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

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

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

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

n	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	The starting vertex (0-indexed)
j	The ending vertex (0-indexed)
cost	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	The starting vertex (0-indexed)
j	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

out	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< elmtype > 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.
- elmtype **atRef** (int i)
  Returns the value of the element at index i.
- elmtype ∗ **atPoint** (int i)
  Returns a pointer to the element at index i.
- elmtype & **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 (elmtype v)
  Adds an element to the front of the array.
- void addEnd (elmtype 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** (elmtype ∗a, elmtype ∗b)
  Swaps two elements.
- elmtype QuickSelect (int k)
  Returns the Kth smallest element of the array using the median as the partition element.
- elmtype 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 (elmtype e)
  Performs a linear search for element e.
- int binSearch (elmtype e)
  Performs a binary search for element e.

3.2.1 Detailed Description

template< typename elmtype >
class CircularDynamicArray< elmtype >

Implements a circular dynamic array that can dynamically resize itself.

Template Parameters

elmtype	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 elmtype>
CircularDynamicArray< elmtype >::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 elmtype>
CircularDynamicArray< elmtype >::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

capacity	The capacity of the circular dynamic array

3.2.2.3 CircularDynamicArray() [3/3]

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

Copy constructor (deep copy)

Deep copies all values from the source circular dynamic array

Parameters

src	The source circular dynamic array to copy

3.2.2.4 ∼CircularDynamicArray()

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

Destructor.

Deletes the array data

3.2.3 Member Function Documentation

3.2.3.1 addEnd()

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

Adds an element to the end of the array.

Time complexity: O(1) (Amortized)

3.2.3.2 addFront()

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

Adds an element to the front of the array.

Time complexity: O(1) (Amortized)

3.2.3.3 binSearch()

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

Performs a binary search for element e.

Time complexity: O(lg(size))

Parameters

e	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(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(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 elmtype>
void CircularDynamicArray< elmtype >::stableSort ( ) [inline]
```

Performs a mergesort on the array.

Time complexity: O(size ∗ lg(size))

3.2.3.9 WCSelect()

```
template<typename elmtype>
elmtype CircularDynamicArray< elmtype >::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(size)

Parameters

k	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

n	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

n	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

keyType	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

k	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

out	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

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

a	First std::string for which to find the LCS
b	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

a	New first std::string for which to find the LCS
b	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

out	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

keyType	The type of key stored in the node
valueType	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

k	The key of the node
v	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

nilCon	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

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

nil	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

out	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(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

out	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(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

out	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(size))

Parameters

k	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

out	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(size))

Parameters

k	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(size))

Parameters

k	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(size))

Parameters

k	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

keyType	The type of key stored in the tree
valueType	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

k	The key of the root node
v	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

k	The array of keys to insert
v	The array of values to insert
s	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(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(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(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(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(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(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 * RBTree< 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(size))

Parameters

k	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

k	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(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

```


#ifndef ALGORITHMS_H
#define ALGORITHMS_H

#include <iostream>
#include <climits>
#include <vector>
#include <ctime>

class APSP {

    std::vector< std::vector<int> > adjacency;

    int n;

    bool built;

    void build() {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    if (adjacency[j][i] == INT_MAX || adjacency[i][k] == INT_MAX) {
                        continue;
                    }
                    if (adjacency[j][i] + adjacency[i][k] < adjacency[j][k]) {
                        adjacency[j][k] = adjacency[j][i] + adjacency[i][k];
                    }
                }
            }
        }

        // After table is built, check for negative cycles and set built to true
        checkNegativeCycle();
        built = true;
    }

    void checkNegativeCycle() {
        for (int i = 0; i < n; i++) {
            if (adjacency[i][i] < 0) {
                throw std::runtime_error("Negative cycle detected at vertex " + std::to_string(i) + "\n");
            }
        }
    }

    public:

    APSP(int n) {
        this->n = n;
        built = false;
        for (int i = 0; i < n; i++) {
            std::vector<int> row;
            for (int j = 0; j < n; j++) {
                if (i == j) {
```

```
                    row.push_back(0);
                    continue;
                }
                row.push_back(INT_MAX);
            }
            adjacency.push_back(row);
        }
    }

    void addEdge(int i, int j, int cost) {
        adjacency[i][j] = cost;
        built = false;
    }

    int getCost(int i, int j) {
        if (!built) {
            build();
        }

        return adjacency[i][j];
    }

    void printAdjacency(std::ostream &out = std::cout) {
        if (!built) {
            build();
        }

        for (int i = 0; i < n; i++) {
            std::cout « i + 1 « ": ";
            for (int j = 0; j < n; j++) {
                if (adjacency[i][j] == INT_MAX) {
                    std::cout « "inf ";
                    continue;
                }
                std::cout « adjacency[i][j] « " ";
            }
            std::cout « std::endl;
        }
    }
};

class Fib {

    __uint128_t fibList[186];

    __uint128_t fib(int *n) {

        // Check if number is too large for __uint128_t
        if (*n > 185) {
            throw std::runtime_error("Fibonacci number too large for unsigned 128-bit int\n");
        }

        if (*n == 0) {
            return 0;
        }

        // Check if number is stored
        if (fibList[*n] != 0) {
            return fibList[*n];
        }

        // If number is not stored, calculate it
        int a = *n - 1;
        int b = *n - 2;
        fibList[*n] = fib(&a) + fib(&b);

        return fibList[*n];
    }

    public:

    Fib() {
        // Set first two fibonacci numbers and clear all other data in map
        fibList[0] = 0;
        fibList[1] = 1;
        for (int i = 2; i < 186; i++) {
            fibList[i] = 0;
        }
    }

    void print(int n, std::ostream &out = std::cout) {

        __uint128_t fibNum = fib(&n);
        if (fibNum == 0) {
            out « "fib(" « n « ") = 0\n";
            return;
        }
```

```
        std::string fibString;
        while (fibNum > 0) {
            fibString.insert(fibString.begin(), '0' + (fibNum % 10));
            fibNum /= 10;
        }

        out « "fib(" « n « ") = " « fibString « std::endl;
    }

    void printAll(int n, std::ostream &out = std::cout) {

        if (n > 185) {
            throw std::runtime_error("Cannot print Fibonacci numbers greater than 185 with unsigned 128-bit int\n");
        }

        // Print all fibonacci numbers up to n (inclusive)
        for (int i = 0; i < n; i++) {
            print(i, out);
        }
    }
};

class Knapsack {

    int n;

    int max;

    int *weights;

    int *profits;

    bool built;

    bool *objects;

    std::vector< std::vector<int> > table;

    void build() {

        // Table building loop
        for (int j = 0; j <= n; j++) {
            for (int k = 0; k <= max; k++) {
                if (j == 0) {
                    table[j][k] = 0;
                } else if (k < weights[j]) {
                    table[j][k] = table[j - 1][k];
                } else {
                    // max() function is not functioning properly, so I manually find the max
                    int
                        a = table[j - 1][k],
                        b = table[j - 1][k - weights[j]] + profits[j];
                    if (a > b) {
                        table[j][k] = a;
                    } else {
                        table[j][k] = b;
                    }
                }
            }
        }

        // After table is built, choose the objects and set built to true
        choose();
        built = true;
    }

    void choose() {

        // Start at the bottom right corner of the table and work backwards
        int k = max;
        for (int i = n; i > 0; i--) {
            if (table[i-1][k] == table[i][k]) {
                objects[i] = false;
            } else {
                objects[i] = true;
                k -= weights[i];
            }
        }
    }

    public:

    Knapsack(int n, int max, int *p, int *w) {

        this->n = n;
```

```
        this->max = max;
        built = false;
        objects = new bool[n+1];
        weights = w;
        profits = p;

        for (int i = 0; i <= n; i++) {
            std::vector<int> temp;
            for (int j = 0; j <= max; j++) {
                temp.push_back(0);
            }
            table.push_back(temp);
        }
    }

    void printTable() {
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= max; j++) {
                std::cout « table[i][j] « " ";
            }
            std::cout « std::endl;
        }
    }

    void printPWO(std::ostream &out = std::cout) {
        if (!built) {
            build();
        }
        out « "Profit: ";
        for (int i = 1; i <= n; i++) {
            out « profits[i] « " ";
        }
        out « std::endl;
        out « "Weight: ";
        for (int i = 1; i <= n; i++) {
            out « weights[i] « " ";
        }
        out « std::endl;
        out « "Objects Chosen: ";
        for (int i = 1; i <= n; i++) {
            out « objects[i] « " ";
        }
    }

    int getProfit() {
        if (!built) {
            build();
        }
        return table[n][max];
    }

};

class LCS {

    private:

        std::string s1;

        std::string s2;

        std::string foundLCS;

        std::vector< std::vector<int> > L;

        bool built;

        std::string build(int j, int k) {
            if (j == 0 || k == 0) {
                return "";
            }
            if (s1[j - 1] == s2[k - 1]) {
                return build(j - 1, k - 1) + s1[j - 1];
            }
            if (L[j][k - 1] >= L[j - 1][k]) {
                return build(j, k - 1);
            } else {
                return build(j - 1, k);
            }
        }

    public:

        LCS(std::string a, std::string b) : s1(std::move(a)), s2(std::move(b)), built(false) {}

        ~LCS() {
        }
```

```
        std::string get() {
            if (built) {
                return foundLCS;
            }

            int n = s1.length();
            int m = s2.length();
            L = std::vector<std::vector<int»(n + 1, std::vector<int>(m + 1, 0));

            for (int j = 1; j <= n; j++) {
                for (int k = 1; k <= m; k++) {
                    if (s1[j - 1] == s2[k - 1]) {
                        L[j][k] = L[j - 1][k - 1] + 1;
                    } else {
                        L[j][k] = std::max(L[j - 1][k], L[j][k - 1]);
                    }
                }
            }

            foundLCS = build(n, m);
            built = true;
            return foundLCS;
        }

        int lcsLength() {
            if (!built) {
                get();
            }
            return foundLCS.length();
        }

        void printMatrix() {

            // Ensure LCS is built before printing the matrix
            if (!built) {
                get();
            }

            std::cout « "Matrix L: " « std::endl;
            for (int i = 0; i <= s1.length(); i++) {
                for (int j = 0; j <= s2.length(); j++) {
                    std::cout « L[i][j] « " ";
                }
                std::cout « std::endl;
            }
        }

        void newStrings(std::string a, std::string b) {
            s1 = a;
            s2 = b;
            built = false;
            foundLCS = "";
            L.clear();
        }
};

class Matrix {
    public:

    int row;

    int col;

    std::vector< std::vector<int> > data;

    Matrix(std::vector< std::vector<int> > m): row(m.at(0).size()), col(m.size()), data(m) {};

    Matrix(int r, int c) : row(r), col(c), data(r, std::vector<int>(c, 0)) {}

    bool operator==(const Matrix &m) {
        if (row != m.row || col != m.col) {
            return false;
        }
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (data[i][j] != m.data[i][j]) {
                    return false;
                }
            }
        }
        return true;
    }

    void clear() {
        row = -1;
        col = -1;
```

```
        data.clear();
    }

    void print(std::ostream &out = std::cout) {
        for (int i = 0; i < col; i++) {
            for (int j = 0; j < row; j++) {
                out « data[i][j] « " ";
            }
            out « std::endl;
        }
    }
};

class MatrixChain {

    int n;

    bool calculated;

    Matrix *solution;

    Matrix *bestK;

    Matrix *cost;

    std::vector<int> dim;

    std::vector<Matrix*> data;

    void calcCosts() {
        for (int L = 1; L < n; L++) {
            for (int i = 0; i < n - L; i++) {
                int j = i + L;
                cost->data[i][j] = INT_MAX;
                for (int k = i; k < j; k++) {
                    int q = cost->data[i][k] + cost->data[k + 1][j] + dim[i] * dim[k + 1] * dim[j + 1];
                    if (q < cost->data[i][j]) {
                        cost->data[i][j] = q;
                        bestK->data[i][j] = k;
                    }
                }
            }
        }
    }

    Matrix *matrixMult(Matrix *a, Matrix *b) {
        Matrix *c = new Matrix(a->row, b->col);
        for (int i = 0; i < a->row; i++) {
            for (int j = 0; j < b->col; j++) {
                for (int k = 0; k < a->col; k++) {
                    c->data[i][j] += a->data[i][k] * b->data[k][j];
                }
            }
        }
        return c;
    }

    Matrix *chainProduct(int i, int j) {
        if (i == j) {
            return data[i];
        }
        int k = bestK->data[i][j];
        Matrix *a = chainProduct(i, k);
        Matrix *b = chainProduct(k + 1, j);
        return matrixMult(a, b);
    }

    public:

    MatrixChain(std::vector<Matrix*> m) {
        n = m.size();
        calculated = false;
        solution = new Matrix(n, n);
        bestK = new Matrix(n, n);
        cost = new Matrix(n, n);

        for (int i = 0; i < n; i++) {
            data.push_back(m[i]);
        }

        for (int i = 0; i < n; i++) {
            dim.push_back(m[i]->row);
            if (i == 0) {
                continue;
            }
            if (m[i-1]->col != m[i]->row) {
```

```
                throw std::runtime_error(
                    "Matrix dimensions do not match in MatrixChain constructor between matrices " + std::to_string(i-1) + " and " + std::to_string(i) + "\n" +
                    "Col size of matrix " + std::to_string(i-1) + " (" + std::to_string(m[i-1]->col) +
                    ") != Row size of matrix " + std::to_string(i) + " (" + std::to_string(m[i]->row) + ")\n"
                );
            }
        }
    }

    void addMatrix(Matrix *m) {
        if (m->row != data[n - 1]->col) {
            throw std::runtime_error("Matrix dimensions do not match in MatrixChain::addMatrix");
            return;
        }

        data.push_back(m);
        if (calculated) {
            calculated = false;
            delete solution;
        }
        n++;
    }

    Matrix *solve() {
        if (!calculated) {
            calcCosts();
            solution = chainProduct(0, n - 1);
        }
        return solution;
    }

};

std::vector<std::vector<double» worldSeries(int n, double *aProb) {

    // Create matrix (Use std::vector as variable length array is not allowed in most compilers)
    std::vector<std::vector<double» x(n + 1, std::vector<double>(n + 1, 0.0));

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {

            // Base cases
            if (i == 0 && j == 0) {
                x[0][0] = 1;
            } else if (i == n && j == n) {
                x[n][n] = 0;

            // Fill in matrix
            } else if ((i != 0 && j == 0) || (i == n && j != n)) {
                x[i][j] = x[i - 1][j] * aProb[i + j];
            } else if ((i == 0 && j != 0) || (i != n && j == n)) {
                x[i][j] = x[i][j - 1] * (1 - aProb[i + j]);
            } else {
                x[i][j] = x[i - 1][j] * aProb[i + j] + x[i][j - 1] * (1 - aProb[i + j]);
            }
        }
    }

    return x;
}

#endif
```

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

```


#ifndef DATASTRUCTS_H
#define DATASTRUCTS_H

#include <iostream>

template <typename elmtype> class CircularDynamicArray {

   public:
    CircularDynamicArray() {
        cap = 2;
        size = 0;
```

```
        start = 0;
        end = -1;
        info = new elmtype[2];
    }

    CircularDynamicArray(int capacity) {
        cap = capacity;
        size = capacity;
        start = 0;
        end = capacity - 1;
        info = new elmtype[capacity];
    };

    CircularDynamicArray(CircularDynamicArray const &src) {
        cap = src.cap;
        size = src.size;
        start = src.start;
        end = src.end;
        elmtype *tempInfo = new elmtype[cap];
        for (int i = 0; i < cap; i++) {
            tempInfo[i] = src.info[i];
        }
        info = tempInfo;
    }

    ~CircularDynamicArray() { if (info != nullptr) {delete[] info;} };

    int length() { return size; }

    int capacity() { return cap; }

    elmtype atRef(int i) {
        if (i < 0 || i > cap) {
            return burner;
        }
        return info[(i + start) % cap];
    }

    elmtype *atPoint(int i) {
        if (i < 0 || i > cap) {
            return &burner;
        }
        return &info[(i + start) % cap];
    }

    elmtype &operator[](int i) {
        if (i < 0 || i > cap) {
            return burner;
        }
        return info[(i + start) % cap];
    }

    CircularDynamicArray &operator=(const CircularDynamicArray &R) {
        cap = R.cap;
        size = R.size;
        start = R.start;
        end = R.end;
        delete[] info;
        info = new elmtype[cap];
        for (int i = 0; i < cap; i++) {
            info[i] = R.info[i];
        }
        return *this;
    }

    void addFront(elmtype v) {
        checkCapIncrease();
        if (start > 0) {
            info[start - 1] = v;
        } else {
            info[cap - 1] = v;
            start = cap;
        }
        start--;
        size++;
    };

    void addEnd(elmtype v) {
        checkCapIncrease();
        info[(end + 1) % cap] = v;
        end++;
        end %= cap;
        size++;
    }

    void delFront() {
        start++;
```

```
        start %= cap;
        size--;
        checkCapDecrease();
    };

    void delEnd() {
        end--;
        if (end == -1) {
            end = cap - 1;
        }
        size--;
        checkCapDecrease();
    };

    void clear() {
        size = 0;
        cap = 2;
        start = 0;
        end = -1;
        delete[] info;
        info = new elmtype[cap];
    };

    void swap(elmtype *a, elmtype *b) {
        elmtype temp = (*a);
        (*a) = (*b);
        (*b) = temp;
    }

    elmtype QuickSelect(int k) {
        if (k <= 0 || k > size) {
            return burner;
        }
        return select(k, standard);
    }

    elmtype WCSelect(int k) {
        if (k <= 0 || k > size) {
            return burner;
        }
        return select(k, worstCase);
    }

    void stableSort() { mergeSort(0, size - 1); };

    int linearSearch(elmtype e) {
        for (int i = 0; i < size; i++) {
            if (atRef(i) == e) return i;
        }
        return -1;
    };

    int binSearch(elmtype e) {
        for (int m = size / 2, l = 0, r = size; l <= r; m = (r + l) / 2) {
            if (e == atRef(m)) {
                return m;
            } else if (e > atRef(m)) {
                l = m + 1;
            } else {
                r = m - 1;
            }
        }
        return -1;
    }

   private:
    // Array Data

    int cap;

    int size;

    int start;

    int end;

    elmtype *info;

    elmtype burner;

    enum searchType { standard, worstCase };

    void checkCapDecrease() {
        if (((cap / 4) - 1) > size) {
            elmtype *newArr = new elmtype[cap / 2];
            for (int i = 0; i < size; i++) {
                newArr[i] = info[start + i];
```

```
            }
            cap /= 2;
            start = 0;
            delete[] info;
            info = newArr;

            if (size > 0) {
                end = size - 1;
            } else {
                end = 0;
            };
        }
        return;
    };

    void checkCapIncrease() {
        if (size == cap) {
            elmtype *newArr = new elmtype[cap * 2];
            for (int i = 0; i < size; i++) {
                newArr[i] = info[(start + i) % cap];
            }
            delete[] this->info;
            info = newArr;
            cap *= 2;
            start = 0;
            end = size - 1;
        }
        return;
    };

    void merge(int l, int m, int r) {
        int sub1 = m - l + 1, sub2 = r - m;

        elmtype *linfo = new elmtype[sub1], *rinfo = new elmtype[sub2];

        for (int i = 0; i < sub1; i++) {
            int tempIndex = (l + i + start) % cap;
            linfo[i] = info[tempIndex];
        }
        for (int j = 0; j < sub2; j++) {
            int tempIndex = (m + j + 1 + start) % cap;
            rinfo[j] = info[tempIndex];
        }

        int indexL = 0, indexR = 0, indexM = l;

        // Primary Merge Loop
        while (indexL < sub1 && indexR < sub2) {
            if (linfo[indexL] <= rinfo[indexR]) {
                info[(indexM + start) % cap] = linfo[indexL];
                indexL++;
            } else {
                info[(indexM + start) % cap] = rinfo[indexR];
                indexR++;
            }
            indexM++;
        }

        // Merge Remaining Lefts
        while (indexL < sub1) {
            info[(indexM + start) % cap] = linfo[indexL];
            indexL++;
            indexM++;
        }

        // Merge Remaining Rights
        while (indexR < sub2) {
            info[(indexM + start) % cap] = rinfo[indexR];
            indexR++;
            indexM++;
        }

        // Clean Up and Return
        delete[] linfo;
        delete[] rinfo;
        return;
    };

    void mergeSort(int l, int r) {
        if (l >= r) {
            return;
        }

        int m = l + (r - l) / 2;
        mergeSort(l, m);
        mergeSort(m + 1, r);
        merge(l, m, r);
```

```
    };

    elmtype select(int k, searchType type) {

        elmtype tempArr[size];
        for (int i = 0; i < size; i++) {
            tempArr[i] = atRef(i);
        }

        // Selects the correct function based on the type
        if (type == standard) {
            return KthSmallest(tempArr, 0, size - 1, k);
        }
        return KthSmallestWC(tempArr, 0, size - 1, k);
    }

    elmtype KthSmallest(elmtype *arr, int l, int r, int k) {
        elmtype pivot = arr[(l + r) / 2];
        int pos = partition(arr, l, r, pivot);

        if (pos - l == k - 1) {
            return arr[pos];
        } else if (pos - l > k - 1) {
            return KthSmallest(arr, l, pos - 1, k);
        } else {
            return KthSmallest(arr, pos + 1, r, k - pos + l - 1);
        }
    }

    elmtype KthSmallestWC(elmtype *arr, int l, int r, int k) {
        int n = r - l + 1, i;
        elmtype median[(n + 4) / 5];
        for (i = 0; i < n / 5; i++) {
            median[i] = getMedian(arr + l + i * 5, 5);
        }
        if (n > i * 5) {
            median[i] = getMedian(arr + l + i * 5, n % 5);
            i++;
        }
        elmtype medOfMed =
            (i == 1) ? median[i - 1] : KthSmallestWC(median, 0, i - 1, i / 2);
        int pos = partition(arr, l, r, medOfMed);

        if (pos - l == k - 1) {
            return arr[pos];
        } else if (pos - l > k - 1) {
            return KthSmallestWC(arr, l, pos - 1, k);
        } else {
            return KthSmallestWC(arr, pos + 1, r, k - pos + l - 1);
        }
    }

    elmtype getMedian(elmtype *arr, int n) {
        sort(arr, n);
        return arr[n / 2];
    }

    int partition(elmtype arr[], int l, int r, elmtype pivot) {
        int i;
        for (i = l; i < r; i++) {
            if (arr[i] == pivot) {
                break;
            }
        }
        swap(&arr[i], &arr[r]);
        i = l;
        for (int j = l; j <= r - 1; j++) {
            if (arr[j] <= pivot) {
                swap(&arr[i], &arr[j]);
                i++;
            }
        }
        swap(&arr[i], &arr[r]);
        return i;
    }

    void sort(elmtype *arr, int n) {
        for (int i = 0; i < n; i++) {
            elmtype *smallest = &arr[0];
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < (*smallest)) {
                    smallest = &arr[j];
                }
            }
            if (&arr[i] != smallest) {
                swap(&arr[i], smallest);
            }
```

```
        }
    }
};

template<typename keyType> class Heap {

    public:

    Heap() { info = new CircularDynamicArray<keyType>; }

    Heap(keyType K[], int s) {
        info = new CircularDynamicArray<keyType>(s);

        for (int i = 0; i < s; i++) {
            (*info)[i] = K[i];
        }

        heapify();
    }

    ~Heap() {
        // The CDA is deleted in its cleanup, deleting here will segfault
    }

    int size() {
        return info->length();
    }

    keyType peekKey() {
        return (*info)[0];
    }

    void insert(keyType k) {
        info->addEnd(k);
        siftUp(info->length() - 1);
    }

    void printKeys(std::ostream &out = std::cout) {
        for (int i = 0; i < info->length(); i++) {
            out « (*info)[i];
            if (i != info->length() - 1) {
                out « " ";
            }
        }
        if (info->length() != 0) {out « std::endl;}
    }

    keyType extractMin() {
        keyType min = (*info)[0];
        (*info)[0] = (*info)[info->length() - 1];
        info->delEnd();
        siftDown(0);
        return min;
    }

    private:

    CircularDynamicArray<keyType> *info;

    void heapify() {
        for (int i = info->length() / 2; i >= 0; i--) {
            siftDown(i);
        }
    }

    void siftDown(int i) {
        int l = lIndex(i), r = rIndex(i), min = i;

        if (l < info->length() && (*info)[l] < (*info)[i]) { min = l; }
        if (r < info->length() && (*info)[r] < (*info)[min]) { min = r; }
        if (min != i) { swap(i, min); siftDown(min); }
    }

    void siftUp(int i) {
        for (; i != 0 && (*info)[i] < (*info)[pIndex(i)]; i = pIndex(i)) {
            swap(i, pIndex(i));
        }
    }

    void swap(int a, int b) {
        keyType temp = (*info)[a];
        (*info)[a] = (*info)[b];
        (*info)[b] = temp;
    }

    int pIndex(int i) {
        return (i-1) / 2;
```

```
    }

    int lIndex(int i) {
        return (i*2) + 1;
    }

    int rIndex(int i) {
        return (i*2) + 2;
    }
};

enum color {Red, Black};

template<typename keyType, typename valueType> class RBNode {

    public:
    keyType *key;

    valueType *val;

    RBNode *l;

    RBNode *r;

    RBNode *p;

    color c;

    int size;

    RBNode() {
        key = new keyType;
        val = new valueType;
        l = nullptr;
        r = nullptr;
        p = nullptr;
        size = 0;
        c = Red;
    }

    RBNode(keyType k, valueType v) {
        key = new keyType; *key = k;
        val = new valueType; *val = v;
        l = nullptr;
        r = nullptr;
        p = nullptr;
        size = 1;
        c = Red;
    }

    RBNode(bool nilCon) {
        l = nullptr;
        r = nullptr;
        p = nullptr;
        key = nullptr;
        val = nullptr;
        size = 0;
        c = Black;
    }

    RBNode (keyType k, valueType v, color setc, int s, RBNode<keyType, valueType> *parent) {
        key = new keyType(k);
        val = new valueType(v);
        l = nullptr;
        r = nullptr;
        p = parent;
        c = setc;
        size = s;
    }

    RBNode(const RBNode<keyType, valueType> &src) {
        *key = *src.key;
        *val = *src.val;
        l = src.l;
        r = src.r;
        p = src.p;
        c = src.c;
        size = src.size;
    }

    ~RBNode() {
        if (key != nullptr) {delete key;}
        if (val != nullptr) {delete val;}
    }

    void cascade(RBNode *nil) {
        if (this == nil) {return;}
```

```
        if (l != nil && l != nullptr) {
            l->cascade(nil);
        }
        if (r != nil && r != nullptr) {
            r->cascade(nil);
        }
        delete this;
    }

    RBNode &operator=(RBNode R) {
        if (this == R) {
            return *this;
        }
        delete key; key = new keyType(R.key);
        delete val; val = new valueType(R.val);
        c = R.c;
        return *this;
    }

    void preorder (std::ostream &out = std::cout) {
        if (key == nullptr) {
            return;
        }
        printNode(out);
        if (l->key != nullptr) {out « " "; l->preorder(out);}
        if (r->key != nullptr) {out « " "; r->preorder(out);}
    }

    void inorder (std::ostream &out = std::cout) {
        if (key == nullptr) {
            return;
        }
        if (l->key != nullptr) {l->inorder(out); out « " ";}
        printNode(out);
        if (r->key != nullptr) {out « " "; r->inorder(out);}
    }

    void postorder (std::ostream &out = std::cout) {
        if (key == nullptr) {
            return;
        }
        if (l->key != nullptr) {l->postorder(out); out « " ";}
        if (r->key != nullptr) {r->postorder(out); out « " ";}
        printNode(out);
    }

    void printNode(std::ostream &out = std::cout) {
        if (key == nullptr) {
            return;
        }
        out « *key;
    }

    void printk(int &k, std::ostream &out = std::cout) {
        if (l->key != nullptr) {
            l->printk(k, out);
            if (k > 0) {
                out « " ";
            }
        }
        if (k < 1) {
            return;
        }
        out « *key;
        k--;
        if (k < 1) {
            return;
        }
        if (r->key != nullptr) {
            out « " ";
            r->printk(k, out);
        }
    }

    valueType *searchValue(keyType k) {

        if (key == nullptr) {
            return nullptr;
        }

        if (k == *key) {
            return val;
        }

        if (k < *key && l != nullptr) {
            return l->searchValue(k);
        }
```

```
        if (k > *key && r != nullptr) {
            return r->searchValue(k);
        }

        return nullptr;
    }

    RBNode<keyType, valueType> *searchNode(keyType k) {
        if (k == *key) {
            return this;
        } else if (k < *key && l->key != nullptr) {
            return l->searchNode(k);
        } else if (k > *key && r->key != nullptr) {
            return r->searchNode(k);
        } else {
            return nullptr;
        }
    }

    RBNode<keyType, valueType> *predecessor() {
        if (l == nullptr) {
            return nullptr;
        }
        RBNode *curr;
        for (curr = l; curr->l != nullptr; curr = curr->l) {continue;}
        return curr;
    }

    RBNode<keyType, valueType> *min() {
        if (l == nullptr) {
            return this;
        }
        RBNode *curr;
        for (curr = l; curr->l != nullptr; curr = curr->l) {continue;}
        return curr;
    }

    keyType select(int k) {
        if (k == l->size + 1) {
            return (*key);
        } else if (k <= l->size) {
            return l->select(k);
        } else {
            return r->select(k - l->size - 1);
        }
    }
};

template<typename keyType, typename valueType> class RBTree {
    public:

    RBTree() {
        nil = new RBNode<keyType, valueType>(true);
        root = nil;
    }

    RBTree(keyType k, valueType v) {
        nil = new RBNode<keyType, valueType>(true);
        root = new RBNode<keyType, valueType>(k, v);
        root->c = Black; // root is black
    }

    RBTree(keyType *k, valueType *v, int s) {
        nil = new RBNode<keyType, valueType>(true); // nil node's special constructor
        root = nil;
        for (int i = 0; i < s; i++) {
            insert(k[i], v[i]);
        }
    }

    RBTree(const RBTree<keyType, valueType> &src) {
        nil = new RBNode<keyType, valueType>(true);
        root = nil;
        copy(src.nil, src.root, root, nil);
    }

    ~RBTree() {
        root->cascade(nil);
    }

    RBTree<keyType, valueType> &operator=(const RBTree<keyType, valueType> &R) {
        if (this == &R) {
            return *this;
        }

        root->cascade(nil);
```

```
        nil = new RBNode<keyType, valueType>(true);
        root = nil;
        copy(*&R.nil, *&R.root, root, nil);
        return *this;
    }

    void preorder (std::ostream &out) {
        root->preorder(out);
        out « std::endl;
    }

    void inorder (std::ostream &out) {
        root->inorder(out);
        out « std::endl;
    }

    void postorder (std::ostream &out) {
        root->postorder(out);
        out « std::endl;
    }

    void printk (int k, std::ostream &out) {
        root->printk(k, out);
        out « std::endl;
    }

    int size() {
        return root->size;
    }

    valueType *search (keyType k) {
        return root->searchValue(k);
    }

    void insert(keyType k, valueType v) {
        RBNode<keyType, valueType> *z = new RBNode<keyType, valueType>(k, v);
        z->size = 1;
        z->l = nil; z->r = nil;
        RBNode<keyType, valueType> *y = nil;
        RBNode<keyType, valueType> *x = root;

        while (x != nil) {
            (x->size)++;

            y = x;

            if (*(z->key) < *(x->key)) {
                x = x->l;
            } else {
                x = x->r;
            }
        }

        z->p = y;
        if (y == nil) {
            root = z;
        } else if (*(z->key) < *(y->key)) {
            y->l = z;
        } else {
            y->r = z;
        }

        z->l = nil;
        z->r = nil;
        insertFixTree(z);
    }
        // Removes the node with key k from the tree.
        // Time complexity: O()

    int remove(keyType k) {
        RBNode<keyType, valueType> *z = root->searchNode(k);
        if (z == nullptr) {
            return 0;
        }
        RBNode<keyType, valueType> *y = z;
        RBNode<keyType, valueType> *x;

        for (RBNode<keyType, valueType> *i = z->p; i != nil; i = i->p) {
            i->size--;
        }

        color yOrigcolor = y->c;

        if (z->l == nil) {
            x = z->r;
            transplant(z, z->r);
        } else if (z->r == nil) {
```

```
            x = z->l;
            transplant(z, z->l);
        } else {
            y = max(z->l);
            yOrigcolor = y->c;
            x = y->l;
            if (y->p == z) {
                x->p = y;
            } else {
                transplant(y, y->l);
                y->l = z->l;
                y->l->p = y;
                for (RBNode<keyType, valueType> *i = x->p; i != nil && i != y; i = i->p) {
                    i->size--;
                }
            }
            transplant(z, y);
            y->r = z->r;
            y->r->p = y;
            y->c = z->c;
            y->size = y->l->size + y->r->size + 1;
        }

        if (yOrigcolor == Black) {
            deleteFixTree(x);
        }

        return 1;
    }

    int rank(keyType k) {
        RBNode<keyType, valueType> *node = root->searchNode(k);
        if (node == nullptr) {
            return 0;
        }
        int rank = node->l->size + 1;

        for (RBNode<keyType, valueType> *curr = node; curr != root; curr = curr->p) {
            if (curr == curr->p->r) {
                rank += curr->p->l->size + 1;
            }
        }
        return rank;
    }

    keyType select(int k) {
        return root->select(k);
    }

    keyType *successor(keyType k) {
        RBNode<keyType, valueType> *curr = root->searchNode(k);
        if (curr == nullptr) {
            return nullptr;
        }

        if (curr->r != nil) {
            for (curr = curr->r; curr->l != nil; curr = curr->l) {continue;}
            return curr->key;
        } else {
            RBNode<keyType, valueType> *i;
            for (i = curr->p; i != nil && curr == i->r; i = i->p) {curr = i;}
            return i->key;
        }

    }

    keyType *predecessor(keyType k) {
        RBNode<keyType, valueType> *curr = root->searchNode(k);
        if (curr == nullptr) {
            return nullptr;
        }

        if (curr->l != nil) {
            for (curr = curr->l; curr->r != nil; curr = curr->r) {continue;}
            return curr->key;
        } else {
            RBNode<keyType, valueType> *i;
            for (i = curr->p; i != nil && curr == i->l; i = i->p) {curr = i;}
            return i->key;
        }
    }

    private:

    RBNode<keyType, valueType> *root;

    RBNode<keyType, valueType> *nil;
```

```
    void copy(RBNode<keyType, valueType> *copiedNil, RBNode<keyType, valueType> *copiedNode, RBNode<keyType, valueType> *&newNode, RBNode<keyType, valueType> *&newParent) {
        newNode = new RBNode<keyType, valueType>(*(copiedNode->key), *(copiedNode->val), copiedNode->c, copiedNode->size, newParent);

        if (copiedNode->l == copiedNil || copiedNode->l == nullptr) {
            newNode->l = nil;
        } else {
            copy(copiedNil, copiedNode->l, newNode->l, newNode);
        }
        if (copiedNode->r == copiedNil || copiedNode->r == nullptr) {
            newNode->r = nil;
        } else {
            copy(copiedNil, copiedNode->r, newNode->r, newNode);
        }
    }

    RBNode<keyType, valueType> *min(RBNode<keyType, valueType> *node) {
        for (node; node->l != nil; node = node->l) {
            continue;
        }
        return node;
    }

    RBNode<keyType, valueType> *max(RBNode<keyType, valueType> *node) {
        for (; node->r != nil; node = node->r) {
            continue;
        }
        return node;
    }

    void insertFixTree(RBNode<keyType, valueType> *z) {
        RBNode<keyType, valueType> *y;

        while (z->p->c == Red) {
            if (z->p == z->p->p->l) {
                y = z->p->p->r;

                if (y->c == Red) {
                    z->p->c = Black;
                    y->c = Black;
                    z->p->p->c = Red;
                    z = z->p->p;
                } else {
                    if (z == z->p->r) {
                        z = z->p;
                        lRotate(z);
                    }

                    z->p->c = Black;
                    z->p->p->c = Red;
                    rRotate(z->p->p);
                }
            } else {
                y = z->p->p->l;

                if (y->c == Red) {
                    z->p->c = Black;
                    y->c = Black;
                    z->p->p->c = Red;
                    z = z->p->p;
                } else {
                    if (z == z->p->l) {
                        z = z->p;
                        rRotate(z);
                    }

                    z->p->c = Black;
                    z->p->p->c = Red;
                    lRotate(z->p->p);
                }
            }
        }
        root->c = Black;
    }

    void deleteFixTree(RBNode<keyType, valueType> *x) {

        RBNode<keyType, valueType> *w;
        while (x != root && x->c == Black) {
            if (x == x->p->l) {
                w = x->p->r;
                if (w->c == Red) {
                    w->c = Black;
                    x->p->c = Red;
                    lRotate(x->p);
```

```
                    w = x->p->r;
                }

                if (w->l->c == Black && w->r->c == Black) {
                    w->c = Red;
                    x = x->p;
                } else {
                    if (w->r->c == Black) {
                        w->l->c = Black;
                        w->c = Red;
                        rRotate(w);
                        w = x->p->r;
                    }

                    w->c = x->p->c;
                    x->p->c = Black;
                    w->r->c = Black;
                    lRotate(x->p);
                    x = root;
                }
            } else {
                w = x->p->l;
                if (w->c == Red) {
                    w->c = Black;
                    x->p->c = Red;
                    rRotate(x->p);
                    w = x->p->l;
                }

                if (w->r->c == Black && w->l->c == Black) {
                    w->c = Red;
                    x = x->p;
                } else {
                    if (w->l->c == Black) {
                        w->r->c = Black;
                        w->c = Red;
                        lRotate(w);
                        w = x->p->l;
                    }

                    w->c = x->p->c;
                    x->p->c = Black;
                    w->l->c = Black;
                    rRotate(x->p);
                    x = root;
                }
            }
        }
    }

    void rRotate(RBNode<keyType, valueType> *x) {
        RBNode<keyType, valueType> *y = x->l;
        x->l = y->r;
        if (y->r != nil) {
            y->r->p = x;
        }
        y->p = x->p;
        if (x->p == nil) {
            root = y;
        } else if (x == x->p->r) {
            x->p->r = y;
        } else {
            x->p->l = y;
        }
        y->r = x;
        x->p = y;
        y->size = x->size;
        x->size = x->l->size + x->r->size + 1;
    }

    void lRotate(RBNode<keyType, valueType> *x) {
        RBNode<keyType, valueType> *y = x->r;
        x->r = y->l;
        if (y->l != nil) {
            y->l->p = x;
        }
        y->p = x->p;
        if (x->p == nil) {
            root = y;
        } else if (x == x->p->l) {
            x->p->l = y;
        } else {
            x->p->r = y;
        }
        y->l = x;
        x->p = y;
        y->size = x->size;
```

```
        x->size = x->l->size + x->r->size + 1;
    }

    void transplant(RBNode<keyType, valueType> *u, RBNode<keyType, valueType> *v) {

        // If u is the root, set the root to v
        if (u->p == nil) {
            root = v;

        // If u is the left child of its parent, set the left child of its parent to v
        } else if (u == u->p->l) {
            u->p->l = v;

        // If u is the right child of its parent, set the right child of its parent to v
        } else {
            u->p->r = v;
        }

        // Set the parent of v to the parent of u
        v->p = u->p;
    }
};
#endif
```

Index