

CPP-Structs-Algorithms

Generated on Mon Apr 28 2025 04:03:35 for CPP-Structs-Algorithms by Doxygen 1.13.2

Mon Apr 28 2025 04:03:35

1 Class Index	1
1.1 Class List	1
2 File Index	3
2.1 File List	3
3 Class Documentation	5
3.1 APSP Class Reference	5
3.1.1 Detailed Description	5
3.1.2 Constructor & Destructor Documentation	5
3.1.2.1 APSP()	5
3.1.3 Member Function Documentation	6
3.1.3.1 addEdge()	6
3.1.3.2 getCost()	6
3.1.3.3 printAdjacency()	6
3.2 CircularDynamicArray< elmttype > Class Template Reference	7
3.2.1 Detailed Description	8
3.2.2 Constructor & Destructor Documentation	8
3.2.2.1 CircularDynamicArray() [1/3]	8
3.2.2.2 CircularDynamicArray() [2/3]	8
3.2.2.3 CircularDynamicArray() [3/3]	8
3.2.2.4 ~CircularDynamicArray()	9
3.2.3 Member Function Documentation	9
3.2.3.1 addEnd()	9
3.2.3.2 addFront()	9
3.2.3.3 binSearch()	9
3.2.3.4 delEnd()	10
3.2.3.5 delFront()	10
3.2.3.6 linearSearch()	10
3.2.3.7 QuickSelect()	10
3.2.3.8 stableSort()	11
3.2.3.9 WCSelect()	11
3.3 Fib Class Reference	11
3.3.1 Detailed Description	12
3.3.2 Constructor & Destructor Documentation	12
3.3.2.1 Fib()	12
3.3.3 Member Function Documentation	12
3.3.3.1 print()	12
3.3.3.2 printAll()	12
3.4 Heap< keyType > Class Template Reference	13
3.4.1 Detailed Description	13
3.4.2 Member Function Documentation	13
3.4.2.1 extractMin()	13

3.4.2.2 insert()	14
3.4.2.3 peekKey()	14
3.4.2.4 printKeys()	14
3.5 Knapsack Class Reference	14
3.5.1 Detailed Description	15
3.5.2 Constructor & Destructor Documentation	15
3.5.2.1 Knapsack()	15
3.6 LCS Class Reference	15
3.6.1 Detailed Description	16
3.6.2 Constructor & Destructor Documentation	16
3.6.2.1 LCS()	16
3.6.2.2 ~LCS()	16
3.6.3 Member Function Documentation	17
3.6.3.1 get()	17
3.6.3.2 newStrings()	17
3.7 Matrix Class Reference	17
3.7.1 Detailed Description	18
3.7.2 Member Function Documentation	18
3.7.2.1 print()	18
3.8 MatrixChain Class Reference	18
3.8.1 Detailed Description	19
3.9 RBNode< keyType, valueType > Class Template Reference	19
3.9.1 Detailed Description	20
3.9.2 Constructor & Destructor Documentation	20
3.9.2.1 RBNode() [1/4]	20
3.9.2.2 RBNode() [2/4]	20
3.9.2.3 RBNode() [3/4]	21
3.9.2.4 RBNode() [4/4]	21
3.9.2.5 ~RBNode()	21
3.9.3 Member Function Documentation	22
3.9.3.1 cascade()	22
3.9.3.2 inorder()	22
3.9.3.3 min()	22
3.9.3.4 postorder()	22
3.9.3.5 predecessor()	23
3.9.3.6 preorder()	23
3.9.3.7 printk()	23
3.9.3.8 printNode()	23
3.9.3.9 searchNode()	24
3.9.3.10 searchValue()	24
3.9.3.11 select()	24
3.10 RBTree< keyType, valueType > Class Template Reference	25

3.10.1 Detailed Description	26
3.10.2 Constructor & Destructor Documentation	26
3.10.2.1 RBTree() [1/2]	26
3.10.2.2 RBTree() [2/2]	27
3.10.2.3 ~RBTree()	27
3.10.3 Member Function Documentation	27
3.10.3.1 inorder()	27
3.10.3.2 insert()	27
3.10.3.3 operator=()	28
3.10.3.4 postorder()	28
3.10.3.5 predecessor()	28
3.10.3.6 preorder()	29
3.10.3.7 printk()	29
3.10.3.8 rank()	29
3.10.3.9 remove()	29
3.10.3.10 search()	30
3.10.3.11 select()	31
3.10.3.12 size()	31
3.10.3.13 successor()	31
4 File Documentation	33
4.1 /Users/jace/Documents/GitHub/CPP-Structs-Algorithms/algorithms.hpp	33
4.2 /Users/jace/Documents/GitHub/CPP-Structs-Algorithms/datastructs.hpp	39
Index	53

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

APSP		
	All-Pairs Shortest Path (APSP) class	5
CircularDynamicArray< elmtype >		
	Implements a circular dynamic array that can dynamically resize itself	7
Fib		
	Calculates fibonacci numbers using a recursive, dynamic programming approach	11
Heap< keyType >		
	Implements a Minimum Heap	13
Knapsack		
	Stores information for and solves the 0-1 knapsack problem using dynamic programming	14
LCS		
	Finds the longest common subsequence (LCS) of two strings	15
Matrix		
	Represents a matrix	17
MatrixChain		
	Represents a chain of matrices	18
RBNode< keyType, valueType >		
	Node for a Red-Black Tree	19
RBTree< keyType, valueType >		
	Implements a Red-Black Tree	25

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

/Users/jace/Documents/GitHub/CPP-Structs-Algorithms/ algorithms.hpp	33
/Users/jace/Documents/GitHub/CPP-Structs-Algorithms/ datastructs.hpp	39

Chapter 3

Class Documentation

3.1 APSP Class Reference

All-Pairs Shortest Path ([APSP](#)) class.

```
#include <algorithms.hpp>
```

Public Member Functions

- [APSP](#) (int n)
Constructor for the [APSP](#) class.
- void [addEdge](#) (int i, int j, int cost)
Adds an edge to the graph.
- int [getCost](#) (int i, int j)
Gets the cost of the edge from vertex i to vertex j.
- void [printAdjacency](#) (std::ostream &out=std::cout)
Prints the adjacency matrix of the graph.

3.1.1 Detailed Description

All-Pairs Shortest Path ([APSP](#)) class.

This class implements the Floyd-Warshall algorithm to find the shortest paths between all pairs of vertices in a directed, weighted graph. Negative vertex cycles will be detected and will throw runtime errors.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 APSP()

```
APSP::APSP (  
    int n) [inline]
```

Constructor for the [APSP](#) class.

Initializes the adjacency matrix for n vertices.

Parameters

<i>n</i>	Number of vertices in the graph
----------	---------------------------------

3.1.3 Member Function Documentation

3.1.3.1 addEdge()

```
void APSP::addEdge (  
    int i,  
    int j,  
    int cost) [inline]
```

Adds an edge to the graph.

Parameters

<i>i</i>	The starting vertex (0-indexed)
<i>j</i>	The ending vertex (0-indexed)
<i>cost</i>	The cost of the edge from vertex i to vertex j

3.1.3.2 getCost()

```
int APSP::getCost (  
    int i,  
    int j) [inline]
```

Gets the cost of the edge from vertex i to vertex j.

Parameters

<i>i</i>	The starting vertex (0-indexed)
<i>j</i>	The ending vertex (0-indexed)

3.1.3.3 printAdjacency()

```
void APSP::printAdjacency (  
    std::ostream & out = std::cout) [inline]
```

Prints the adjacency matrix of the graph.

Parameters

<i>out</i>	Output stream to print to (default is std::cout)
------------	--

The documentation for this class was generated from the following file:

- /Users/jace/Documents/GitHub/Cpp-Structs-Algorithms/algorithms.hpp

3.2 CircularDynamicArray< elmtyp> Class Template Reference

Implements a circular dynamic array that can dynamically resize itself.

```
#include <datastructs.hpp>
```

Public Member Functions

- [CircularDynamicArray](#) ()
Default Constructor.
- [CircularDynamicArray](#) (int capacity)
Constructor for a set capacity.
- [CircularDynamicArray](#) (CircularDynamicArray const &src)
Copy constructor (deep copy)
- [~CircularDynamicArray](#) ()
Destructor.
- int **length** ()
Returns the size of the array.
- int **capacity** ()
Returns the capacity of the array.
- elmtyp **atRef** (int i)
Returns the value of the element at index i.
- elmtyp * **atPoint** (int i)
Returns a pointer to the element at index i.
- elmtyp & **operator[]** (int i)
Bracket operator. Returns burner element if index is invalid.
- [CircularDynamicArray](#) & **operator=** (const [CircularDynamicArray](#) &R)
Equals operator. Deep copies all values.
- void **addFront** (elmtyp v)
Adds an element to the front of the array.
- void **addEnd** (elmtyp v)
Adds an element to the end of the array.
- void **delFront** ()
Removes the element at the front of the array.
- void **delEnd** ()
Removes the element at the end of the array.
- void **clear** ()
Clears all array data and resets the array to its default state.
- void **swap** (elmtyp *a, elmtyp *b)
Swaps two elements.
- elmtyp **QuickSelect** (int k)
Returns the Kth smallest element of the array using the median as the partition element.
- elmtyp **WCSelect** (int k)
Returns the Kth smallest element of the array using the median of medians (subarray size = 5) as the partition element.
- void **stableSort** ()
Performs a mergesort on the array.
- int **linearSearch** (elmtyp e)
Performs a linear search for element e.
- int **binSearch** (elmtyp e)
Performs a binary search for element e.

3.2.1 Detailed Description

```
template<typename elmttype>
class CircularDynamicArray< elmttype >
```

Implements a circular dynamic array that can dynamically resize itself.

Template Parameters

<i>elmttype</i>	The type of element stored in the circular dynamic array
-----------------	--

3.2.2 Constructor & Destructor Documentation

3.2.2.1 CircularDynamicArray() [1/3]

```
template<typename elmttype>
CircularDynamicArray< elmttype >::CircularDynamicArray () [inline]
```

Default Constructor.

Initializes the circular dynamic array with a capacity of 2. End is set to -1 for [addEnd\(\)](#) and [addFront\(\)](#) functionality.

3.2.2.2 CircularDynamicArray() [2/3]

```
template<typename elmttype>
CircularDynamicArray< elmttype >::CircularDynamicArray (
    int capacity) [inline]
```

Constructor for a set capacity.

Initializes the circular dynamic array with a set capacity. End is set to capacity - 1 for [addEnd\(\)](#) and [addFront\(\)](#) functionality.

Parameters

<i>capacity</i>	The capacity of the circular dynamic array
-----------------	--

3.2.2.3 CircularDynamicArray() [3/3]

```
template<typename elmttype>
CircularDynamicArray< elmttype >::CircularDynamicArray (
    CircularDynamicArray< elmttype > const & src) [inline]
```

Copy constructor (deep copy)

Deep copies all values from the source circular dynamic array

Parameters

<i>src</i>	The source circular dynamic array to copy
------------	---

3.2.2.4 ~CircularDynamicArray()

```
template<typename elmtyp>
CircularDynamicArray< elmtyp >::~~CircularDynamicArray () [inline]
```

Destructor.

Deletes the array data

3.2.3 Member Function Documentation

3.2.3.1 addEnd()

```
template<typename elmtyp>
void CircularDynamicArray< elmtyp >::addEnd (
    elmtyp v) [inline]
```

Adds an element to the end of the array.

Time complexity: O(1) (Amortized)

3.2.3.2 addFront()

```
template<typename elmtyp>
void CircularDynamicArray< elmtyp >::addFront (
    elmtyp v) [inline]
```

Adds an element to the front of the array.

Time complexity: O(1) (Amortized)

3.2.3.3 binSearch()

```
template<typename elmtyp>
int CircularDynamicArray< elmtyp >::binSearch (
    elmtyp e) [inline]
```

Performs a binary search for element e.

Time complexity: O(lg(size))

Parameters

<i>e</i>	The element to search for
----------	---------------------------

Returns

The index of the element if found, -1 otherwise

3.2.3.4 delEnd()

```
template<typename elmtype>
void CircularDynamicArray< elmtype >::delEnd () [inline]
```

Removes the element at the end of the array.

Time complexity: $O(1)$ (Amortized)

3.2.3.5 delFront()

```
template<typename elmtype>
void CircularDynamicArray< elmtype >::delFront () [inline]
```

Removes the element at the front of the array.

Time complexity: $O(1)$ (Amortized)

3.2.3.6 linearSearch()

```
template<typename elmtype>
int CircularDynamicArray< elmtype >::linearSearch (
    elmtype e) [inline]
```

Performs a linear search for element e .

Time complexity: $O(\text{size})$

Parameters

e	The element to search for
-----	---------------------------

Returns

The index of the element if found, -1 otherwise

3.2.3.7 QuickSelect()

```
template<typename elmtype>
elmtype CircularDynamicArray< elmtype >::QuickSelect (
    int k) [inline]
```

Returns the Kth smallest element of the array using the median as the partition element.

Time complexity: $O(\text{size})$

Parameters

k	The index of the element to find
-----	----------------------------------

Returns

The Kth smallest element of the array

3.2.3.8 stableSort()

```
template<typename elmttype>
void CircularDynamicArray< elmttype >::stableSort () [inline]
```

Performs a mergesort on the array.

Time complexity: $O(\text{size} * \lg(\text{size}))$

3.2.3.9 WCSelect()

```
template<typename elmttype>
elmttype CircularDynamicArray< elmttype >::WCSelect (
    int k) [inline]
```

Returns the Kth smallest element of the array using the median of medians (subarray size = 5) as the partition element.

Time complexity: $O(\text{size})$

Parameters

<i>k</i>	The index of the element to find
----------	----------------------------------

Returns

The Kth smallest element of the array

The documentation for this class was generated from the following file:

- /Users/jace/Documents/GitHub/Cpp-Structs-Algorithms/datastructs.hpp

3.3 Fib Class Reference

Calculates fibonacci numbers using a recursive, dynamic programming approach.

```
#include <algorithms.hpp>
```

Public Member Functions

- [Fib](#) ()
Constructor to initialize the Fibonacci class.
- void [print](#) (int n, std::ostream &out=std::cout)
Print the Fibonacci number for a given n.
- void [printAll](#) (int n, std::ostream &out=std::cout)
Print all Fibonacci numbers from 0 to n.

3.3.1 Detailed Description

Calculates fibonacci numbers using a recursive, dynamic programming approach.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 Fib()

```
Fib::Fib () [inline]
```

Constructor to initialize the Fibonacci class.

Initializes the first two Fibonacci numbers and clears the rest of the array

3.3.3 Member Function Documentation

3.3.3.1 print()

```
void Fib::print (
    int n,
    std::ostream & out = std::cout) [inline]
```

Print the Fibonacci number for a given n.

Parameters

<i>n</i>	The integer value for which to calculate and print the Fibonacci number
----------	---

3.3.3.2 printAll()

```
void Fib::printAll (
    int n,
    std::ostream & out = std::cout) [inline]
```

Print all Fibonacci numbers from 0 to n.

Parameters

<i>n</i>	Max number to print Fibonacci numbers for (inclusive) (max 185)
----------	---

The documentation for this class was generated from the following file:

- /Users/jace/Documents/GitHub/CPP-Structs-Algorithms/algorithms.hpp

3.4 Heap< keyType > Class Template Reference

Implements a Minimum [Heap](#).

```
#include <datastructs.hpp>
```

Public Member Functions

- **Heap** ()
Default Constructor.
- **Heap** (keyType K[], int s)
Constructs a heap with keys K and size s.
- **~Heap** ()
Destructor.
- int **size** ()
Returns the size of the heap.
- keyType **peekKey** ()
Returns the minimum key in the heap.
- void **insert** (keyType k)
Inserts a new node with key k.
- void **printKeys** (std::ostream &out=std::cout)
Prints all keys in the heap in level order.
- keyType **extractMin** ()
Removes the minimum key from the heap and restores heap priority.

3.4.1 Detailed Description

```
template<typename keyType>
class Heap< keyType >
```

Implements a Minimum [Heap](#).

Template Parameters

<i>keyType</i>	The type of key stored in the heap
----------------	------------------------------------

3.4.2 Member Function Documentation

3.4.2.1 extractMin()

```
template<typename keyType>
keyType Heap< keyType >::extractMin () [inline]
```

Removes the minimum key from the heap and restores heap priority.

Time complexity: $O(\lg(n))$, n = size

Returns

The minimum key

3.4.2.2 insert()

```
template<typename keyType>
void Heap< keyType >::insert (
    keyType k) [inline]
```

Inserts a new node with key k.

Time complexity: $O(\lg(n))$, n = size

Parameters

<i>k</i>	The key to insert
----------	-------------------

3.4.2.3 peekKey()

```
template<typename keyType>
keyType Heap< keyType >::peekKey () [inline]
```

Returns the minimum key in the heap.

Time complexity: $O(1)$

Returns

The minimum key

3.4.2.4 printKeys()

```
template<typename keyType>
void Heap< keyType >::printKeys (
    std::ostream & out = std::cout) [inline]
```

Prints all keys in the heap in level order.

Parameters

<i>out</i>	The output stream to print to, defaulting to std::cout
------------	--

The documentation for this class was generated from the following file:

- /Users/jace/Documents/GitHub/CPP-Structs-Algorithms/datastructs.hpp

3.5 Knapsack Class Reference

Stores information for and solves the 0-1 knapsack problem using dynamic programming.

```
#include <algorithms.hpp>
```

Public Member Functions

- [Knapsack](#) (int n, int max, int *p, int *w)
Constructor.
- void **printTable** ()
Print the dynamic programming table for the knapsack problem.
- void **printPWO** (std::ostream &out=std::cout)
Print the profits, weights, and objects chosen for the knapsack problem.
- int **getProfit** ()
Get the maximum profit of the knapsack problem.

3.5.1 Detailed Description

Stores information for and solves the 0-1 knapsack problem using dynamic programming.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Knapsack()

```
Knapsack::Knapsack (
    int n,
    int max,
    int * p,
    int * w) [inline]
```

Constructor.

NOTE: p and w MUST be 1-indexed

Parameters

<i>n</i>	Number of items in the knapsack
<i>max</i>	Maximum weight of the knapsack
<i>p</i>	Array of profits for each item (1-indexed)
<i>w</i>	Array of weights for each item (1-indexed)

The documentation for this class was generated from the following file:

- /Users/jace/Documents/GitHub/Cpp-Structs-Algorithms/algorithms.hpp

3.6 LCS Class Reference

Finds the longest common subsequence ([LCS](#)) of two strings.

```
#include <algorithms.hpp>
```

Public Member Functions

- [LCS](#) (std::string a, std::string b)
Constructor to initialize the [LCS](#) object.
- [~LCS](#) ()
Destructor for the [LCS](#) object.
- std::string [get](#) ()
Retrieve the longest common subsequence of the two strings.
- int [lcsLength](#) ()
Get the length of the longest common subsequence.
- void [printMatrix](#) ()
Print the matrix used to find the [LCS](#).
- void [newStrings](#) (std::string a, std::string b)
Update the strings for which to find the [LCS](#).

3.6.1 Detailed Description

Finds the longest common subsequence ([LCS](#)) of two strings.

Functions requiring the [LCS](#) will calculate it before returning.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 [LCS\(\)](#)

```
LCS::LCS (
    std::string a,
    std::string b) [inline]
```

Constructor to initialize the [LCS](#) object.

Parameters

<i>a</i>	First std::string for which to find the LCS
<i>b</i>	Second std::string for which to find the LCS

3.6.2.2 [~LCS\(\)](#)

```
LCS::~LCS () [inline]
```

Destructor for the [LCS](#) object.

Nothing special, but destructor should be declared even if empty

3.6.3 Member Function Documentation

3.6.3.1 get()

```
std::string LCS::get () [inline]
```

Retrieve the longest common subsequence of the two strings.

Returns

The longest common subsequence of the two strings

3.6.3.2 newStrings()

```
void LCS::newStrings (
    std::string a,
    std::string b) [inline]
```

Update the strings for which to find the [LCS](#).

Parameters

<i>a</i>	New first std::string for which to find the LCS
<i>b</i>	New second std::string for which to find the LCS

The documentation for this class was generated from the following file:

- `/Users/jace/Documents/GitHub/Cpp-Structs-Algorithms/algorithms.hpp`

3.7 Matrix Class Reference

Represents a matrix.

```
#include <algorithms.hpp>
```

Public Member Functions

- **Matrix** (std::vector< std::vector< int > > m)
Default constructor.
- **Matrix** (int r, int c)
Constructor with dimensions; initializes the matrix with 0s.
- bool **operator==** (const [Matrix](#) &m)
Equality operator for matrices.
- void **clear** ()
Clears the matrix data.
- void **print** (std::ostream &out=std::cout)
Prints the matrix.

Public Attributes

- int **row**
Number of rows in the matrix.
- int **col**
Number of columns in the matrix.
- std::vector< std::vector< int > > **data**
Data of the matrix.

3.7.1 Detailed Description

Represents a matrix.

3.7.2 Member Function Documentation

3.7.2.1 print()

```
void Matrix::print (
    std::ostream & out = std::cout) [inline]
```

Prints the matrix.

Parameters

<i>out</i>	Output stream to print to (default is std::cout)
------------	--

The documentation for this class was generated from the following file:

- /Users/jace/Documents/GitHub/CPP-Structs-Algorithms/algorithms.hpp

3.8 MatrixChain Class Reference

Represents a chain of matrices.

```
#include <algorithms.hpp>
```

Public Member Functions

- **MatrixChain** (std::vector< [Matrix](#) * > m)
Copy constructor (deep copy)
- void **addMatrix** ([Matrix](#) *m)
Add a matrix to the chain.
- [Matrix](#) * **solve** ()
Multiply the matrices and return the product.

3.8.1 Detailed Description

Represents a chain of matrices.

Given a chain of matrices, this class finds the optimal parenthesization of the matrices and returns the product

The documentation for this class was generated from the following file:

- /Users/jace/Documents/GitHub/Cpp-Structs-Algorithms/algorithms.hpp

3.9 RBNode< keyType, valueType > Class Template Reference

Node for a Red-Black Tree.

```
#include <datastructs.hpp>
```

Public Member Functions

- [RBNode](#) ()
Default constructor.
- [RBNode](#) (keyType k, valueType v)
Constructor for a node with specified key and value.
- [RBNode](#) (bool nilCon)
Constructor for a nil node.
- [RBNode](#) (keyType k, valueType v, color setc, int s, [RBNode](#)< keyType, valueType > *parent)
Constructor for a node with specified key, value, color, size, and parent.
- [RBNode](#) (const [RBNode](#)< keyType, valueType > &src)
Copy constructor (deep copy)
- [~RBNode](#) ()
Destructor.
- void [cascade](#) ([RBNode](#) *nil)
Deletes a node and all its children.
- [RBNode](#) & **operator=** ([RBNode](#) R)
Copy equals operator.
- void [preorder](#) (std::ostream &out=std::cout)
Prints the preorder traversal of the tree.
- void [inorder](#) (std::ostream &out=std::cout)
Prints the inorder traversal of the tree.
- void [postorder](#) (std::ostream &out=std::cout)
Prints the postorder traversal of the tree.
- void [printNode](#) (std::ostream &out=std::cout)
Prints the node's key.
- void [printk](#) (int &k, std::ostream &out=std::cout)
Prints the K smallest elements of the subtree rooted at this node.
- valueType * [searchValue](#) (keyType k)
Searches for the node with key k and returns a pointer to the value.
- [RBNode](#)< keyType, valueType > * [searchNode](#) (keyType k)
Searches the subtree rooted at this node for the node with key k.
- [RBNode](#)< keyType, valueType > * [predecessor](#) ()
Returns the predecessor of the node.
- [RBNode](#)< keyType, valueType > * [min](#) ()
Returns the smallest node of the subtree rooted at this node.
- keyType [select](#) (int k)
Returns the Kth smallest element in the subtree rooted at this node.

Public Attributes

- `keyType * key`
The key of the node.
- `valueType * val`
The value of the node.
- `RBNode * l`
The left child of the node.
- `RBNode * r`
The right child of the node.
- `RBNode * p`
The parent of the node.
- `color c`
The color of the node.
- `int size`
The size of the subtree rooted at this node.

3.9.1 Detailed Description

```
template<typename keyType, typename valueType>
class RBNode< keyType, valueType >
```

Node for a Red-Black Tree.

This class represents a node in a Red-Black Tree. It contains pointers to its left and right children, its parent, and its key and value. It also contains the color of the node (Red or Black) and the size of the subtree rooted at this node.

This class also has all of its elements set to public. This is because it is not intended to be used on its own; rather, it is for use by the `RBTree` class, which requires direct access to the node's elements to avoid unnecessary function calls.

Template Parameters

<code>keyType</code>	The type of key stored in the node
<code>valueType</code>	The type of value stored in the node

3.9.2 Constructor & Destructor Documentation

3.9.2.1 `RBNode()` [1/4]

```
template<typename keyType, typename valueType>
RBNode< keyType, valueType >::RBNode () [inline]
```

Default constructor.

Initializes the node with default values. The key and value pointers are dynamically allocated. Color is set to Red.

3.9.2.2 `RBNode()` [2/4]

```
template<typename keyType, typename valueType>
RBNode< keyType, valueType >::RBNode (
    keyType k,
    valueType v) [inline]
```

Constructor for a node with specified key and value.

Initializes the node with the specified key and value. The color is set to Red, size is set to 1, and relatives are set to null.

Parameters

<i>k</i>	The key of the node
<i>v</i>	The value of the node

3.9.2.3 RBNode() [3/4]

```
template<typename keyType, typename valueType>
RBNode< keyType, valueType >::RBNode (
    bool nilCon) [inline]
```

Constructor for a nil node.

Initializes the node as a nil node. The key and value pointers are set to null, and the color is set to Black.

Parameters

<i>nilCon</i>	A boolean value to indicate that this is a nil node. The value is not used.
---------------	---

3.9.2.4 RBNode() [4/4]

```
template<typename keyType, typename valueType>
RBNode< keyType, valueType >::RBNode (
    keyType k,
    valueType v,
    color setc,
    int s,
    RBNode< keyType, valueType > * parent) [inline]
```

Constructor for a node with specified key, value, color, size, and parent.

Parameters

<i>k</i>	The key of the node
<i>v</i>	The value of the node
<i>setc</i>	The color of the node
<i>s</i>	The size of the subtree rooted at this node
<i>parent</i>	The parent of the node

3.9.2.5 ~RBNode()

```
template<typename keyType, typename valueType>
RBNode< keyType, valueType >::~~RBNode () [inline]
```

Destructor.

Checks if the key and value pointers are not null before deleting them.

3.9.3 Member Function Documentation

3.9.3.1 cascade()

```
template<typename keyType, typename valueType>
void RBNode< keyType, valueType >::cascade (
    RBNode< keyType, valueType > * nil) [inline]
```

Deletes a node and all its children.

Deletes the node and all its children recursively. The nil node is passed as a parameter to avoid deleting it.

Parameters

<i>nil</i>	The nil node of the tree
------------	--------------------------

3.9.3.2 inorder()

```
template<typename keyType, typename valueType>
void RBNode< keyType, valueType >::inorder (
    std::ostream & out = std::cout) [inline]
```

Prints the inorder traversal of the tree.

Parameters

<i>out</i>	The output stream to print, default is cout
------------	---

3.9.3.3 min()

```
template<typename keyType, typename valueType>
RBNode< keyType, valueType > * RBNode< keyType, valueType >::min () [inline]
```

Returns the smallest node of the subtree rooted at this node.

Returns the smallest node of the subtree rooted at this node. The smallest node is the leftmost node in the subtree. If the left child is null, it returns this node. Time complexity is $O(\lg(\text{size}))$

Returns

A pointer to the smallest node in the subtree rooted at this node

3.9.3.4 postorder()

```
template<typename keyType, typename valueType>
void RBNode< keyType, valueType >::postorder (
    std::ostream & out = std::cout) [inline]
```

Prints the postorder traversal of the tree.

Parameters

<i>out</i>	The output stream to print, default is cout
------------	---

3.9.3.5 predecessor()

```
template<typename keyType, typename valueType>
RBNode< keyType, valueType > * RBNode< keyType, valueType >::predecessor () [inline]
```

Returns the predecessor of the node.

Returns the predecessor of the node. The predecessor is the largest node in the left subtree. If the left child is null, it returns nullptr. Time complexity is $O(\lg(\text{size}))$

Returns

A pointer to the predecessor node, or nullptr if the left child is null

3.9.3.6 preorder()

```
template<typename keyType, typename valueType>
void RBNode< keyType, valueType >::preorder (
    std::ostream & out = std::cout) [inline]
```

Prints the preorder traversal of the tree.

Parameters

<i>out</i>	The output stream to print, default is cout
------------	---

3.9.3.7 printk()

```
template<typename keyType, typename valueType>
void RBNode< keyType, valueType >::printk (
    int & k,
    std::ostream & out = std::cout) [inline]
```

Prints the K smallest elements of the subtree rooted at this node.

Prints the K smallest elements of the subtree rooted at this node. The function is called recursively on the left and right children. Time complexity is $O(k + \lg(\text{size}))$

Parameters

<i>k</i>	The number of elements to print
----------	---------------------------------

3.9.3.8 printNode()

```
template<typename keyType, typename valueType>
void RBNode< keyType, valueType >::printNode (
    std::ostream & out = std::cout) [inline]
```

Prints the node's key.

Parameters

<i>out</i>	The output stream to print, default is cout
------------	---

3.9.3.9 searchNode()

```
template<typename keyType, typename valueType>
RBNode< keyType, valueType > * RBNode< keyType, valueType >::searchNode (
    keyType k) [inline]
```

Searches the subtree rooted at this node for the node with key k.

Performs a DFS search for the node with key k. If the key is found, it returns a pointer to the node. Otherwise, it returns nullptr. Time complexity is $O(\lg(\text{size}))$

Parameters

<i>k</i>	The key to search for
----------	-----------------------

Returns

A pointer to the node with key k, or nullptr if the key is not found

3.9.3.10 searchValue()

```
template<typename keyType, typename valueType>
valueType * RBNode< keyType, valueType >::searchValue (
    keyType k) [inline]
```

Searches for the node with key k and returns a pointer to the value.

Performs a DFS search for the node with key k. If the key is found, it returns a pointer to the value of the node. Otherwise, it returns nullptr. Time complexity is $O(\lg(\text{size}))$

Parameters

<i>k</i>	The key to search for
----------	-----------------------

Returns

A pointer to the value of the node with key k, or nullptr if the key is not found

3.9.3.11 select()

```
template<typename keyType, typename valueType>
keyType RBNode< keyType, valueType >::select (
    int k) [inline]
```

Returns the Kth smallest element in the subtree rooted at this node.

Returns the Kth smallest element in the subtree rooted at this node. The function is called recursively on the left and right children. Time complexity is $O(\lg(\text{size}))$

Parameters

<i>k</i>	The index of the smallest element to return
----------	---

The documentation for this class was generated from the following file:

- /Users/jace/Documents/GitHub/Cpp-Structs-Algorithms/datastructs.hpp

3.10 RBTREE< keyType, valueType > Class Template Reference

Implements a Red-Black Tree.

```
#include <datastructs.hpp>
```

Public Member Functions

- **RBTREE** ()
Default constructor.
- **RBTREE** (keyType k, valueType v)
Constructs a tree with a single node with key k and value v.
- **RBTREE** (keyType *k, valueType *v, int s)
Constructs a tree with arrays k and v as the insert values and size = s.
- **RBTREE** (const **RBTREE**< keyType, valueType > &src)
Copy constructor (deep copy)
- **~RBTREE** ()
Destructor.
- **RBTREE**< keyType, valueType > & **operator=** (const **RBTREE**< keyType, valueType > &R)
Copy equals operator.
- void **preorder** (std::ostream &out)
Prints the preorder traversal of the tree.
- void **inorder** (std::ostream &out)
Prints the inorder traversal of the tree.
- void **postorder** (std::ostream &out)
Prints the postorder traversal of the tree.
- void **printk** (int k, std::ostream &out)
Prints the K smallest elements of the tree.
- int **size** ()
Returns the size of the tree.
- valueType * **search** (keyType k)
Searches the tree for key k and returns a pointer to the node's value.
- void **insert** (keyType k, valueType v)
Inserts a new node with key k and value v into the tree.
- int **remove** (keyType k)
Removes the node with key k from the tree.
- int **rank** (keyType k)
Returns the rank of the node with key k.
- keyType **select** (int k)
Returns the Kth smallest element in the tree.
- keyType * **successor** (keyType k)
Finds the successor of the node with key k and returns a pointer to its key.
- keyType * **predecessor** (keyType k)
Finds the predecessor of the node with key k and returns a pointer to its key.

3.10.1 Detailed Description

```
template<typename keyType, typename valueType>
class RBTree< keyType, valueType >
```

Implements a Red-Black Tree.

Template Parameters

<i>keyType</i>	The type of key stored in the tree
<i>valueType</i>	The type of value stored in the tree

This class implements a Red-Black Tree, a self-balancing binary search tree. It maintains balance using rotations and recoloring during insertions and deletions, ensuring efficient operations with a time complexity of $O(\log n)$.

The tree has the following properties:

- Each node has a color (Red or Black).
- The root node is always Black.
- Red nodes cannot have Red children (no two consecutive Red nodes).
- Every path from a node to its descendant nil nodes has the same number of Black nodes (Black height).
- The nil node (a sentinel node representing the end of the tree) is always Black.

These properties ensure that the tree remains approximately balanced, making it suitable for applications requiring fast lookups, insertions, and deletions.

3.10.2 Constructor & Destructor Documentation

3.10.2.1 RBTree() [1/2]

```
template<typename keyType, typename valueType>
RBTree< keyType, valueType >::RBTree (
    keyType k,
    valueType v) [inline]
```

Constructs a tree with a single node with key *k* and value *v*.

This constructor initializes the tree with a single node with the given key and value. The nil node is created, and the root is set to the new node. The color of the root is set to Black. Time complexity: $O(1)$

Parameters

<i>k</i>	The key of the root node
<i>v</i>	The value of the root node

3.10.2.2 RBTree() [2/2]

```
template<typename keyType, typename valueType>
RBTree< keyType, valueType >::RBTree (
    keyType * k,
    valueType * v,
    int s) [inline]
```

Constructs a tree with arrays k and v as the insert values and size = s.

This constructor initializes the tree with the given arrays of keys and values. It creates a nil node and inserts each key-value pair into the tree. The root is set to nil initially. Time complexity: $O(s)$

Note

If s is larger than the size of either array, this will cause a segmentation fault. Any s smaller than the size of either array will work up to the specified index, but the rest of the array will be ignored.

Parameters

<i>k</i>	The array of keys to insert
<i>v</i>	The array of values to insert
<i>s</i>	The size of the arrays

3.10.2.3 ~RBTree()

```
template<typename keyType, typename valueType>
RBTree< keyType, valueType >::~~RBTree () [inline]
```

Destructor.

This destructor deletes the tree by calling the cascade function on the root node. The cascade function recursively deletes all nodes in the tree. Time complexity: $O(\text{size})$

3.10.3 Member Function Documentation

3.10.3.1 inorder()

```
template<typename keyType, typename valueType>
void RBTree< keyType, valueType >::inorder (
    std::ostream & out) [inline]
```

Prints the inorder traversal of the tree.

This function prints the inorder traversal of the tree. It calls the root's inorder function, which prints the nodes in inorder. Time complexity: $O(n)$

3.10.3.2 insert()

```
template<typename keyType, typename valueType>
void RBTree< keyType, valueType >::insert (
    keyType k,
    valueType v) [inline]
```

Inserts a new node with key k and value v into the tree.

This function inserts a new node with key k and value v into the tree. It first creates a new node and sets its size to 1. Then, it traverses the tree to find the correct position for the new node. Finally, it calls the insertFixTree function to restore the Red-Black Tree properties. Time complexity: $O(\lg(\text{size}))$

Parameters

k	The key of the new node
v	The value of the new node

3.10.3.3 operator=()

```
template<typename keyType, typename valueType>
RBTree< keyType, valueType > & RBTree< keyType, valueType >::operator= (
    const RBTree< keyType, valueType > & R) [inline]
```

Copy equals operator.

This operator assigns the values of the source tree to the current tree. It first deletes the current tree and then copies the values from the source tree. Time complexity: $O(n)$, $n = R.root->size$

Parameters

R	The source tree to copy from
-----	------------------------------

Returns

A reference to the current tree

3.10.3.4 postorder()

```
template<typename keyType, typename valueType>
void RBTree< keyType, valueType >::postorder (
    std::ostream & out) [inline]
```

Prints the postorder traversal of the tree.

This function prints the postorder traversal of the tree. It calls the root's postorder function, which prints the nodes in postorder. Time complexity: $O(n)$

3.10.3.5 predecessor()

```
template<typename keyType, typename valueType>
keyType * RBTree< keyType, valueType >::predecessor (
    keyType k) [inline]
```

Finds the predecessor of the node with key k and returns a pointer to its key.

This function finds the predecessor of the node with key k . The predecessor is the largest node in the left subtree. If the left child is null, it returns nullptr. Time complexity: $O(\lg(\text{size}))$

Parameters

k	The key of the node whose predecessor is to be found
-----	--

Returns

A pointer to the key of the predecessor node, or nullptr if there is no predecessor

3.10.3.6 preorder()

```
template<typename keyType, typename valueType>
void RBTREE< keyType, valueType >::preorder (
    std::ostream & out) [inline]
```

Prints the preorder traversal of the tree.

This function prints the preorder traversal of the tree. It calls the root's preorder function, which prints the nodes in preorder. Time complexity: $O(n)$

3.10.3.7 printk()

```
template<typename keyType, typename valueType>
void RBTREE< keyType, valueType >::printk (
    int k,
    std::ostream & out) [inline]
```

Prints the K smallest elements of the tree.

This function prints the K smallest elements of the tree. It calls the root's printk function, which prints the K smallest elements in preorder traversal. Time complexity: $O(k + \lg(\text{size}))$

3.10.3.8 rank()

```
template<typename keyType, typename valueType>
int RBTREE< keyType, valueType >::rank (
    keyType k) [inline]
```

Returns the rank of the node with key k.

This function returns the rank of the node with key k. The rank is the number of nodes with keys less than k. If the key is not found, it returns 0. Time complexity: $O(\lg(\text{root->size}))$

3.10.3.9 remove()

```
template<typename keyType, typename valueType>
int RBTREE< keyType, valueType >::remove (
    keyType k) [inline]
```

Removes the node with key k from the tree.

This function removes the node with key k from the tree. It first searches for the node with key k. If the node is found, it removes it and restores the Red-Black Tree properties. Time complexity: $O(\lg(\text{size}))$

Parameters

k	The key of the node to be removed
-----	-----------------------------------

Returns

1 if the node was removed, 0 if the node was not found.

3.10.3.10 search()

```
template<typename keyType, typename valueType>
valueType * RBTtree< keyType, valueType >::search (
    keyType k) [inline]
```

Searches the tree for key k and returns a pointer to the node's value.

This function searches the tree for key k and returns a pointer to the node's value. If the key is not found, it returns nullptr. Time complexity: $O(\lg(\text{size}))$

Parameters

<i>k</i>	The key to search for
----------	-----------------------

Returns

A pointer to the value of the node with key *k*, or nullptr if the key is not found

3.10.3.11 select()

```
template<typename keyType, typename valueType>
keyType RBTREE< keyType, valueType >::select (
    int k) [inline]
```

Returns the Kth smallest element in the tree.

Note

This function currently is delegated to a function in the Node but should be migrated to the tree. This was done due to the original project specifications.

This function returns the Kth smallest element in the tree. The Kth smallest element is the node with rank *k*. If *k* is out of bounds or not found, it returns nullptr.

Parameters

<i>k</i>	The rank of the smallest element to return
----------	--

Returns

A pointer to the Kth smallest element in the tree, or nullptr if *k* is out of bounds

3.10.3.12 size()

```
template<typename keyType, typename valueType>
int RBTREE< keyType, valueType >::size () [inline]
```

Returns the size of the tree.

Returns

The size of the tree

3.10.3.13 successor()

```
template<typename keyType, typename valueType>
keyType * RBTREE< keyType, valueType >::successor (
    keyType k) [inline]
```

Finds the successor of the node with key *k* and returns a pointer to its key.

This function finds the successor of the node with key *k*. The successor is the smallest node in the right subtree. If the right child is null, it returns nullptr. Time complexity: $O(\lg(\text{size}))$

Parameters

k	The key of the node whose successor is to be found
-----	--

Returns

A pointer to the key of the successor node, or nullptr if there is no successor

The documentation for this class was generated from the following file:

- /Users/jace/Documents/GitHub/Cpp-Structs-Algorithms/datastructs.hpp

Chapter 4

File Documentation

4.1 /Users/jace/Documents/GitHub/CPP-Structs- Algorithms/algorithms.hpp

```
00001
00014
00015 #ifndef ALGORITHMS_H
00016 #define ALGORITHMS_H
00017
00018 #include <iostream>
00019 #include <climits>
00020 #include <vector>
00021 #include <ctime>
00022
00030 class APSP {
00031
00037     std::vector< std::vector<int> > adjacency;
00038
00042     int n;
00043
00047     bool built;
00048
00052     void build() {
00053         for (int i = 0; i < n; i++) {
00054             for (int j = 0; j < n; j++) {
00055                 for (int k = 0; k < n; k++) {
00056                     if (adjacency[j][i] == INT_MAX || adjacency[i][k] == INT_MAX) {
00057                         continue;
00058                     }
00059                     if (adjacency[j][i] + adjacency[i][k] < adjacency[j][k]) {
00060                         adjacency[j][k] = adjacency[j][i] + adjacency[i][k];
00061                     }
00062                 }
00063             }
00064         }
00065
00066         // After table is built, check for negative cycles and set built to true
00067         checkNegativeCycle();
00068         built = true;
00069     }
00070
00074     void checkNegativeCycle() {
00075         for (int i = 0; i < n; i++) {
00076             if (adjacency[i][i] < 0) {
00077                 throw std::runtime_error("Negative cycle detected at vertex " + std::to_string(i) +
00078                                         "\n");
00079             }
00080         }
00081
00082     public:
00083
00091     APSP(int n) {
00092         this->n = n;
00093         built = false;
00094         for (int i = 0; i < n; i++) {
00095             std::vector<int> row;
00096             for (int j = 0; j < n; j++) {
00097                 if (i == j) {
```

```

00098         row.push_back(0);
00099         continue;
00100     }
00101     row.push_back(INT_MAX);
00102 }
00103 adjacency.push_back(row);
00104 }
00105 }
00106
00114 void addEdge(int i, int j, int cost) {
00115     adjacency[i][j] = cost;
00116     built = false;
00117 }
00118
00125 int getCost(int i, int j) {
00126     if (!built) {
00127         build();
00128     }
00129
00130     return adjacency[i][j];
00131 }
00132
00138 void printAdjacency(std::ostream &out = std::cout) {
00139     if (!built) {
00140         build();
00141     }
00142
00143     for (int i = 0; i < n; i++) {
00144         std::cout << i + 1 << ": ";
00145         for (int j = 0; j < n; j++) {
00146             if (adjacency[i][j] == INT_MAX) {
00147                 std::cout << "inf ";
00148                 continue;
00149             }
00150             std::cout << adjacency[i][j] << " ";
00151         }
00152         std::cout << std::endl;
00153     }
00154 }
00155 };
00156
00160 class Fib {
00161
00162     __uint128_t fibList[186];
00163
00169     __uint128_t fib(int *n) {
00170
00171         // Check if number is too large for __uint128_t
00172         if (*n > 185) {
00173             throw std::runtime_error("Fibonacci number too large for unsigned 128-bit int\n");
00174         }
00175
00176         if (*n == 0) {
00177             return 0;
00178         }
00179
00180         // Check if number is stored
00181         if (fibList[*n] != 0) {
00182             return fibList[*n];
00183         }
00184
00185         // If number is not stored, calculate it
00186         int a = *n - 1;
00187         int b = *n - 2;
00188         fibList[*n] = fib(&a) + fib(&b);
00189
00190         return fibList[*n];
00191     }
00192
00193 public:
00194
00200     Fib() {
00201         // Set first two fibonacci numbers and clear all other data in map
00202         fibList[0] = 0;
00203         fibList[1] = 1;
00204         for (int i = 2; i < 186; i++) {
00205             fibList[i] = 0;
00206         }
00207     }
00208
00214     void print(int n, std::ostream &out = std::cout) {
00215
00216         __uint128_t fibNum = fib(&n);
00217         if (fibNum == 0) {
00218             out << "fib(" << n << ") = 0\n";
00219             return;
00220         }

```



```

00221
00222     std::string fibString;
00223     while (fibNum > 0) {
00224         fibString.insert(fibString.begin(), '0' + (fibNum % 10));
00225         fibNum /= 10;
00226     }
00227
00228     out << "fib(" << n << ") = " << fibString << std::endl;
00229 }
00230
00231 void printAll(int n, std::ostream &out = std::cout) {
00232
00233     if (n > 185) {
00234         throw std::runtime_error("Cannot print Fibonacci numbers greater than 185 with unsigned
128-bit int\n");
00240     }
00241
00242     // Print all fibonacci numbers up to n (inclusive)
00243     for (int i = 0; i < n; i++) {
00244         print(i, out);
00245     }
00246 }
00247 };
00248
00249 class Knapsack {
00250
00251     int n;
00252
00253     int max;
00254
00255     int *weights;
00256
00257     int *profits;
00258
00259     bool built;
00260
00261     bool *objects;
00262
00263     std::vector< std::vector<int> > table;
00264
00265     void build() {
00266
00267         // Table building loop
00268         for (int j = 0; j <= n; j++) {
00269             for (int k = 0; k <= max; k++) {
00270                 if (j == 0) {
00271                     table[j][k] = 0;
00272                 } else if (k < weights[j]) {
00273                     table[j][k] = table[j - 1][k];
00274                 } else {
00275                     // max() function is not functioning properly, so I manually find the max
00276                     int
00277                     a = table[j - 1][k],
00278                     b = table[j - 1][k - weights[j]] + profits[j];
00279                     if (a > b) {
00280                         table[j][k] = a;
00281                     } else {
00282                         table[j][k] = b;
00283                     }
00284                 }
00285             }
00286         }
00287
00288         // After table is built, choose the objects and set built to true
00289         choose();
00290         built = true;
00291     }
00292
00293     void choose() {
00294
00295         // Start at the bottom right corner of the table and work backwards
00296         int k = max;
00297         for (int i = n; i > 0; i--) {
00298             if (table[i-1][k] == table[i][k]) {
00299                 objects[i] = false;
00300             } else {
00301                 objects[i] = true;
00302                 k -= weights[i];
00303             }
00304         }
00305     }
00306
00307 public:
00308     Knapsack(int n, int max, int *p, int *w) {
00309
00310         this->n = n;

```

```

00352         this->max = max;
00353         built = false;
00354         objects = new bool[n+1];
00355         weights = w;
00356         profits = p;
00357
00358         for (int i = 0; i <= n; i++) {
00359             std::vector<int> temp;
00360             for (int j = 0; j <= max; j++) {
00361                 temp.push_back(0);
00362             }
00363             table.push_back(temp);
00364         }
00365     }
00366
00370     void printTable() {
00371         for (int i = 0; i <= n; i++) {
00372             for (int j = 0; j <= max; j++) {
00373                 std::cout << table[i][j] << " ";
00374             }
00375             std::cout << std::endl;
00376         }
00377     }
00378
00382     void printPWO(std::ostream &out = std::cout) {
00383         if (!built) {
00384             build();
00385         }
00386         out << "Profit: ";
00387         for (int i = 1; i <= n; i++) {
00388             out << profits[i] << " ";
00389         }
00390         out << std::endl;
00391         out << "Weight: ";
00392         for (int i = 1; i <= n; i++) {
00393             out << weights[i] << " ";
00394         }
00395         out << std::endl;
00396         out << "Objects Chosen: ";
00397         for (int i = 1; i <= n; i++) {
00398             out << objects[i] << " ";
00399         }
00400     }
00401
00405     int getProfit() {
00406         if (!built) {
00407             build();
00408         }
00409         return table[n][max];
00410     }
00411
00412 };
00413
00421 class LCS {
00422     private:
00423
00424         std::string s1;
00425
00426         std::string s2;
00427
00428         std::string foundLCS;
00429
00430         std::vector< std::vector<int> > L;
00431
00432         bool built;
00433
00434         std::string build(int j, int k) {
00435             if (j == 0 || k == 0) {
00436                 return "";
00437             }
00438             if (s1[j - 1] == s2[k - 1]) {
00439                 return build(j - 1, k - 1) + s1[j - 1];
00440             }
00441             if (L[j][k - 1] >= L[j - 1][k]) {
00442                 return build(j, k - 1);
00443             } else {
00444                 return build(j - 1, k);
00445             }
00446         }
00447
00448     public:
00449
00450         LCS(std::string a, std::string b) : s1(std::move(a)), s2(std::move(b)), built(false) {}
00451
00452         ~LCS() {
00453

```

```

00489
00495     std::string get() {
00496         if (built) {
00497             return foundLCS;
00498         }
00499
00500         int n = s1.length();
00501         int m = s2.length();
00502         L = std::vector<std::vector<int>>(n + 1, std::vector<int>(m + 1, 0));
00503
00504         for (int j = 1; j <= n; j++) {
00505             for (int k = 1; k <= m; k++) {
00506                 if (s1[j - 1] == s2[k - 1]) {
00507                     L[j][k] = L[j - 1][k - 1] + 1;
00508                 } else {
00509                     L[j][k] = std::max(L[j - 1][k], L[j][k - 1]);
00510                 }
00511             }
00512         }
00513
00514         foundLCS = build(n, m);
00515         built = true;
00516         return foundLCS;
00517     }
00518
00522     int lcsLength() {
00523         if (!built) {
00524             get();
00525         }
00526         return foundLCS.length();
00527     }
00528
00532     void printMatrix() {
00533
00534         // Ensure LCS is built before printing the matrix
00535         if (!built) {
00536             get();
00537         }
00538
00539         std::cout << "Matrix L: " << std::endl;
00540         for (int i = 0; i <= s1.length(); i++) {
00541             for (int j = 0; j <= s2.length(); j++) {
00542                 std::cout << L[i][j] << " ";
00543             }
00544             std::cout << std::endl;
00545         }
00546     }
00547
00554     void newStrings(std::string a, std::string b) {
00555         s1 = a;
00556         s2 = b;
00557         built = false;
00558         foundLCS = "";
00559         L.clear();
00560     }
00561 };
00562
00569 class Matrix {
00570     public:
00571
00575     int row;
00576
00580     int col;
00581
00585     std::vector< std::vector<int> > data;
00586
00590     Matrix(std::vector< std::vector<int> > m): row(m.at(0).size()), col(m.size()), data(m) {}
00591
00595     Matrix(int r, int c): row(r), col(c), data(r, std::vector<int>(c, 0)) {}
00596
00600     bool operator==(const Matrix &m) {
00601         if (row != m.row || col != m.col) {
00602             return false;
00603         }
00604         for (int i = 0; i < row; i++) {
00605             for (int j = 0; j < col; j++) {
00606                 if (data[i][j] != m.data[i][j]) {
00607                     return false;
00608                 }
00609             }
00610         }
00611         return true;
00612     }
00613
00617     void clear() {
00618         row = -1;
00619         col = -1;

```

```

00620         data.clear();
00621     }
00622
00628     void print(std::ostream &out = std::cout) {
00629         for (int i = 0; i < col; i++) {
00630             for (int j = 0; j < row; j++) {
00631                 out << data[i][j] << " ";
00632             }
00633             out << std::endl;
00634         }
00635     }
00636 };
00637
00646 class MatrixChain {
00647
00651     int n;
00652
00656     bool calculated;
00657
00661     Matrix *solution;
00662
00666     Matrix *bestK;
00667
00671     Matrix *cost;
00672
00676     std::vector<int> dim;
00677
00681     std::vector<Matrix*> data;
00682
00686     void calcCosts() {
00687         for (int L = 1; L < n; L++) {
00688             for (int i = 0; i < n - L; i++) {
00689                 int j = i + L;
00690                 cost->data[i][j] = INT_MAX;
00691                 for (int k = i; k < j; k++) {
00692                     int q = cost->data[i][k] + cost->data[k + 1][j] + dim[i] * dim[k + 1] * dim[j] +
11;
00693                     if (q < cost->data[i][j]) {
00694                         cost->data[i][j] = q;
00695                         bestK->data[i][j] = k;
00696                     }
00697                 }
00698             }
00699         }
00700     }
00701
00705     Matrix *matrixMult(Matrix *a, Matrix *b) {
00706         Matrix *c = new Matrix(a->row, b->col);
00707         for (int i = 0; i < a->row; i++) {
00708             for (int j = 0; j < b->col; j++) {
00709                 for (int k = 0; k < a->col; k++) {
00710                     c->data[i][j] += a->data[i][k] * b->data[k][j];
00711                 }
00712             }
00713         }
00714         return c;
00715     }
00716
00725     Matrix *chainProduct(int i, int j) {
00726         if (i == j) {
00727             return data[i];
00728         }
00729         int k = bestK->data[i][j];
00730         Matrix *a = chainProduct(i, k);
00731         Matrix *b = chainProduct(k + 1, j);
00732         return matrixMult(a, b);
00733     }
00734
00735 public:
00736
00740     MatrixChain(std::vector<Matrix*> m) {
00741         n = m.size();
00742         calculated = false;
00743         solution = new Matrix(n, n);
00744         bestK = new Matrix(n, n);
00745         cost = new Matrix(n, n);
00746
00747         for (int i = 0; i < n; i++) {
00748             data.push_back(m[i]);
00749         }
00750
00751         for (int i = 0; i < n; i++) {
00752             dim.push_back(m[i]->row);
00753             if (i == 0) {
00754                 continue;
00755             }
00756             if (m[i-1]->col != m[i]->row) {

```

```

00757         throw std::runtime_error(
00758             "Matrix dimensions do not match in MatrixChain constructor between matrices " +
std::to_string(i-1) + " and " + std::to_string(i) + "\n" +
00759             "Col size of matrix " + std::to_string(i-1) + " (" + std::to_string(m[i-1]->col) +
00760             ") != Row size of matrix " + std::to_string(i) + " (" + std::to_string(m[i]->row)
+ ") \n"
00761         );
00762     }
00763 }
00764 }
00765
00766 void addMatrix(Matrix *m) {
00770     if (m->row != data[n - 1]->col) {
00771         throw std::runtime_error("Matrix dimensions do not match in MatrixChain::addMatrix");
00772         return;
00773     }
00774
00775     data.push_back(m);
00776     if (calculated) {
00777         calculated = false;
00778         delete solution;
00779     }
00780     n++;
00781 }
00782
00783 Matrix *solve() {
00787     if (!calculated) {
00788         calcCosts();
00789         solution = chainProduct(0, n - 1);
00790     }
00791     return solution;
00792 }
00793
00794 };
00795
00806 std::vector<std::vector<double>> worldSeries(int n, double *aProb) {
00807
00808     // Create matrix (Use std::vector as variable length array is not allowed in most compilers)
00809     std::vector<std::vector<double>> x(n + 1, std::vector<double>(n + 1, 0.0));
00810
00811     for (int i = 0; i <= n; i++) {
00812         for (int j = 0; j <= n; j++) {
00813
00814             // Base cases
00815             if (i == 0 && j == 0) {
00816                 x[0][0] = 1;
00817             } else if (i == n && j == n) {
00818                 x[n][n] = 0;
00819
00820             // Fill in matrix
00821             } else if ((i != 0 && j == 0) || (i == n && j != n)) {
00822                 x[i][j] = x[i - 1][j] * aProb[i + j];
00823             } else if ((i == 0 && j != 0) || (i != n && j == n)) {
00824                 x[i][j] = x[i][j - 1] * (1 - aProb[i + j]);
00825             } else {
00826                 x[i][j] = x[i - 1][j] * aProb[i + j] + x[i][j - 1] * (1 - aProb[i + j]);
00827             }
00828         }
00829     }
00830
00831     return x;
00832 }
00833
00834 #endif

```

4.2 /Users/jace/Documents/GitHub/CPP-Structs-Algorithms/datastructs.hpp

```

00001
00011
00012 #ifndef DATASTRUCTS_H
00013 #define DATASTRUCTS_H
00014
00015 #include <iostream>
00016
00022 template <typename elmtype> class CircularDynamicArray {
00023
00024     public:
00030     CircularDynamicArray() {
00031         cap = 2;
00032         size = 0;

```

```

00033         start = 0;
00034         end = -1;
00035         info = new elmttype[2];
00036     }
00037
00045     CircularDynamicArray(int capacity) {
00046         cap = capacity;
00047         size = capacity;
00048         start = 0;
00049         end = capacity - 1;
00050         info = new elmttype[capacity];
00051     };
00052
00060     CircularDynamicArray(CircularDynamicArray const &src) {
00061         cap = src.cap;
00062         size = src.size;
00063         start = src.start;
00064         end = src.end;
00065         elmttype *tempInfo = new elmttype[cap];
00066         for (int i = 0; i < cap; i++) {
00067             tempInfo[i] = src.info[i];
00068         }
00069         info = tempInfo;
00070     }
00071
00077     ~CircularDynamicArray() { if (info != nullptr) {delete[] info;} };
00078
00082     int length() { return size; }
00083
00087     int capacity() { return cap; }
00088
00092     elmttype atRef(int i) {
00093         if (i < 0 || i > cap) {
00094             return burner;
00095         }
00096         return info[(i + start) % cap];
00097     }
00098
00102     elmttype *atPoint(int i) {
00103         if (i < 0 || i > cap) {
00104             return &burner;
00105         }
00106         return &info[(i + start) % cap];
00107     }
00108
00112     elmttype &operator[](int i) {
00113         if (i < 0 || i > cap) {
00114             return burner;
00115         }
00116         return info[(i + start) % cap];
00117     }
00118
00122     CircularDynamicArray &operator=(const CircularDynamicArray &R) {
00123         cap = R.cap;
00124         size = R.size;
00125         start = R.start;
00126         end = R.end;
00127         delete[] info;
00128         info = new elmttype[cap];
00129         for (int i = 0; i < cap; i++) {
00130             info[i] = R.info[i];
00131         }
00132         return *this;
00133     }
00134
00140     void addFront(elmttype v) {
00141         checkCapIncrease();
00142         if (start > 0) {
00143             info[start - 1] = v;
00144         } else {
00145             info[cap - 1] = v;
00146             start = cap;
00147         }
00148         start--;
00149         size++;
00150     };
00151
00157     void addEnd(elmttype v) {
00158         checkCapIncrease();
00159         info[(end + 1) % cap] = v;
00160         end++;
00161         end %= cap;
00162         size++;
00163     }
00164
00170     void delFront() {
00171         start++;

```

```

00172         start %= cap;
00173         size--;
00174         checkCapDecrease();
00175     };
00176
00182     void delEnd() {
00183         end--;
00184         if (end == -1) {
00185             end = cap - 1;
00186         }
00187         size--;
00188         checkCapDecrease();
00189     };
00190
00194     void clear() {
00195         size = 0;
00196         cap = 2;
00197         start = 0;
00198         end = -1;
00199         delete[] info;
00200         info = new elmttype[cap];
00201     };
00202
00206     void swap(elmttype *a, elmttype *b) {
00207         elmttype temp = (*a);
00208         (*a) = (*b);
00209         (*b) = temp;
00210     }
00211
00221     elmttype QuickSelect(int k) {
00222         if (k <= 0 || k > size) {
00223             return burner;
00224         }
00225         return select(k, standard);
00226     }
00227
00237     elmttype WSelect(int k) {
00238         if (k <= 0 || k > size) {
00239             return burner;
00240         }
00241         return select(k, worstCase);
00242     }
00243
00249     void stableSort() { mergeSort(0, size - 1); };
00250
00260     int linearSearch(elmttype e) {
00261         for (int i = 0; i < size; i++) {
00262             if (atRef(i) == e) return i;
00263         }
00264         return -1;
00265     };
00266
00276     int binSearch(elmttype e) {
00277         for (int m = size / 2, l = 0, r = size; l <= r; m = (r + l) / 2) {
00278             if (e == atRef(m)) {
00279                 return m;
00280             } else if (e > atRef(m)) {
00281                 l = m + 1;
00282             } else {
00283                 r = m - 1;
00284             }
00285         }
00286         return -1;
00287     }
00288
00289 private:
00290     // Array Data
00291
00295     int cap;
00296
00300     int size;
00301
00305     int start;
00306
00310     int end;
00311
00315     elmttype *info;
00316
00320     elmttype burner;
00321
00328     enum searchType { standard, worstCase };
00329
00335     void checkCapDecrease() {
00336         if (((cap / 4) - 1) > size) {
00337             elmttype *newArr = new elmttype[cap / 2];
00338             for (int i = 0; i < size; i++) {
00339                 newArr[i] = info[start + i];

```

```

00340         }
00341         cap /= 2;
00342         start = 0;
00343         delete[] info;
00344         info = newArr;
00345
00346         if (size > 0) {
00347             end = size - 1;
00348         } else {
00349             end = 0;
00350         };
00351     }
00352     return;
00353 };
00354
00355 void checkCapIncrease() {
00356     if (size == cap) {
00357         elmttype *newArr = new elmttype[cap * 2];
00358         for (int i = 0; i < size; i++) {
00359             newArr[i] = info[(start + i) % cap];
00360         }
00361         delete[] this->info;
00362         info = newArr;
00363         cap *= 2;
00364         start = 0;
00365         end = size - 1;
00366     }
00367     return;
00368 };
00369
00370 void merge(int l, int m, int r) {
00371     int sub1 = m - l + 1, sub2 = r - m;
00372
00373     elmttype *linfo = new elmttype[sub1], *rinfo = new elmttype[sub2];
00374
00375     for (int i = 0; i < sub1; i++) {
00376         int tempIndex = (l + i + start) % cap;
00377         linfo[i] = info[tempIndex];
00378     }
00379     for (int j = 0; j < sub2; j++) {
00380         int tempIndex = (m + j + 1 + start) % cap;
00381         rinfo[j] = info[tempIndex];
00382     }
00383
00384     int indexL = 0, indexR = 0, indexM = l;
00385
00386     // Primary Merge Loop
00387     while (indexL < sub1 && indexR < sub2) {
00388         if (linfo[indexL] <= rinfo[indexR]) {
00389             info[(indexM + start) % cap] = linfo[indexL];
00390             indexL++;
00391         } else {
00392             info[(indexM + start) % cap] = rinfo[indexR];
00393             indexR++;
00394         }
00395         indexM++;
00396     }
00397
00398     // Merge Remaining Lefts
00399     while (indexL < sub1) {
00400         info[(indexM + start) % cap] = linfo[indexL];
00401         indexL++;
00402         indexM++;
00403     }
00404
00405     // Merge Remaining Rights
00406     while (indexR < sub2) {
00407         info[(indexM + start) % cap] = rinfo[indexR];
00408         indexR++;
00409         indexM++;
00410     }
00411
00412     // Clean Up and Return
00413     delete[] linfo;
00414     delete[] rinfo;
00415     return;
00416 };
00417
00418 void mergeSort(int l, int r) {
00419     if (l >= r) {
00420         return;
00421     }
00422
00423     int m = l + (r - l) / 2;
00424     mergeSort(l, m);
00425     mergeSort(m + 1, r);
00426     merge(l, m, r);

```



```

00449     };
00450
00461     elmttype select(int k, searchType type) {
00462         elmttype tempArr[size];
00463         for (int i = 0; i < size; i++) {
00464             tempArr[i] = atRef(i);
00465         }
00466
00467         // Selects the correct function based on the type
00468         if (type == standard) {
00469             return KthSmallest(tempArr, 0, size - 1, k);
00470         }
00471         return KthSmallestWC(tempArr, 0, size - 1, k);
00472     }
00473
00474
00487     elmttype KthSmallest(elmttype *arr, int l, int r, int k) {
00488         elmttype pivot = arr[(l + r) / 2];
00489         int pos = partition(arr, l, r, pivot);
00490
00491         if (pos - l == k - 1) {
00492             return arr[pos];
00493         } else if (pos - l > k - 1) {
00494             return KthSmallest(arr, l, pos - 1, k);
00495         } else {
00496             return KthSmallest(arr, pos + 1, r, k - pos + l - 1);
00497         }
00498     }
00499
00512     elmttype KthSmallestWC(elmttype *arr, int l, int r, int k) {
00513         int n = r - l + 1, i;
00514         elmttype median[(n + 4) / 5];
00515         for (i = 0; i < n / 5; i++) {
00516             median[i] = getMedian(arr + l + i * 5, 5);
00517         }
00518         if (n > i * 5) {
00519             median[i] = getMedian(arr + l + i * 5, n % 5);
00520             i++;
00521         }
00522         elmttype medOfMed =
00523             (i == 1) ? median[i - 1] : KthSmallestWC(median, 0, i - 1, i / 2);
00524         int pos = partition(arr, l, r, medOfMed);
00525
00526         if (pos - l == k - 1) {
00527             return arr[pos];
00528         } else if (pos - l > k - 1) {
00529             return KthSmallestWC(arr, l, pos - 1, k);
00530         } else {
00531             return KthSmallestWC(arr, pos + 1, r, k - pos + l - 1);
00532         }
00533     }
00534
00540     elmttype getMedian(elmttype *arr, int n) {
00541         sort(arr, n);
00542         return arr[n / 2];
00543     }
00544
00557     int partition(elmttype arr[], int l, int r, elmttype pivot) {
00558         int i;
00559         for (i = l; i < r; i++) {
00560             if (arr[i] == pivot) {
00561                 break;
00562             }
00563         }
00564         swap(&arr[i], &arr[r]);
00565         i = l;
00566         for (int j = l; j <= r - 1; j++) {
00567             if (arr[j] <= pivot) {
00568                 swap(&arr[i], &arr[j]);
00569                 i++;
00570             }
00571         }
00572         swap(&arr[i], &arr[r]);
00573         return i;
00574     }
00575
00581     void sort(elmttype *arr, int n) {
00582         for (int i = 0; i < n; i++) {
00583             elmttype *smallest = &arr[0];
00584             for (int j = i + 1; j < n; j++) {
00585                 if (arr[j] < (*smallest)) {
00586                     smallest = &arr[j];
00587                 }
00588             }
00589             if (&arr[i] != smallest) {
00590                 swap(&arr[i], smallest);
00591             }

```

```

00592     }
00593 }
00594 };
00595
00601 template<typename keyType> class Heap {
00602
00603     public:
00604
00608     Heap() { info = new CircularDynamicArray<keyType>; }
00609
00613     Heap(keyType K[], int s) {
00614         info = new CircularDynamicArray<keyType>(s);
00615
00616         for (int i = 0; i < s; i++) {
00617             (*info)[i] = K[i];
00618         }
00619
00620         heapify();
00621     }
00622
00626     ~Heap() {
00627         // The CDA is deleted in its cleanup, deleting here will segfault
00628     }
00629
00633     int size() {
00634         return info->length();
00635     }
00636
00644     keyType peekKey() {
00645         return (*info)[0];
00646     }
00647
00655     void insert(keyType k) {
00656         info->addEnd(k);
00657         siftUp(info->length() - 1);
00658     }
00659
00665     void printKeys(std::ostream &out = std::cout) {
00666         for (int i = 0; i < info->length(); i++) {
00667             out << (*info)[i];
00668             if (i != info->length() - 1) {
00669                 out << " ";
00670             }
00671         }
00672         if (info->length() != 0) {out << std::endl;}
00673     }
00674
00682     keyType extractMin() {
00683         keyType min = (*info)[0];
00684         (*info)[0] = (*info)[info->length() - 1];
00685         info->delEnd();
00686         siftDown(0);
00687         return min;
00688     }
00689
00690     private:
00691
00695     CircularDynamicArray<keyType> *info;
00696
00702     void heapify() {
00703         for (int i = info->length() / 2; i >= 0; i--) {
00704             siftDown(i);
00705         }
00706     }
00707
00715     void siftDown(int i) {
00716         int l = lIndex(i), r = rIndex(i), min = i;
00717
00718         if (l < info->length() && (*info)[l] < (*info)[i]) { min = l; }
00719         if (r < info->length() && (*info)[r] < (*info)[min]) { min = r; }
00720         if (min != i) { swap(i, min); siftDown(min); }
00721     }
00722
00730     void siftUp(int i) {
00731         for (; i != 0 && (*info)[i] < (*info)[pIndex(i)]; i = pIndex(i)) {
00732             swap(i, pIndex(i));
00733         }
00734     }
00735
00744     void swap(int a, int b) {
00745         keyType temp = (*info)[a];
00746         (*info)[a] = (*info)[b];
00747         (*info)[b] = temp;
00748     }
00749
00759     int pIndex(int i) {
00760         return (i-1) / 2;

```

```

00761     }
00762
00772     int lIndex(int i) {
00773         return (i*2) + 1;
00774     }
00775
00785     int rIndex(int i) {
00786         return (i*2) + 2;
00787     }
00788 };
00789
00793 enum color {Red, Black};
00794
00804 template<typename keyType, typename valueType> class RBNode {
00805
00806     public:
00810     keyType *key;
00811
00815     valueType *val;
00816
00820     RBNode *l;
00821
00825     RBNode *r;
00826
00830     RBNode *p;
00831
00835     color c;
00836
00840     int size;
00841
00847     RBNode() {
00848         key = new keyType;
00849         val = new valueType;
00850         l = nullptr;
00851         r = nullptr;
00852         p = nullptr;
00853         size = 0;
00854         c = Red;
00855     }
00856
00865     RBNode(keyType k, valueType v) {
00866         key = new keyType; *key = k;
00867         val = new valueType; *val = v;
00868         l = nullptr;
00869         r = nullptr;
00870         p = nullptr;
00871         size = 1;
00872         c = Red;
00873     }
00874
00882     RBNode(bool nilCon) {
00883         l = nullptr;
00884         r = nullptr;
00885         p = nullptr;
00886         key = nullptr;
00887         val = nullptr;
00888         size = 0;
00889         c = Black;
00890     }
00891
00901     RBNode (keyType k, valueType v, color setc, int s, RBNode<keyType, valueType> *parent) {
00902         key = new keyType(k);
00903         val = new valueType(v);
00904         l = nullptr;
00905         r = nullptr;
00906         p = parent;
00907         c = setc;
00908         size = s;
00909     }
00910
00914     RBNode(const RBNode<keyType, valueType> &src) {
00915         *key = *src.key;
00916         *val = *src.val;
00917         l = src.l;
00918         r = src.r;
00919         p = src.p;
00920         c = src.c;
00921         size = src.size;
00922     }
00923
00929     ~RBNode() {
00930         if (key != nullptr) {delete key;}
00931         if (val != nullptr) {delete val;}
00932     }
00933
00941     void cascade(RBNode *nil) {
00942         if (this == nil) {return;}

```

```

00943         if (l != nil && l != nullptr) {
00944             l->cascade(nil);
00945         }
00946         if (r != nil && r != nullptr) {
00947             r->cascade(nil);
00948         }
00949         delete this;
00950     }
00951
00952     RBNode &operator=(RBNode R) {
00953         if (this == R) {
00954             return *this;
00955         }
00956         delete key; key = new keyType(R.key);
00957         delete val; val = new valueType(R.val);
00958         c = R.c;
00959         return *this;
00960     }
00961
00962     void preorder (std::ostream &out = std::cout) {
00963         if (key == nullptr) {
00964             return;
00965         }
00966         printNode(out);
00967         if (l->key != nullptr) {out << " "; l->preorder(out);}
00968         if (r->key != nullptr) {out << " "; r->preorder(out);}
00969     }
00970
00971     void inorder (std::ostream &out = std::cout) {
00972         if (key == nullptr) {
00973             return;
00974         }
00975         if (l->key != nullptr) {l->inorder(out); out << " ";}
00976         printNode(out);
00977         if (r->key != nullptr) {out << " "; r->inorder(out);}
00978     }
00979
00980     void postorder (std::ostream &out = std::cout) {
00981         if (key == nullptr) {
00982             return;
00983         }
00984         if (l->key != nullptr) {l->postorder(out); out << " ";}
00985         if (r->key != nullptr) {r->postorder(out); out << " ";}
00986         printNode(out);
00987     }
00988
00989     void printNode(std::ostream &out = std::cout) {
00990         if (key == nullptr) {
00991             return;
00992         }
00993         out << *key;
00994     }
00995
00996     void printk(int &k, std::ostream &out = std::cout) {
00997         if (l->key != nullptr) {
00998             l->printk(k, out);
00999             if (k > 0) {
01000                 out << " ";
01001             }
01002         }
01003         if (k < 1) {
01004             return;
01005         }
01006         out << *key;
01007         k--;
01008         if (k < 1) {
01009             return;
01010         }
01011         if (r->key != nullptr) {
01012             out << " ";
01013             r->printk(k, out);
01014         }
01015     }
01016
01017     valueType *searchValue(keyType k) {
01018         if (key == nullptr) {
01019             return nullptr;
01020         }
01021         if (k == *key) {
01022             return val;
01023         }
01024         if (k < *key && l != nullptr) {
01025             return l->searchValue(k);
01026         }

```

```

01069
01070     if (k > *key && r != nullptr) {
01071         return r->searchValue(k);
01072     }
01073
01074     return nullptr;
01075 }
01076
01086 RNode<keyType, valueType> *searchNode(keyType k) {
01087     if (k == *key) {
01088         return this;
01089     } else if (k < *key && l->key != nullptr) {
01090         return l->searchNode(k);
01091     } else if (k > *key && r->key != nullptr) {
01092         return r->searchNode(k);
01093     } else {
01094         return nullptr;
01095     }
01096 }
01097
01105 RNode<keyType, valueType> *predecessor() {
01106     if (l == nullptr) {
01107         return nullptr;
01108     }
01109     RNode *curr;
01110     for (curr = l; curr->l != nullptr; curr = curr->l) {continue;}
01111     return curr;
01112 }
01113
01121 RNode<keyType, valueType> *min() {
01122     if (l == nullptr) {
01123         return this;
01124     }
01125     RNode *curr;
01126     for (curr = l; curr->l != nullptr; curr = curr->l) {continue;}
01127     return curr;
01128 }
01129
01137 keyType select(int k) {
01138     if (k == l->size + 1) {
01139         return (*key);
01140     } else if (k <= l->size) {
01141         return l->select(k);
01142     } else {
01143         return r->select(k - l->size - 1);
01144     }
01145 }
01146 };
01147
01166 template<typename keyType, typename valueType> class RBTree {
01167 public:
01168
01172 RBTree() {
01173     nil = new RNode<keyType, valueType>(true);
01174     root = nil;
01175 }
01176
01185 RBTree(keyType k, valueType v) {
01186     nil = new RNode<keyType, valueType>(true);
01187     root = new RNode<keyType, valueType>(k, v);
01188     root->c = Black; // root is black
01189 }
01190
01202 RBTree(keyType *k, valueType *v, int s) {
01203     nil = new RNode<keyType, valueType>(true); // nil node's special constructor
01204     root = nil;
01205     for (int i = 0; i < s; i++) {
01206         insert(k[i], v[i]);
01207     }
01208 }
01209
01214 RBTree(const RBTree<keyType, valueType> &src) {
01215     nil = new RNode<keyType, valueType>(true);
01216     root = nil;
01217     copy(src.nil, src.root, root, nil);
01218 }
01219
01225 ~RBTree() {
01226     root->cascade(nil);
01227 }
01228
01238 RBTree<keyType, valueType> &operator=(const RBTree<keyType, valueType> &R) {
01239     if (this == &R) {
01240         return *this;
01241     }
01242
01243     root->cascade(nil);

```

```

01244     nil = new RBNode<keyType, valueType>(true);
01245     root = nil;
01246     copy(&R.nil, *&R.root, root, nil);
01247     return *this;
01248 }
01249
01255 void preorder (std::ostream &out) {
01256     root->preorder(out);
01257     out << std::endl;
01258 }
01259
01265 void inorder (std::ostream &out) {
01266     root->inorder(out);
01267     out << std::endl;
01268 }
01269
01275 void postorder (std::ostream &out) {
01276     root->postorder(out);
01277     out << std::endl;
01278 }
01279
01285 void printk (int k, std::ostream &out) {
01286     root->printk(k, out);
01287     out << std::endl;
01288 }
01289
01295 int size() {
01296     return root->size;
01297 }
01298
01308 valueType *search (keyType k) {
01309     return root->searchValue(k);
01310 }
01311
01320 void insert(keyType k, valueType v) {
01321     RBNode<keyType, valueType> *z = new RBNode<keyType, valueType>(k, v);
01322     z->size = 1;
01323     z->l = nil; z->r = nil;
01324     RBNode<keyType, valueType> *y = nil;
01325     RBNode<keyType, valueType> *x = root;
01326
01327     while (x != nil) {
01328         (x->size)++;
01329
01330         y = x;
01331
01332         if (*(z->key) < *(x->key)) {
01333             x = x->l;
01334         } else {
01335             x = x->r;
01336         }
01337     }
01338
01339     z->p = y;
01340     if (y == nil) {
01341         root = z;
01342     } else if (*(z->key) < *(y->key)) {
01343         y->l = z;
01344     } else {
01345         y->r = z;
01346     }
01347
01348     z->l = nil;
01349     z->r = nil;
01350     insertFixTree(z);
01351 }
01352 // Removes the node with key k from the tree.
01353 // Time complexity: O()
01354
01364 int remove(keyType k) {
01365     RBNode<keyType, valueType> *z = root->searchNode(k);
01366     if (z == nullptr) {
01367         return 0;
01368     }
01369     RBNode<keyType, valueType> *y = z;
01370     RBNode<keyType, valueType> *x;
01371
01372     for (RBNode<keyType, valueType> *i = z->p; i != nil; i = i->p) {
01373         i->size--;
01374     }
01375
01376     color yOrigcolor = y->c;
01377
01378     if (z->l == nil) {
01379         x = z->r;
01380         transplant(z, z->r);
01381     } else if (z->r == nil) {

```

```

01382         x = z->l;
01383         transplant(z, z->l);
01384     } else {
01385         y = max(z->l);
01386         yOrigcolor = y->c;
01387         x = y->l;
01388         if (y->p == z) {
01389             x->p = y;
01390         } else {
01391             transplant(y, y->l);
01392             y->l = z->l;
01393             y->l->p = y;
01394             for (RBNode<keyType, valueType> *i = x->p; i != nil && i != y; i = i->p) {
01395                 i->size--;
01396             }
01397         }
01398         transplant(z, y);
01399         y->r = z->r;
01400         y->r->p = y;
01401         y->c = z->c;
01402         y->size = y->l->size + y->r->size + 1;
01403     }
01404
01405     if (yOrigcolor == Black) {
01406         deleteFixTree(x);
01407     }
01408
01409     return l;
01410 }
01411
01412 int rank(keyType k) {
01413     RBNode<keyType, valueType> *node = root->searchNode(k);
01414     if (node == nullptr) {
01415         return 0;
01416     }
01417     int rank = node->l->size + 1;
01418
01419     for (RBNode<keyType, valueType> *curr = node; curr != root; curr = curr->p) {
01420         if (curr == curr->p->r) {
01421             rank += curr->p->l->size + 1;
01422         }
01423     }
01424     return rank;
01425 }
01426
01427 keyType select(int k) {
01428     return root->select(k);
01429 }
01430
01431 keyType *successor(keyType k) {
01432     RBNode<keyType, valueType> *curr = root->searchNode(k);
01433     if (curr == nullptr) {
01434         return nullptr;
01435     }
01436
01437     if (curr->r != nil) {
01438         for (curr = curr->r; curr->l != nil; curr = curr->l) {continue;}
01439         return curr->key;
01440     } else {
01441         RBNode<keyType, valueType> *i;
01442         for (i = curr->p; i != nil && curr == i->r; i = i->p) {curr = i;}
01443         return i->key;
01444     }
01445 }
01446
01447 keyType *predecessor(keyType k) {
01448     RBNode<keyType, valueType> *curr = root->searchNode(k);
01449     if (curr == nullptr) {
01450         return nullptr;
01451     }
01452
01453     if (curr->l != nil) {
01454         for (curr = curr->l; curr->r != nil; curr = curr->r) {continue;}
01455         return curr->key;
01456     } else {
01457         RBNode<keyType, valueType> *i;
01458         for (i = curr->p; i != nil && curr == i->l; i = i->p) {curr = i;}
01459         return i->key;
01460     }
01461 }
01462
01463 private:
01464
01465     RBNode<keyType, valueType> *root;
01466
01467     RBNode<keyType, valueType> *nil;

```

```

01509
01520     void copy(RBNode<keyType, valueType> *copiedNil, RBNode<keyType, valueType> *copiedNode,
RBNode<keyType, valueType> *&newNode, RBNode<keyType, valueType> *&newParent) {
01521         newNode = new RBNode<keyType, valueType>(*copiedNode->key, *copiedNode->val),
copiedNode->c, copiedNode->size, newParent);
01522
01523         if (copiedNode->l == copiedNil || copiedNode->l == nullptr) {
01524             newNode->l = nil;
01525         } else {
01526             copy(copiedNil, copiedNode->l, newNode->l, newNode);
01527         }
01528         if (copiedNode->r == copiedNil || copiedNode->r == nullptr) {
01529             newNode->r = nil;
01530         } else {
01531             copy(copiedNil, copiedNode->r, newNode->r, newNode);
01532         }
01533     }
01534
01546     RBNode<keyType, valueType> *min(RBNode<keyType, valueType> *node) {
01547         for (node; node->l != nil; node = node->l) {
01548             continue;
01549         }
01550         return node;
01551     }
01552
01564     RBNode<keyType, valueType> *max(RBNode<keyType, valueType> *node) {
01565         for (; node->r != nil; node = node->r) {
01566             continue;
01567         }
01568         return node;
01569     }
01570
01578     void insertFixTree(RBNode<keyType, valueType> *z) {
01579         RBNode<keyType, valueType> *y;
01580
01581         while (z->p->c == Red) {
01582             if (z->p == z->p->p->l) {
01583                 y = z->p->p->r;
01584
01585                 if (y->c == Red) {
01586                     z->p->c = Black;
01587                     y->c = Black;
01588                     z->p->p->c = Red;
01589                     z = z->p->p;
01590                 } else {
01591                     if (z == z->p->r) {
01592                         z = z->p;
01593                         lRotate(z);
01594                     }
01595
01596                     z->p->c = Black;
01597                     z->p->p->c = Red;
01598                     rRotate(z->p->p);
01599                 }
01600             } else {
01601                 y = z->p->p->l;
01602
01603                 if (y->c == Red) {
01604                     z->p->c = Black;
01605                     y->c = Black;
01606                     z->p->p->c = Red;
01607                     z = z->p->p;
01608                 } else {
01609                     if (z == z->p->l) {
01610                         z = z->p;
01611                         rRotate(z);
01612                     }
01613
01614                     z->p->c = Black;
01615                     z->p->p->c = Red;
01616                     lRotate(z->p->p);
01617                 }
01618             }
01619         }
01620         root->c = Black;
01621     }
01622
01630     void deleteFixTree(RBNode<keyType, valueType> *x) {
01631
01632         RBNode<keyType, valueType> *w;
01633         while (x != root && x->c == Black) {
01634             if (x == x->p->l) {
01635                 w = x->p->r;
01636                 if (w->c == Red) {
01637                     w->c = Black;
01638                     x->p->c = Red;
01639                     lRotate(x->p);

```



```

01640         w = x->p->r;
01641     }
01642
01643     if (w->l->c == Black && w->r->c == Black) {
01644         w->c = Red;
01645         x = x->p;
01646     } else {
01647         if (w->r->c == Black) {
01648             w->l->c = Black;
01649             w->c = Red;
01650             rRotate(w);
01651             w = x->p->r;
01652         }
01653
01654         w->c = x->p->c;
01655         x->p->c = Black;
01656         w->r->c = Black;
01657         lRotate(x->p);
01658         x = root;
01659     }
01660 } else {
01661     w = x->p->l;
01662     if (w->c == Red) {
01663         w->c = Black;
01664         x->p->c = Red;
01665         rRotate(x->p);
01666         w = x->p->l;
01667     }
01668
01669     if (w->r->c == Black && w->l->c == Black) {
01670         w->c = Red;
01671         x = x->p;
01672     } else {
01673         if (w->l->c == Black) {
01674             w->r->c = Black;
01675             w->c = Red;
01676             lRotate(w);
01677             w = x->p->l;
01678         }
01679
01680         w->c = x->p->c;
01681         x->p->c = Black;
01682         w->l->c = Black;
01683         rRotate(x->p);
01684         x = root;
01685     }
01686 }
01687 }
01688 }
01689
01690 void rRotate(RBNode<keyType, valueType> *x) {
01691     RBNode<keyType, valueType> *y = x->l;
01692     x->l = y->r;
01693     if (y->r != nil) {
01694         y->r->p = x;
01695     }
01696     y->p = x->p;
01697     if (x->p == nil) {
01698         root = y;
01699     } else if (x == x->p->r) {
01700         x->p->r = y;
01701     } else {
01702         x->p->l = y;
01703     }
01704     y->r = x;
01705     x->p = y;
01706     y->size = x->size;
01707     x->size = x->l->size + x->r->size + 1;
01708 }
01709
01710 void lRotate(RBNode<keyType, valueType> *x) {
01711     RBNode<keyType, valueType> *y = x->r;
01712     x->r = y->l;
01713     if (y->l != nil) {
01714         y->l->p = x;
01715     }
01716     y->p = x->p;
01717     if (x->p == nil) {
01718         root = y;
01719     } else if (x == x->p->l) {
01720         x->p->l = y;
01721     } else {
01722         x->p->r = y;
01723     }
01724     y->l = x;
01725     x->p = y;
01726     y->size = x->size;
01727 }

```

```
01741     x->size = x->l->size + x->r->size + 1;
01742 }
01743
01750 void transplant(RBNode<keyType, valueType> *u, RBNode<keyType, valueType> *v) {
01751
01752     // If u is the root, set the root to v
01753     if (u->p == nil) {
01754         root = v;
01755
01756         // If u is the left child of its parent, set the left child of its parent to v
01757     } else if (u == u->p->l) {
01758         u->p->l = v;
01759
01760         // If u is the right child of its parent, set the right child of its parent to v
01761     } else {
01762         u->p->r = v;
01763     }
01764
01765     // Set the parent of v to the parent of u
01766     v->p = u->p;
01767 }
01768 };
01769 #endif
```

Index

- ~CircularDynamicArray
 - CircularDynamicArray< elmtyp> , 9
- ~LCS
 - LCS, 16
- ~RBNode
 - RBNode< keyType, valueType> , 21
- ~RBTre
 - RBTre< keyType, valueType> , 27
- addEdge
 - APSP, 6
- addEnd
 - CircularDynamicArray< elmtyp> , 9
- addFront
 - CircularDynamicArray< elmtyp> , 9
- APSP, 5
 - addEdge, 6
 - APSP, 5
 - getCost, 6
 - printAdjacency, 6
- binSearch
 - CircularDynamicArray< elmtyp> , 9
- cascade
 - RBNode< keyType, valueType> , 22
- CircularDynamicArray
 - CircularDynamicArray< elmtyp> , 8
- CircularDynamicArray< elmtyp> , 7
 - ~CircularDynamicArray, 9
 - addEnd, 9
 - addFront, 9
 - binSearch, 9
 - CircularDynamicArray, 8
 - delEnd, 9
 - delFront, 10
 - linearSearch, 10
 - QuickSelect, 10
 - stableSort, 10
 - WCSelect, 11
- delEnd
 - CircularDynamicArray< elmtyp> , 9
- delFront
 - CircularDynamicArray< elmtyp> , 10
- extractMin
 - Heap< keyType> , 13
- Fib, 11
 - Fib, 12
- print, 12
- printAll, 12
- get
 - LCS, 17
- getCost
 - APSP, 6
- Heap< keyType> , 13
 - extractMin, 13
 - insert, 13
 - peekKey, 14
 - printKeys, 14
- inorder
 - RBNode< keyType, valueType> , 22
 - RBTre< keyType, valueType> , 27
- insert
 - Heap< keyType> , 13
 - RBTre< keyType, valueType> , 27
- Knapsack, 14
 - Knapsack, 15
- LCS, 15
 - ~LCS, 16
 - get, 17
 - LCS, 16
 - newStrings, 17
- linearSearch
 - CircularDynamicArray< elmtyp> , 10
- Matrix, 17
 - print, 18
- MatrixChain, 18
- min
 - RBNode< keyType, valueType> , 22
- newStrings
 - LCS, 17
- operator=
 - RBTre< keyType, valueType> , 28
- peekKey
 - Heap< keyType> , 14
- postorder
 - RBNode< keyType, valueType> , 22
 - RBTre< keyType, valueType> , 28
- predecessor
 - RBNode< keyType, valueType> , 23

- RBTree< keyType, valueType >, 28
- preorder
 - RBNode< keyType, valueType >, 23
 - RBTree< keyType, valueType >, 28
- print
 - Fib, 12
 - Matrix, 18
- printAdjacency
 - APSP, 6
- printAll
 - Fib, 12
- printk
 - RBNode< keyType, valueType >, 23
 - RBTree< keyType, valueType >, 29
- printKeys
 - Heap< keyType >, 14
- printNode
 - RBNode< keyType, valueType >, 23
- QuickSelect
 - CircularDynamicArray< elmtype >, 10
- rank
 - RBTree< keyType, valueType >, 29
- RBNode
 - RBNode< keyType, valueType >, 20, 21
- RBNode< keyType, valueType >, 19
 - ~RBNode, 21
 - cascade, 22
 - inorder, 22
 - min, 22
 - postorder, 22
 - predecessor, 23
 - preorder, 23
 - printk, 23
 - printNode, 23
 - RBNode, 20, 21
 - searchNode, 24
 - searchValue, 24
 - select, 24
- RBTree
 - RBTree< keyType, valueType >, 26
- RBTree< keyType, valueType >, 25
 - ~RBTree, 27
 - inorder, 27
 - insert, 27
 - operator=, 28
 - postorder, 28
 - predecessor, 28
 - preorder, 28
 - printk, 29
 - rank, 29
 - RBTree, 26
 - remove, 29
 - search, 29
 - select, 31
 - size, 31
 - successor, 31
- remove
 - RBTree< keyType, valueType >, 29
- search
 - RBTree< keyType, valueType >, 29
- searchNode
 - RBNode< keyType, valueType >, 24
- searchValue
 - RBNode< keyType, valueType >, 24
- select
 - RBNode< keyType, valueType >, 24
 - RBTree< keyType, valueType >, 31
- size
 - RBTree< keyType, valueType >, 31
- stableSort
 - CircularDynamicArray< elmtype >, 10
- successor
 - RBTree< keyType, valueType >, 31
- WCSelect
 - CircularDynamicArray< elmtype >, 11