

COIS 3020 ASSIGNMENT 3

Names: Khushi Chauhan(0722283); Farhana Zahan(0691212), Anubhav Mehandru(0752670)

References:

- COIS 2020H - Biran Shrivastava's Assignments and notes

Part A:

Testing:

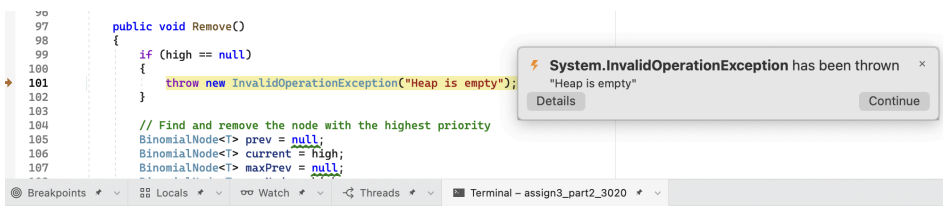
Test Case 1	2D
Input	<ul style="list-style-type: none">• Current Point: {35, 50}• New Point: {15, 60}
Expected output	<ul style="list-style-type: none">• Binary Index: 10• Decimal Index: 2
Output	<pre>Testing 2D: Index (Binary): 10 Index (Decimal): 2</pre>

Test Case 2	3D
Input	<ul style="list-style-type: none">• Current Point: {35, 50, 70}• New Point: {15, 60, 80}
Expected output	<ul style="list-style-type: none">• Binary Index: 100• Decimal Index: 4
Output	<pre>Testing 3D: Index (Binary): 100 Index (Decimal): 4</pre>

Test Case 3	4D
Input	<ul style="list-style-type: none">• Current Point: {35, 50, 70, 90}• New Point: {15, 60, 80, 100}
Expected output	<ul style="list-style-type: none">• Binary Index: 1000• Decimal Index: 8
Output	<pre>Testing 4D: Index (Binary): 1000 Index (Decimal): 8</pre>

Part B Testing:

Test Case 1	Heap before adding and item
Input	nothing
Expected output	Lazy binomial heap is empty
Output	<pre> Heap before adding an item: Lazy binomial heap is empty. </pre>

Test Case 2	Try remove when the heap is empty
Input	BH.Remove();
Expected output	Throw Invalid operation exception
Output	 <pre> 97 public void Remove() 98 { 99 if (high == null) 100 { 101 throw new InvalidOperationException("Heap is empty"); 102 } 103 104 // Find and remove the node with the highest priority 105 BinomialNode<T> prev = null; 106 BinomialNode<T> current = high; 107 BinomialNode<T> maxPrev = null; </pre> <p>Heap before adding an item: Lazy binomial heap is empty.</p> <p>Removing when the heap is empty:</p>

Test Case 3	Add items and print the item with highest priority
Input	<pre> BH.Add(new PriorityClass(20, 'a')); BH.Add(new PriorityClass(30, 'b')); BH.Add(new PriorityClass(25, 'c')); BH.Add(new PriorityClass(40, 'd')); BH.Add(new PriorityClass(35, 'e')); </pre>
Expected output	Item with highest priority: d with priority 40
Output	<pre> Heap before adding an item: Lazy binomial heap is empty. Items in the lazy binomial heap: Item with highest priority: d with priority 40 </pre>

Test Case 4	Printing of heap after adding elements
Input	<pre>BH.Add(new PriorityClass(20, 'a')); BH.Add(new PriorityClass(30, 'b')); BH.Add(new PriorityClass(25, 'c')); BH.Add(new PriorityClass(40, 'd')); BH.Add(new PriorityClass(35, 'e'));</pre>
Expected output	<pre>Degree 0: Item: a with priority 20 Item: b with priority 30 Item: c with priority 25 Item: d with priority 40 Item: e with priority 35</pre>
Output	<pre>Heap before adding an item: Lazy binomial heap is empty. Items in the lazy binomial heap: Item with highest priority: d with priority 40 Degree 0 trees: Item: a with priority 20 Item: b with priority 30 Item: c with priority 25 Item: d with priority 40 Item: e with priority 35</pre>

Test Case 5	Highest priority item after a remove and coalesce
Input	<pre>BH.Remove();</pre>
Expected output	Item with highest priority: e with priority 35
Output	<pre>Removing the highest priority item: Heap after coalesce: Items in the lazy binomial heap: Item with highest priority: e with priority 35</pre>

Test Case 6	Print the heap after remove and coalesce
Input	BH.Remove();
Expected output	Degree 4: Item: a with priority 20 Item: b with priority 30 Item: c with priority 25 Item: e with priority 35
Comment	Even though the coalesce worked well, the item that is removed is still there in the heap
Output	<p>Heap before adding an item: Lazy binomial heap is empty.</p> <p>Items in the lazy binomial heap: Item with highest priority: d with priority 40 Degree 0 trees: Item: a with priority 20</p> <p>Item: b with priority 30 Item: c with priority 25 Item: d with priority 40 Item: e with priority 35</p> <p>Removing the highest priority item:</p> <p>Heap after coalesce: Items in the lazy binomial heap: Item with highest priority: e with priority 35 Degree 4 trees: Item: a with priority 20</p> <p>Item: b with priority 30 Item: c with priority 25 Item: d with priority 40 Item: e with priority 35</p>

Test Case 7	Add more items and see if the highest is updated and print the new heap
Input	<pre>BH.Add(new PriorityClass(33, 'f')); BH.Add(new PriorityClass(47, 'g')); BH.Add(new PriorityClass(45, 'h'));</pre>
Expected output	<p>Items in the lazy binomial heap: Item with highest priority: g with priority 47 Degree 0 trees: Item: f with priority 33 Item: g with priority 47 Item: h with priority 45</p> <p>Degree 4 trees: Item: a with priority 20 Item: b with priority 30 Item: c with priority 25 Item: e with priority 35</p>
Comment	This case works well, except displaying of item d due to some error in Test 6
Output	<pre>Heap after adding more items Items in the lazy binomial heap: Item with highest priority: g with priority 47 Degree 0 trees: Item: f with priority 33 Item: g with priority 47 Item: h with priority 45 Degree 4 trees: Item: a with priority 20 Item: b with priority 30 Item: c with priority 25 Item: d with priority 40 Item: e with priority 35</pre>

Test Case 8	Duplicate items
Input	BH.Add(new PriorityClass(33, 'f'));
Expected output	<p>Heap after adding duplicate items</p> <p>Items in the lazy binomial heap:</p> <p>Item with highest priority: g with priority 47</p> <p>Degree 0 trees:</p> <p>Item: f with priority 33</p> <p>Item: g with priority 47</p> <p>Item: h with priority 45</p> <p>Item: f with priority 33</p> <p>Degree 4 trees:</p> <p>Item: a with priority 20</p> <p>Item: b with priority 30</p> <p>Item: c with priority 25</p> <p>Item: e with priority 35</p>
Comment	This case works well, except displaying of item d due to some error in Test 6
Output	<pre> Heap after adding duplicate items Items in the lazy binomial heap: Item with highest priority: g with priority 47 Degree 0 trees: Item: f with priority 33 Item: g with priority 47 Item: h with priority 45 Item: f with priority 33 Degree 4 trees: Item: a with priority 20 Item: b with priority 30 Item: c with priority 25 Item: d with priority 40 Item: e with priority 35 </pre>

Test Case 9	Removing again to see another coalesce
Input	BH.Remove();
Expected output	<p>Removing the highest priority item:</p> <p>Heap after coalesce Items in the lazy binomial heap: Item with highest priority: f with priority 33 Degree 3 trees: Item: f with priority 33 Item: h with priority 45 Item: f with priority 33</p> <p>Degree 4 trees: Item: a with priority 20 Item: b with priority 30 Item: c with priority 25 Item: e with priority 35</p>
Comment	This case works well, except displaying of item d and item g due to some error in Test 6
Output	<p>Removing the highest priority item:</p> <p>Heap after coalesce Items in the lazy binomial heap: Item with highest priority: f with priority 33 Degree 3 trees: Item: f with priority 33</p> <p>Item: g with priority 47 Item: h with priority 45 Item: f with priority 33</p> <p>Degree 4 trees: Item: a with priority 20</p> <p>Item: b with priority 30 Item: c with priority 25 Item: d with priority 40 Item: e with priority 35</p>

Part C:

- **Demonstrate the equivalence between 2-3-4- trees and red-black trees in a separate document (5 marks)**

The equivalence between 2-3-4 trees and Red-Black trees can be understood as follows:

In a 2-3-4 tree, which is a type of B-tree, each node can contain 1 to 3 keys and can have 2 to 4 children. A 2-node contains one key and two children, a 3-node contains two keys and three children, and a 4-node contains three keys and four children.

On the other hand, a Red-Black tree is a balanced binary search tree with an extra bit of storage per node: its colour, which can be either red or black. The balance is maintained by painting each node with one of these two colours in a way that satisfies certain properties which ensures that the tree remains approximately balanced during insertions and deletions.

The equivalence between the two trees can be demonstrated through the following transformation rules:

1. A 2-node is equivalent to a black node in a Red-Black tree.
2. A 3-node is equivalent to a black node with a red child in a Red-Black tree. The black node contains the larger key and the red node contains the smaller key.
3. A 4-node is equivalent to a black node with two red children in a Red-Black tree. The black node contains the middle key, the left red child contains the smallest key, and the right red child contains the largest key.

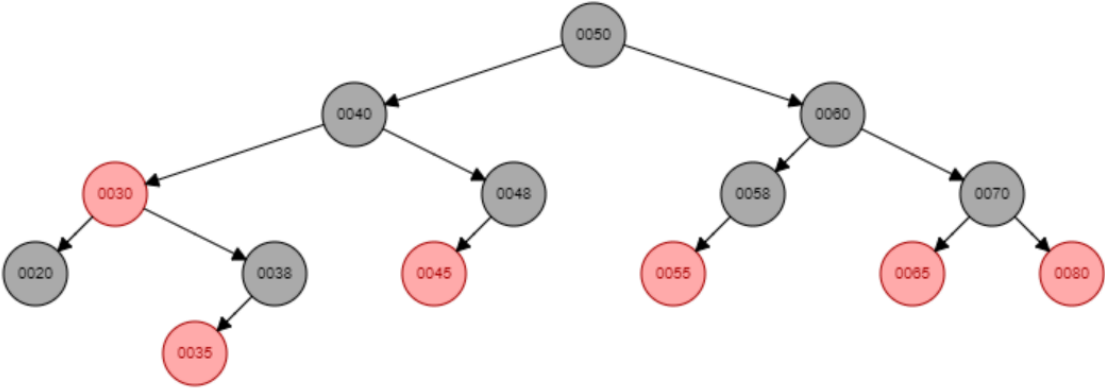
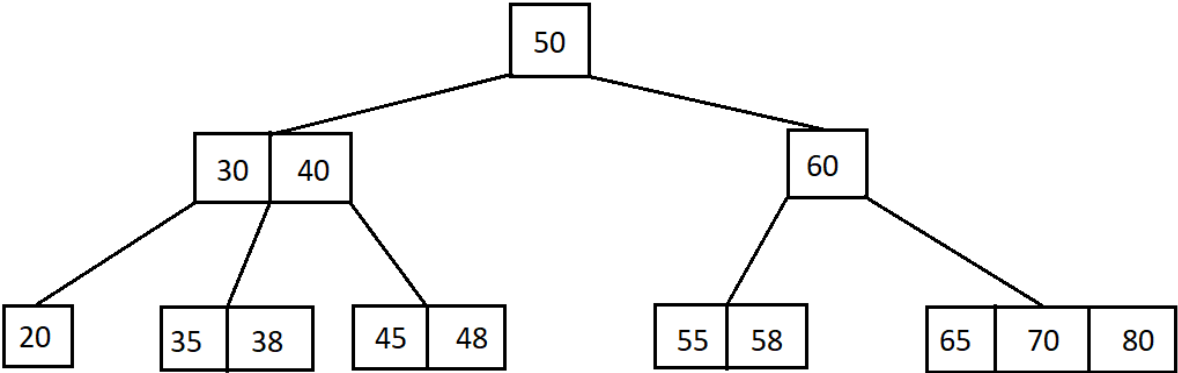
In the Red-Black tree representation:

- The black height of all leaf nodes is the same due to the black depth property of Red-Black trees.
- The children of red nodes are always black (or the equivalent of a leaf in a 2-3-4 tree), which ensures no two red nodes are adjacent, maintaining the property that red nodes must have black children.
- The tree remains balanced because operations on Red-Black trees are designed to maintain the black height balanced after insertions and deletions.

Given these transformation rules and the structure of the 2-3-4 tree, we can see how the example 2-3-4 tree can be transformed into an equivalent Red-Black tree. For instance:

- The 3-node with keys [30, 40] transforms into a black node with key 40 and a red child with key 30.
- The 4-node with keys [65, 70, 80] becomes a black node with key 70, and it has two red children with keys 65 and 80.

Following the Red-Black tree rules, the black nodes on any path from the root to a leaf have the same number of black nodes, ensuring the tree remains balanced. This structure allows the Red-Black tree to maintain efficient operations, with time complexities for insertion, deletion, and search remaining $O(\log n)$, similar to those of the 2-3-4 tree.



- Testing:

Test Case 1	Insert elements and print a 234 tree in order
Input	<pre>// Create a 2-3-4 tree and insert some items. TwoThreeFourTree<int> tree234 = new TwoThreeFourTree<int>(); int[] valuesToInsert = { 50, 40, 60, 30, 45, 55, 70, 20, 35, 48, 58, 65, 80, 38}; foreach (var value in valuesToInsert) { tree234.Insert(value); }</pre>
Expected output	(20, 30, 35, 38, 40, 45, 48, 50, 55, 58, 60, 65, 70, 80)
Output	<pre>2-3-4 Tree before conversion: 20, 30, 35, 38, 40, 45, 48, 50, 55, 58, 60, 65, 70, 80</pre>

Test Case 2	Convert the tree to an rb tree
Input	<pre>// Create a 2-3-4 tree and insert some items. TwoThreeFourTree<int> tree234 = new TwoThreeFourTree<int>(); int[] valuesToInsert = { 50, 40, 60, 30, 45, 55, 70, 20, 35, 48, 58, 65, 80, 38}; foreach (var value in valuesToInsert) { tree234.Insert(value); }</pre> <pre>// Convert the 2-3-4 tree to a Red-Black tree. var rbTree = tree234.Convert(); // Print the Red-Black tree after conversion. Console.WriteLine("\nRed-Black Tree after conversion:"); rbTree.Print();</pre>
Expected output	Red - Black Tree
Output	<pre>Red-Black Tree after conversion: 80R 70B 65R 60R 58B 55R 50B 48B 45R 40B 38B 35R 30R 20B</pre>

Test Case 3	Remove an existing element and print a rb tree
Input	<pre>// Delete from 2-3-4 tree and print R-B tree if (tree234.Delete(38)) { Console.WriteLine("\nItem deleted."); Console.WriteLine("\n2-3-4 Tree After delete:"); tree234.Print(); rbTree = tree234.Convert(); Console.WriteLine("\nR-B Tree After delete:"); rbTree.Print(); }</pre>
Expected output	New 2-3-4 tree and Red-Black tree without 38 in it
Output	<pre>Item deleted. 2-3-4 Tree After delete: 20, 30, 35, 40, 45, 48, 50, 55, 58, 60, 65, 70, 80 R-B Tree After delete: 80R 70B 65R 60R 58B 55R 50B 48B 45R 40B 35B 30R 20B</pre>

Test Case 4	Remove a non-existing element
Input	<pre>// Delete from 2-3-4 tree and print R-B tree if (tree234.Delete(90)) { Console.WriteLine("\nItem deleted."); Console.WriteLine("\n2-3-4 Tree After delete:"); tree234.Print(); rbTree = tree234.Convert(); Console.WriteLine("\nR-B Tree After delete:"); rbTree.Print(); }</pre>
Expected output	Error saying item not found in tree
Output	Error : Item not found in tree

Test Case 5	Insert an already existing element
Input	<pre>// Trying to insert an item already in the tree if (tree234.Insert(38)) Console.WriteLine("\n Item Inserted"); else Console.WriteLine("\n Error : Item not Inserted");</pre>
Expected output	Item will not be inserted and will go for the else case
Output	Error : Item not Inserted