buckets (the 11th element triggers a bucket count increase from 10 to 18) (lines 18-32). We then copy construct a table *t* from *h* and print *t* (lines 35-39), generating the output

```
c Kinshasa      // new keys, copied from h to t
c Jakarta
c Sao Paulo
c Moscow
c Seoul
c Tokyo
c Istanbul
c Delhi
c Lagos
c Karachi
c Shanghai
```

Non-empty buckets (out of 18 total):

```
1: (Seoul,South Korea)
2: (Shanghai,China)
7: (Lagos,Nigeria) (Istanbul,Turkey)
8: (Karachi,Pakistan)
9: (Moscow,Russia) (Sao Paulo,Brazil) (Jakarta,Indonesia)
11: (Tokyo,Japan)
13: (Delhi,India) (Kinshasa,Congo)
```

```
(Kinshasa,Congo) (Jakarta,Indonesia) (Sao Paulo,Brazil) (Moscow,Russia)
(Seoul,South Korea) (Tokyo,Japan) (Istanbul,Turkey) (Delhi,India)
(Lagos,Nigeria) (Karachi,Pakistan) (Shanghai,China)
```

To test the assignment operator, we construct an empty table *x*, assign *t* to *x*, and print *x* (lines 42-47), generating the output

```
c Kinshasa      // new keys, copied from t to x
c Jakarta
c Sao Paulo
c Moscow
c Seoul
c Tokyo
c Istanbul
c Delhi
c Lagos
c Karachi
c Shanghai
```

Non-empty buckets (out of 11 total):

```
0: (Kinshasa,Congo)
1: (Shanghai,China)
2: (Karachi,Pakistan)
3: (Delhi,India) (Sao Paulo,Brazil)
4: (Seoul,South Korea)
5: (Moscow,Russia)
```

```
6: (Tokyo,Japan) (Jakarta,Indonesia)
8: (Istanbul,Turkey)
9: (Lagos,Nigeria)
```

```
(Kinshasa,Congo) (Jakarta,Indonesia) (Sao Paulo,Brazil) (Moscow,Russia)
(Seoul,South Korea) (Tokyo,Japan) (Istanbul,Turkey) (Delhi,India)
(Lagos,Nigeria) (Karachi,Pakistan) (Shanghai,China)
```

Note the difference in bucket count between t (18) and x (11):

- t was copy constructed from h , so t 's *BucketArray* was initialized to the same size as h 's (18).
- x was default constructed with an initial capacity of 10, so when we assigned t to x , the assignment operator reserved a new capacity of $t.size()$ (11). Based on x 's load factor of 1.0, the bucket count only needed to be increased to 11.

At the end of *main* all 3 tables are destroyed, generating the output

```
~ Kinshasa      // x's destructor
~ Jakarta
~ Sao Paulo
~ Moscow
~ Seoul
~ Tokyo
~ Istanbul
~ Delhi
~ Lagos
~ Karachi
~ Shanghai
~ Kinshasa      // t's destructor
~ Jakarta
~ Sao Paulo
~ Moscow
~ Seoul
~ Tokyo
~ Istanbul
~ Delhi
~ Lagos
~ Karachi
~ Shanghai
~ Kinshasa      // h's destructor
~ Jakarta
~ Sao Paulo
~ Moscow
~ Seoul
~ Tokyo
~ Istanbul
~ Delhi
~ Lagos
~ Karachi
~ Shanghai
```

All Traceables destroyed

12.6: Adjusting the Max Load Factor

Source files and folders

- *HashTable/5*
- *HashTable/common/memberFunctions_5.h*

Chapter outline

- *Manually increasing the capacity via the bucket count*
- *Manually increasing the capacity or search speed via the max load factor*

In the previous chapter we implemented the *reserve* method, which lets the user request a specific capacity in terms of the total number of elements. To complete the *HashTable* class, we'll add two more functions that let the user manually increase the capacity and search speed.

Recall from Chapter 12.2 that according to the formula

```
capacity = bucket count x max load factor
```

we can increase the total capacity by increasing the bucket count, max load factor, or both. Our first new function (*HashTable.h*, line 68),

```
void rehash(size_type newBucketCount);
```

increases the bucket count to the *newBucketCount* by simply calling *_rehash* (*memberFunctions_5.h*, lines 23-28), thereby increasing the total capacity.

The second function (*HashTable.h*, line 67),

```
void max_load_factor(float newMaxLoadFactor);
```

sets the max load factor to the *newMaxLoadFactor*:

- If the *newMaxLoadFactor* is less than the current *_maxLoadFactor*, we're restricting the average bucket size, so the bucket count must increase to compensate. After determining the required *newBucketCount*, we *_rehash* the table accordingly and update the *_maxLoadFactor* (*memberFunctions_5.h*, lines 9-16). Because the *increase* in bucket count is proportional to the *decrease* in max load factor, the total capacity is unchanged. The higher bucket count, however, does reduce the average bucket size (load factor), thereby speeding up the average search time (at the cost of a higher bucket count / larger memory footprint).
- If the *newMaxLoadFactor* is greater than the current *_maxLoadFactor*, we're relaxing the average bucket size, so we don't need to increase the bucket count. We simply increase the *_maxLoadFactor*, thereby increasing the total capacity (lines 17-20).

The following table summarizes the effects of changing the bucket count and max load factor (*u/c* stands for “unchanged”):

	total capacity	average bucket size	average search time	total memory footprint
increase bucket count	higher	smaller	faster	larger
increase max load factor	higher	larger	slower	u/c
decrease max load factor	u/c	smaller	faster	larger

Our test program (*main.cpp*) begins by constructing a table *h* and filling it to capacity (lines 17-33):

Non-empty buckets (out of 10 total) :

```

1: (Jakarta,Indonesia)
3: (Delhi,India)
4: (Karachi,Pakistan) (Shanghai,China)
5: (Lagos,Nigeria)
7: (Sao Paulo,Brazil) (Seoul,South Korea) (Tokyo,Japan)
9: (Moscow,Russia) (Istanbul,Turkey)

// capacity = max load factor x bucket count
//           = 1.0 elements / bucket x 10 buckets
//           = 10 elements

```

We then decrease the max load factor to 0.5 (lines 35-36), which increases the bucket count to 20. The total capacity is unchanged, but the higher bucket count reduces the average bucket size, speeding up the average search time:

Non-empty buckets (out of 20 total) :

```

1: (Jakarta,Indonesia)
4: (Shanghai,China)
5: (Lagos,Nigeria)
7: (Seoul,South Korea)
9: (Moscow,Russia)
13: (Delhi,India)
14: (Karachi,Pakistan)
17: (Tokyo,Japan) (Sao Paulo,Brazil)
19: (Istanbul,Turkey)

// capacity = max load factor x bucket count
//           = 0.5 elements / bucket x 20 buckets
//           = 10 elements

```

Lastly, we increase the bucket count to 100 (lines 38-39), further reducing the average bucket size. The max load factor is unchanged, so the total capacity increases:

Non-empty buckets (out of 100 total):

```
17: (Sao Paulo, Brazil)
27: (Seoul, South Korea)
41: (Jakarta, Indonesia)
57: (Tokyo, Japan)
73: (Delhi, India)
74: (Karachi, Pakistan)
84: (Shanghai, China)
85: (Lagos, Nigeria)
89: (Moscow, Russia)
99: (Istanbul, Turkey)
```

```
// capacity = max load factor x bucket count
//           = 0.5 elements / bucket x 100 buckets
//           = 50 elements
```


Index

&, 400
|, 400
^, 400
~, 401
<<, See left shift
>>, See right shift
/0, 377
= 0, 337
2-3-4 tree, 205

A

abstract class, 339
`_advance`, 88
`advance`, 88
<algorithm>, 38, 48
allocator, 102
and (bitwise), 400
arithmetic right shift, 405
ASCII, 375
auto, 67, 81

B

B-tree
erasure, 163, 166
insertion, 120
leaf element, 163
left child, 100
left rotation, 171
merge, 174, 178, 182
order, 99
overflow, 109, 120
overview, 99
promote-and-split, 120
right child, 100
right rotation, 168
base, 389
base-10 system, 371
base-2 system, 372

base class, 86, 338, 352
Bayer, Rudolf, 99
biased exponent, 392, 395
bidirectional_iterator_tag, 86
big-endian, 411
binary search, 89, 97
binary system, 372
binary16 (32 / 64 / 128), 392, 412
binarySearch, 98
bit, 374
bitwise logical operators, 400, 402
bitwise shift operators, 400, 405, 407
bound, 95, 105
BTree
 back, 119
 _balanceOnErase, 166, 187
 _balanceOnInsert, 135, 138
 begin, 160
 clear, 201
 const_iterator, 160
 const_reverse_iterator, 160
 copy constructor, 201
 _copyNode, 199
 _createNode, 119
 data members, 119
 default constructor, 119
 _destroyNode, 119
 _destroyTree, 119
 destructor, 120
 empty, 119
 end, 160
 erase, 163
 _extractMiddleElementAndSplitNode, 136
 find, 160
 _findInsertionPointForNewKey, 133
 front, 119
 _getLeafElementForDeletion, 164
 head, 119
 insert, 132, 160
 iterator, 160
 key_comp, 119
 LeafElement, 164
 Location, 151
 _location, 160
 _mergeRootWithItsTwoChildren, 175
 _mergeWithLeftSibling, 180
 _mergeWithRightSibling, 185
 NewKeySearch, 133
 Node, 119
 operator=, 201

BTree (continued)

rbegin, 160
rend, 160
reverse_iterator, 160
root, 119
_rotateParentElementLeft, 173
_rotateParentElementRight, 169
size, 119
SplitNode, 136
_swapElements, 165
tail, 119
value_type, 101
bTreeInOrderPredecessor, 154
bTreeInOrderSuccessor, 151
BTreeIter, 159
BTreeNode
backChild, 110
backElement, 111
backElementIndex, 111
child, 111
default constructor, 109
eraseChild, 113
eraseElement, 113
frontChild, 110
frontElement, 111
hasElement, 109
hasLeftChild, 110
hasOverflow, 109
hasRightChild, 110
insertChild, 113
insertElement, 112
isAtLeastHalfFull, 109
isLeaf, 110
isLeftChild, 110
isMoreThanHalfFull, 109
isRightChild, 110
isRoot, 110
key, 111
kmPair, 112
leftChild, 111
leftSibling, 112
lowerBound, 111
overwriteElement, 112
parent, 111
popBackChild, 113
popBackElement, 113
popFrontChild, 113
popFrontElement, 113
pushBackChild, 113
pushBackElement, 113

BTreeNode (continued)

pushFrontChild, 113
pushFrontElement, 113
rightChild, 111
rightSibling, 112
setParent, 113
totalChildren, 109
totalElements, 109
byte, 375

C

C-string, 377
capacitor, 375
cbegin, 68
ceil, 6
cend, 68
char, 377
char[], 377
CHAR_BIT, 397
character encoding
 ASCII, 375
 non-ASCII, 387
Circle, 338
<climits>, 397
<cmath>, 6, 301
coefficient, *See* mantissa
compound assignment operators (bitwise), 402
concrete class, 339
const, 397
*const unsigned char**, 409
control code, 376
crbegin, 67
createShape, 340
crend, 67
<cstdlib>, 289, 290
<cstring>, 378
<ctime>, 289

D

decimal system, 371
derived class, 86, 337, 352
dispersion, 417
_distance, 87
distance, 87

downcast, 352
dynamic dispatch, 340

E

ForwardListElementNode, 352
ForwardListIter, 359
ForwardListNode, 352
ForwardListSize, 369
Fowler, Glenn, 417
<functional>, 4

endianness, 411
eraseAndPrint, 317
error, 391
exponent, 389, 395

F

findExtreme, 46
floor, 6
FNV hash function, 417
FnvHash, 420, 443
fnvHash, 420
forward list, 351
forward_iterator_tag, 86
forward_list, 367
<forward_list>, 367
ForwardList
 before_begin, 359
 begin, 360
 clear, 357
 const_iterator, 359
 copy constructor, 366
 _copyElementsFrom, 365
 _createElementNode, 353
 data members, 352
 default constructor, 352
 _destroyAllElementNodes, 354
 _destroyElementNode, 353
 destructor, 354
 downcast, 352
 ElementNode, 352
 empty, 352
 end, 360
 erase_after, 361
 front, 352
 insert_after, 361
 iterator, 359
 Node, 352
 operator=, 366
 pop_front, 357
 push_front, 354

G

getKeyAndEraseElement, 188
getKeyAndFindElement, 161
getKeyAndInsertElement, 142, 161
getSubtype, 340
greater, 4
Guibas, Leonidas, 205

H

hash function, 415, 417
hash table
 bucket, 417
 bucket count, 424, 447
 capacity, 425, 447
 collision, 416
 load factor, 424
 local iterator, 433
 max load factor, 425, 447
 rehash, 425
 traversal, 424
HashTable
 begin, 426, 434
 Bucket, 423
 bucket, 426
 bucket_count, 426, 445
 bucket_size, 426
 BucketArray, 423
 BucketIter, 426
 capacity, 426
 clear, 437
 const_local_iterator, 433
 copy constructor, 441
 _createBucket, 425
 data members, 423
 default constructor, 425
 _destroyBucket, 425
 destructor, 426

<i>HashTable</i> (<i>continued</i>)	
<i>ElementList</i> , 423	heap transform (namespace <i>ht</i>) (<i>continued</i>)
<i>empty</i> , 426	<i>leftChild</i> , 28
<i>end</i> , 426, 434	<i>makeHeap</i> , 27
<i>erase</i> , 437	<i>moveElementDownTree</i> , 29
<i>find</i> , 435	<i>parent</i> , 28
<i>_findIter</i> , 426	<i>popHeap</i> , 37
<i>_findIterBefore</i> , 438	<i>pushHeap</i> , 33
<i>hash_function</i> , 426	<i>rightChild</i> , 28
<i>_index</i> , 426	<i>heapSort</i> , 41-43
<i>insert</i> , 426, 428, 431	
<i>iterator</i> , 423	
<i>key_eq</i> , 426	
<i>load_factor</i> , 426	
<i>local_iterator</i> , 433	IEEE 754, 389, 412
<i>LocallIter</i> , 433	inheritance, 85, 343
<i>max_load_factor</i> , 426, 447	<i>input_iterator_tag</i> , 86
<i>operator=</i> , 443	<i>instantiate</i> , 339
<i>_rehash</i> , 428	integral type, 398, 406, 407
<i>rehash</i> , 447	invert (bits), 402
<i>reserve</i> , 442	<i>iterator_category</i> , 87
<i>size</i> , 426	
standard member types, 423	
template parameters, 423	
<i>heap</i>	
bottom element, 14	
max vs. min, 4	
root, 4	
top element, 1	
<i>Heap</i>	
assignment operator, 4	<i>KmPair</i> , 102
constructor, 4	<i>kmPairLowerBound</i> , 107
<i>container_type</i> , 4	<i>KmPairLowerBoundPred</i> , 106
destructor, 4	<i>kmPairUpperBound</i> , 107
<i>empty</i> , 4	<i>KmPairUpperBoundPred</i> , 107
<i>_getChildForComparison</i> , 17	
<i>_hasTwoChildren</i> , 17	
<i>_isLeaf</i> , 17	
<i>_leftChild</i> , 16	
<i>Node</i> , 4	late binding, 340
<i>pop</i> , 14, 17	least significant byte, 411
<i>Predicate</i> , 4	left shift, 400, 403, 405
<i>push</i> , 5	<i>less</i> , 4
<i>_rightChild</i> , 16	<i><limits></i> , 397
<i>size</i> , 4	little-endian, 411
<i>top</i> , 4	load factor, 424
<i>heap transform (namespace <i>ht</i>)</i>	<i>LocallIter</i> , 433
<i>getChildForComparison</i> , 29	logical left shift, <i>See</i> left shift
<i>hasTwoChildren</i> , 28	logical right shift, <i>See</i> right shift
<i>isLeaf</i> , 28	lower bound, 89

I**J****K****L**

lowerBound, 95
LSB, 411

P

macro, 397
magnitude, 381
make_heap, 38
mantissa, 389, 395
max_element, 48
McCreight, Edward M., 99
<memory>, 102, 349
mergeSort (namespace *msCommon*)
 implementation, 83
 merge, 79
 overview, 69
 split, 73
min_element, 48
minimal perfect hash function, 417
most significant byte, 411
MSB, 411

M

perfect hash function, 417
pit, 375
pointer to *bool* conversion, 346
polymorphism, 343
pop_heap, 38
precision, 391
preprocessor, 397
printAvailableNodes, 293
printBinaryRep, 409, 410
PrintBtNode, 211
PrintBTreeNode, 117
printBuckets, 435
printByte, 407
printByteSequence, 409
PrintRedBlackNode, 211
printSkipList, 293
printTypeTraits, 397
printUseCountAndArea, 348
priority queue, *See* heap
priority_queue, 38
probabilistic, 275
public, 86
Pugh, William, 273
punch card, 375
pure virtual function, 337
push_heap, 38

N

NaN, 394
Noll, Landon Curt, 417
not (bitwise), 401
null character, 377
null-terminated, 377
numeric_limits, 397

Q

<queue>, 38

O

R

octet, 375
or (bitwise), 400
output_iterator_tag, 86
overflow, 403
overridden, 337
radix point, 371
rand, 290
random_access_iterator_tag, 86
range-based *for* loop, 80, 341, 425
raw pointer, 345
red-black tree
 2-node, 206
 3-node
 black / red portion, 245

red-black tree (*continued*)3-node (*continued*)

- left-leaning, 206
- right-leaning, 207

4-node, 207

B-tree left rotation, 252

B-tree right rotation, 257

demote parent element, 250

erasure

- from 2-node, 249
- from 3-node or 4-node, 243

grandparent, 214*hasGrandparent*, 213*hasUncle*, 213

insertion, 215

left-left violation, 219

left-right violation, 220

merge, 263

overflow, 217

promote-and-split, 217

properties, 207

right-left violation, 222

right-right violation, 221

sibling, 214*uncle*, 214

underflow, 250, 252, 263

RedBlackTree

- _balanceOn2NodeErase*, 243, 249
- _balanceOn3Or4NodeErase*, 243
- _balanceOnErase*, 243
- _balanceOnInsert*, 215, 224
- const_iterator*, 211
- const_reverse_iterator*, 211
- _convertToLeftLeaning3Node*, 247
- _convertToRightLeaning3Node*, 248
- _copyNode*, 211
- data members, 211
- erase*, 243
 - _fixLeftLeft*, 220
 - _fixLeftRight*, 221
 - _fixRightLeft*, 224
 - _fixRightRight*, 222
- insert*, 215
 - _is2Node*, 245
 - _is4Node*, 246
 - _is4NodeOrLeftLeaning3Node*, 246
 - _is4NodeOrRightLeaning3Node*, 246
 - _isLeftLeaning3Node*, 246
 - _isRightLeaning3Node*, 246
- iterator*, 211

RedBlackTree (*continued*)

member functions reused from *BinaryTree*, 211

Node, 211*_overwriteElement*, 211*_performBTTreeLeftRotation*, 253*_performBTTreeRightRotation*, 257*reverse_iterator*, 211*RedBlackTreeNode*, 210*reinterpret_cast*, 410, 420*relatives.h* (namespace *ds2::bt*), 213

right shift, 400, 404, 405

S

scientific notation, 389

Sedgewick, Robert, 205

seed, 289

selectionSort, 45, 51

Shape, 337

shared pointer, 345

shared_ptr, 349

SharedPtr, 345

Shell, Donald, 55

shellSort (namespace *shell*)

- generateGapSizes*, 64

- implementation, 67

- moveElement*, 57

- overview, 55

- sortAllSubsequences*, 63

- sortSubsequence*, 59

shift, *See* left shift, right shift

sign bit, 381

sign-magnitude, 381

signed integral type, 407

significand, *See* mantissa

sizeof, 398

skip list

- available nodes, 276

- bottom level, 273

- capacity, 275, 299

- erasure, 311

- head, 274

- height, 273

- insertion, 280

- level, 273

- overview, 273

- search, 274

- tail, 274

skip list (*continued*)

top level, 275

SkipList

_attachNewNode, 290

availableNodes, 278

back, 278

begin, 307

capacity, 278

const_iterator, 307

const_reverse_iterator, 307

copy constructor, 323

_createNode, 279

data members, 278

default constructor, 279

_destroyAllNodes, 279

_destroyNode, 279

destructor, 279

_detachNode, 314

empty, 278

end, 308

erase, 311, 317

find, 308

_findInsertionPointForNewKey, 280, 284

_findRightLinksOnErase, 311

front, 278

_getLevelForNewNode, 289

head, 278

increaseTopLevel, 299

_initAvailableNodeCount, 279

insert, 279, 300, 308

iterator, 307

key_comp, 278

LinkVector, 280

NewKeySearch, 280

NodeVector, 280

operator=, 330

rbegin, 308

rend, 308

reserve, 300

reverse_iterator, 307

size, 278

topLevel, 278

SkipListIter, 307

SkipListNode, 277

smart pointer, 345

Square, 339

rand, 289

stored mantissa, 392, 395

string, 379

subclass, *See* derived class

subtype, *See* derived class

swap, 6

T

template constructor, 433

time, 289

Tokuda's sequence, 63

Traceable, 104, 443

traverseBTreeInOrder, 116

traverseInOrder, 115

Triangle, 339

true exponent, 392, 395

true mantissa, 392, 395

two's complement, 381

type_info, 398

typeid, 398

U

Unicode, 381

Unix time, 289

unsigned char, 406

unsigned integral type, 406

upcast, 352

upper bound, 89

upperBound, 96

UpperBoundPredicate, 96

<utility>, 6

V

virtual, 337, 339

virtual destructor, 341

virtual function, 338, 340

Vo, Phong, 417

von Neumann, John, 69

W

Williams, John, 41

458

word, 375

X

xor (bitwise), 400

Y

Z