Lab 2. Search algorithms and data structures

Gilyazov Danila https://github.com/skyfxllexe/mef_of_prog_lab2

Generated by Doxygen 1.9.8

1 Class Index 1

1.1 Class List . . . 1

2 File Index 3

2.1 File List . . . 3

3 Class Documentation 5

3.1 BinTree< T > Class Template Reference . . . 5

3.1.1 Detailed Description . . . 5

3.1.2 Member Function Documentation . . . 6

3.1.2.1 deleteTree_rec() . . . 6

3.1.2.2 find() . . . 6

3.1.2.3 insert() . . . 6

3.1.2.4 printTree() . . . 7

3.2 Employee Class Reference . . . 7

3.3 Node< T > Class Template Reference . . . 8

3.3.1 Detailed Description . . . 8

3.3.2 Member Function Documentation . . . 8

3.3.2.1 add_object() . . . 8

3.4 RBTree< T > Class Template Reference . . . 10

3.4.1 Detailed Description . . . 10

3.4.2 Member Function Documentation . . . 11

3.4.2.1 deleteTree_rec() . . . 11

3.4.2.2 find_object() . . . 11

3.4.2.3 fixInsert() . . . 11

3.4.2.4 insert() . . . 12

3.4.2.5 printTree() . . . 12

3.4.2.6 rotateLeft() . . . 12

3.4.2.7 rotateRight() . . . 12

3.5 TableHash< T > Class Template Reference . . . 13

3.5.1 Detailed Description . . . 13

3.5.2 Member Function Documentation . . . 14

3.5.2.1 add() . . . 14

3.5.2.2 find() . . . 14

3.5.2.3 get_count() . . . 14

4 File Documentation 15

4.1 bin_tree.hpp . . . 15

4.2 employee.cpp File Reference . . . 16

4.2.1 Detailed Description . . . 16

4.3 employee.hpp . . . 17

4.4 main.cpp File Reference . . . 17

4.4.1 Detailed Description . . . 18

4.4.2 Function Documentation 18
4.4.2.1 linear_search() 18
4.5 node.hpp 18
4.6 rbtree.hpp 18
4.7 tablehash.hpp 21

Index **23**

Chapter 1

Class Index

1.1 Class List

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

BinTree< T >
: A generic binary search tree for storing objects with string keys 5
Employee 7
Node< T >
: A generic node class for use in binary and red-black trees 8
RBTree< T >
: Red-Black Tree implementation 10
TableHash< T >
: Hash table implementation using open addressing with linear probing 13

Chapter 2

File Index

2.1 File List

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

bin_tree.hpp 15
employee.cpp
File with implementation of Employee class methods 16
employee.hpp 17
main.cpp
Main 17
node.hpp 18
rbtree.hpp 18
tablehash.hpp 21

Chapter 3

Class Documentation

3.1 BinTree< T > Class Template Reference

A generic binary search tree for storing objects with string keys.

`#include <bin_tree.hpp>`

Public Member Functions

- void insert (T object)
  Inserts an object into the binary search tree.
- std::vector< T > find (std::string key)
  Finds all objects with the specified key.
- ~**BinTree** ()
  Destructor that deletes the entire binary tree.
- void deleteTree_rec (Node< T > *node)
  Recursively deletes nodes in the binary tree.
- void printTree (Node< T > *node, int level)
  Recursively prints the tree contents starting from a node.

Public Attributes

- Node< T > * **root** = nullptr
  Pointer to the root of the binary tree.

3.1.1 Detailed Description

template<class T>
class BinTree< T >

A generic binary search tree for storing objects with string keys.

Template Parameters

T	Type of elements stored. Must contain a `key` field of type std::string and support `add_object`.

3.1.2 Member Function Documentation

3.1.2.1 deleteTree_rec()

```
template<class T >
void BinTree< T >::deleteTree_rec (
            Node< T > * node )  [inline]
```

Recursively deletes nodes in the binary tree.

Parameters

node	Current node to delete.

3.1.2.2 find()

```
template<class T >
std::vector< T > BinTree< T >::find (
            std::string key )  [inline]
```

Finds all objects with the specified key.

Parameters

key	The key to search for.

Returns

A vector of matching objects. Returns an empty vector if key is not found.

3.1.2.3 insert()

```
template<class T >
void BinTree< T >::insert (
            T object )  [inline]
```

Inserts an object into the binary search tree.

If the key already exists, the object is added to the node's object list.

Parameters

object	The object to insert.

3.1.2.4 printTree()

```
template<class T >
void BinTree< T >::printTree (
             Node< T > * node,
             int level )  [inline]
```

Recursively prints the tree contents starting from a node.

Parameters

node	The current node to print.
level	The indentation level (used for visualizing the tree).

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

- bin_tree.hpp

3.2 Employee Class Reference

Public Member Functions

- **Employee** (std::string, std::string, int)
- bool **operator**> (Employee &other)
- bool **operator**< (Employee &other)
- bool **operator**>= (Employee &other)
- bool **operator**<= (Employee &other)
- bool **operator==** (Employee &other)

Public Attributes

- std::string **key**
- std::string **employee_name**
- std::string **subdivision**
- int **salary**

Friends

- std::ostream & **operator**<< (std::ostream &os, const Employee &person)

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

- employee.hpp
- employee.cpp

3.3 Node< T > Class Template Reference

A generic node class for use in binary and red-black trees.

`#include <node.hpp>`

Public Member Functions

- void add_object (T object)

  Adds an object to the node's object list.

Public Attributes

- T **object**

  A single object of type T (not used in main logic, kept for compatibility).
- std::string **key**

  The key used to identify this node (typically from the first object).
- std::vector< T > **objects**

  A list of objects with the same key.
- int **color** = 2

  Node color for Red-Black Tree: 0 = black, 1 = red, 2 = null/uninitialized.
- Node< T > * **parent** = nullptr

  Pointer to the parent node.
- Node< T > * **left** = nullptr

  Pointer to the left child node.
- Node< T > * **right** = nullptr

  Pointer to the right child node.

3.3.1 Detailed Description

template<class T>
class Node< T >

A generic node class for use in binary and red-black trees.

Template Parameters

T	The type of object stored in the node. Must have a `key` field.

3.3.2 Member Function Documentation

3.3.2.1 add_object()

```
template<class T >
void Node< T >::add_object (
            T object )  [inline]
```

Adds an object to the node's object list.

Parameters

object	The object to add.

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

- node.hpp

3.4 RBTree< T > Class Template Reference

Red-Black Tree implementation.

`#include <rbtree.hpp>`

Public Member Functions

- ~**RBTree** ()

  Destructor. Recursively deletes all nodes in the tree.
- void deleteTree_rec (Node< T > *node)

  Recursively deletes all nodes in the tree starting from the given node.
- void rotateLeft (Node< T > *&node)

  Performs a left rotation around the given node.
- void rotateRight (Node< T > *&node)

  Performs a right rotation around the given node.
- void fixInsert (Node< T > *&node)

  Fixes red-black tree properties after insertion.
- void insert (T object)

  Inserts a new object into the red-black tree.
- void printTree (Node< T > *node, int level)

  Prints the red-black tree to the console (for debugging).
- std::vector< T > * find_object (std::string key)

  Searches for a node by key and returns all associated objects.

Public Attributes

- Node< T > * **root** = nullptr

  Root of the red-black tree.
- int **elements**

  Number of elements (not directly used)

3.4.1 Detailed Description

template<class T>
class RBTree< T >

Red-Black Tree implementation.

Template Parameters

T	Type of elements stored in the tree.

3.4.2 Member Function Documentation

3.4.2.1 deleteTree_rec()

```
template<class T >
void RBTree< T >::deleteTree_rec (
            Node< T > * node )  [inline]
```

Recursively deletes all nodes in the tree starting from the given node.

Parameters

node	The node to start deletion from.

3.4.2.2 find_object()

```
template<class T >
std::vector< T > * RBTree< T >::find_object (
            std::string key )  [inline]
```

Searches for a node by key and returns all associated objects.

Parameters

key	Key to search for.

Returns

Pointer to a vector of objects if found, otherwise nullptr.

3.4.2.3 fixInsert()

```
template<class T >
void RBTree< T >::fixInsert (
            Node< T > *& node )  [inline]
```

Fixes red-black tree properties after insertion.

Parameters

node	Newly inserted node.

3.4.2.4 insert()

```
template<class T >
void RBTree< T >::insert (
            T object )  [inline]
```

Inserts a new object into the red-black tree.

Parameters

object	Object to insert.

3.4.2.5 printTree()

```
template<class T >
void RBTree< T >::printTree (
            Node< T > * node,
            int level )  [inline]
```

Prints the red-black tree to the console (for debugging).

Parameters

node	Current node.
level	Current depth level (used for indentation).

3.4.2.6 rotateLeft()

```
template<class T >
void RBTree< T >::rotateLeft (
            Node< T > *& node )  [inline]
```

Performs a left rotation around the given node.

Parameters

node	Reference to the node to rotate.

3.4.2.7 rotateRight()

```
template<class T >
void RBTree< T >::rotateRight (
            Node< T > *& node )  [inline]
```

Performs a right rotation around the given node.

Parameters

node	Reference to the node to rotate.

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

- rbtree.hpp

3.5 TableHash< T > Class Template Reference

Hash table implementation using open addressing with linear probing.

`#include <tablehash.hpp>`

Public Member Functions

- std::vector< T > * find (std::string key)
  Finds the list of elements with the specified key.
- **TableHash** ()
  Constructs a new TableHash object with default size.
- ~**TableHash** ()
  Destroys the TableHash object.
- void add (T object)
  Adds an object to the hash table.
- int get_count ()
  Returns the number of unique keys stored in the table.

Public Attributes

- int **size** = 100
  Current capacity of the hash table.

3.5.1 Detailed Description

template<class T>
class TableHash< T >

Hash table implementation using open addressing with linear probing.

Template Parameters

T	Type of elements stored. Must have a `key` field of type std::string.

3.5.2 Member Function Documentation

3.5.2.1 add()

```
template<class T >
void TableHash< T >::add (
            T object )
```

Adds an object to the hash table.

Parameters

object	The object to add.

3.5.2.2 find()

```
template<class T >
std::vector< T > * TableHash< T >::find (
            std::string key )
```

Finds the list of elements with the specified key.

Parameters

key	The key to search for.

Returns

Pointer to the vector of matching elements.

3.5.2.3 get_count()

```
template<class T >
int TableHash< T >::get_count ( )
```

Returns the number of unique keys stored in the table.

Returns

The count of keys.

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

- tablehash.hpp

Chapter 4

File Documentation

4.1 bin_tree.hpp

```
00001
00006 template<class T>
00007 class BinTree {
00008 public:
00009     Node<T>* root = nullptr;
00010
00018     void insert(T object) {
00019         Node<T>* current = root;
00020         Node<T>* parent = nullptr;
00021         int flag_new_node = 1;
00022         while(current != nullptr) {
00023             parent = current;
00024             if(object.key > current->key) {
00025                 current = current->right;
00026             }
00027             else if (object.key < current->key) {
00028                 current = current->left;
00029             }
00030             else {
00031                 current->add_object(object);
00032                 return;
00033             }
00034         }
00035         current = new Node<T>;
00036         current->key = object.key;
00037         if(parent != nullptr) {
00038             current->parent = parent;
00039             if(current->key > parent->key) {
00040                 parent->right = current;
00041             }
00042             else {
00043                 parent->left = current;
00044             }
00045         }
00046         else {
00047             root = current;
00048         }
00049         current->add_object(object);
00050     }
00051
00058     std::vector<T> find(std::string key) {
00059         std::vector<T> ans;
00060         Node<T>* current = root;
00061         while(current != nullptr) {
00062             if(key > current->key) {
00063                 current = current->right;
00064             }
00065             else if(key < current->key) {
00066                 current = current->left;
00067             }
00068             else {
00069                 int k = ans.size();
00070                 ans.resize(ans.size() + current->objects.size());
00071                 for(int i = k; i < ans.size(); i++) {
00072                     ans[i] = current->objects[i-k];
00073                 }
00074                 break;
00075             }
```

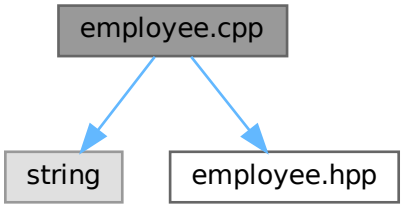
```
00076         }
00077         return ans;
00078     }
00079
00083     ~BinTree(){
00084         deleteTree_rec(root);
00085     }
00086
00092     void deleteTree_rec(Node<T>* node)
00093     {
00094         if (node != nullptr) {
00095             deleteTree_rec(node->left);
00096             deleteTree_rec(node->right);
00097             delete node;
00098         }
00099     }
00100
00107     void printTree(Node<T>* node, int level)
00108     {
00109         for(int i = 0; i < level; i++) {
00110             std::cout << "     ";
00111         }
00112         std::cout << node->objects[0] << std::endl;
00113         if(node->left != nullptr) { printTree(node->left, level+1); }
00114         if(node->right != nullptr) { printTree(node->right, level + 1); }
00115     }
00116 };
```

4.2 employee.cpp File Reference

File with implementation of Employee class methods.

```
#include <string>
#include "employee.hpp"
```

Include dependency graph for employee.cpp:



Functions

- std::ostream & **operator**<< (std::ostream &os, const Employee &employee)

4.2.1 Detailed Description

File with implementation of Employee class methods.

4.3 employee.hpp

```
00001 class Employee {
00002 public:
00003     Employee(std::string, std::string, int);
00004     Employee();
00005     std::string key;
00006     std::string employee_name;
00007     std::string subdivision;
00008     int salary;
00009     bool operator>(Employee &other);
00010     bool operator<(Employee &other);
00011     bool operator>=(Employee &other);
00012     bool operator<=(Employee &other);
00013     bool operator==(Employee &other);
00014     friend std::ostream& operator«(std::ostream& os, const Employee& person);
00015 };
00016
00017
```

4.4 main.cpp File Reference

main

```
#include <list>
#include <iostream>
#include <fstream>
#include <sstream>
#include <chrono>
#include <algorithm>
#include <cmath>
#include <vector>
#include <utility>
#include <map>
#include "employee.hpp"
#include "node.hpp"
#include "bin_tree.hpp"
#include "rbtree.hpp"
#include "tablehash.hpp"
```

Include dependency graph for main.cpp:

Functions

- template<class T >
  int linear_search (T a[ ], long start, long size, std::string key)

  Linear search function.
- int **main** ()

Variables

- Employee * **employees**

4.4.1 Detailed Description

main

4.4.2 Function Documentation

4.4.2.1 linear_search()

```
template<class T >
int linear_search (
             T a[],
             long start,
             long size,
             std::string key )
```

Linear search function.

Parameters

in	*a*	Array for search
in	*start*	Start index for search
in	*size*	Size of an array
in	*key*	Element key to find in array

Returns

Index of element with key

4.5 node.hpp

```
00001
00006 template<class T>
00007 class Node {
00008 public:
00009     T object;
00010     std::string key;
00011     std::vector<T> objects;
00012     int color = 2;
00013     Node<T>* parent = nullptr;
00014     Node<T>* left = nullptr;
00015     Node<T>* right = nullptr;
00016
00022     void add_object(T object) {
00023         objects.resize(objects.size() + 1);
00024         objects[objects.size() - 1] = object;
00025     }
00026 };
```

4.6 rbtree.hpp

```
00001
00006 template<class T>
00007 class RBTree {
00008 public:
00009     Node<T>* root = nullptr;
00010     int elements;
00011
00015     ~RBTree(){
```

```
00016         deleteTree_rec(root);
00017     }
00018
00024     void deleteTree_rec(Node<T>* node)
00025     {
00026         if (node != nullptr) {
00027             deleteTree_rec(node->left);
00028             deleteTree_rec(node->right);
00029             delete node;
00030         }
00031     }
00032
00038     void rotateLeft(Node<T>*& node)
00039     {
00040         Node<T>* child = node->right;
00041         node->right = child->left;
00042         if (node->right != nullptr)
00043             node->right->parent = node;
00044         child->parent = node->parent;
00045         if (node->parent == nullptr)
00046             root = child;
00047         else if (node == node->parent->left)
00048             node->parent->left = child;
00049         else
00050             node->parent->right = child;
00051         child->left = node;
00052         node->parent = child;
00053     }
00054
00060     void rotateRight(Node<T>*& node)
00061     {
00062         Node<T>* child = node->left;
00063         node->left = child->right;
00064         if (node->left != nullptr)
00065             node->left->parent = node;
00066         child->parent = node->parent;
00067         if (node->parent == nullptr)
00068             root = child;
00069         else if (node == node->parent->left)
00070             node->parent->left = child;
00071         else
00072             node->parent->right = child;
00073         child->right = node;
00074         node->parent = child;
00075     }
00076
00082     void fixInsert(Node<T>*& node)
00083     {
00084         Node<T>* parent = node->parent;
00085         Node<T>* grandparent = nullptr;
00086         if (parent != nullptr) {
00087             grandparent = node->parent->parent;
00088         }
00089
00090         while (node != root && node->color == 1 && node->parent->color == 1) {
00091             parent = node->parent;
00092             grandparent = parent->parent;
00093
00094             if (parent == grandparent->left) {
00095                 Node<T>* uncle = grandparent->right;
00096
00097                 if (uncle != nullptr && uncle->color == 1) {
00098                     grandparent->color = 1;
00099                     parent->color = 0;
00100                     uncle->color = 0;
00101                     node = grandparent;
00102                 } else {
00103                     if (node == parent->right) {
00104                         rotateLeft(parent);
00105                         node = parent;
00106                         parent = node->parent;
00107                     }
00108                     rotateRight(grandparent);
00109                     std::swap(parent->color, grandparent->color);
00110                     node = parent;
00111                 }
00112             } else {
00113                 Node<T>* uncle = grandparent->left;
00114
00115                 if (uncle != nullptr && uncle->color == 1) {
00116                     grandparent->color = 1;
00117                     parent->color = 0;
00118                     uncle->color = 0;
00119                     node = grandparent;
00120                 } else {
00121                     if (node == parent->left) {
00122                         rotateRight(parent);
```

```
00123                         node = parent;
00124                         parent = node->parent;
00125                     }
00126                     rotateLeft(grandparent);
00127                     std::swap(parent->color, grandparent->color);
00128                     node = parent;
00129                 }
00130             }
00131         }
00132
00133         root->color = 0;
00134     }
00135
00141     void insert(T object)
00142     {
00143         Node<T>* node = new Node<T>;
00144         node->add_object(object);
00145         node->color = 1;
00146
00147         Node<T>* parent = nullptr;
00148         Node<T>* current = root;
00149
00150         while (current != nullptr) {
00151             parent = current;
00152
00153             if (node->objects[0].key < current->objects[0].key) {
00154                 current = current->left;
00155             } else if (node->objects[0].key > current->objects[0].key) {
00156                 current = current->right;
00157             } else {
00158                 break;
00159             }
00160         }
00161
00162         node->parent = parent;
00163
00164         if (parent == nullptr) {
00165             root = node;
00166         } else if (node->objects[0].key < parent->objects[0].key) {
00167             parent->left = node;
00168         } else if (node->objects[0].key > parent->objects[0].key) {
00169             parent->right = node;
00170         } else {
00171             parent->add_object(node->objects[0]);
00172             delete node;
00173         }
00174
00175         fixInsert(node);
00176     }
00177
00184     void printTree(Node<T>* node, int level)
00185     {
00186         for (int i = 0; i < level; i++) {
00187             std::cout « "     ";
00188         }
00189         std::cout « node->objects[0] « " " « node->color « std::endl;
00190
00191         if (node->left != nullptr) {
00192             printTree(node->left, level + 1);
00193         }
00194
00195         if (node->right != nullptr) {
00196             printTree(node->right, level + 1);
00197         }
00198     }
00199
00206     std::vector<T>* find_object(std::string key) {
00207         Node<T>* current = root;
00208
00209         while (current != nullptr) {
00210             if (key > (current->objects[0]).key) {
00211                 current = current->right;
00212             } else if (key < (current->objects[0]).key) {
00213                 current = current->left;
00214             } else {
00215                 return &current->objects;
00216             }
00217         }
00218
00219         return nullptr;
00220     }
00221 };
```

4.7 tablehash.hpp

```
00001
00006 template<class T>
00007 class TableHash
00008 {
00009 public:
00016     std::vector<T>* find(std::string key);
00017
00021     TableHash();
00022
00026     ~TableHash();
00027
00033     void add(T object);
00034
00040     int get_count();
00041
00042     int size = 100;
00043
00044 private:
00048     void resize();
00049
00050     std::vector<std::vector<T> items;
00051     int count = 0;
00052     int a = 9;
00053 };
00054
00062 size_t pow(int a, int power)
00063 {
00064     if (power == 1)
00065         return a;
00066     if (power == 0)
00067         return 1;
00068     if (power & 1)
00069         return pow(a, power - 1) * a;
00070     return pow(a, power / 2) * pow(a, power / 2);
00071 }
00072
00081 size_t hash(std::string s, int a, int m)
00082 {
00083     size_t ans = 0;
00084     int size = s.size();
00085     for (int i = 0; i < size; i++)
00086     {
00087         ans += ((int)s[i] * pow(a, size - 1 - i)) % m;
00088     }
00089     ans %= m;
00090     return ans;
00091 }
00092
00093 template<class T>
00094 TableHash<T>::TableHash() {
00095     items.resize(size);
00096 }
00097
00098 template<class T>
00099 TableHash<T>::~TableHash() {
00100 }
00101
00102 template<class T>
00103 void TableHash<T>::resize() {
00104     size *= 2;
00105     std::vector<std::vector<T> new_items;
00106     new_items.resize(size);
00107     for (int i = 0; i < size / 2; i++) {
00108         size_t h = std::hash<std::string>{}(items[i][0].key) % size;
00109
00110         while (true) {
00111             if (new_items[h].empty()) {
00112                 break;
00113             } else {
00114                 h = (h + 1) % size;
00115             }
00116         }
00117
00118         new_items[h].resize(items[i].size());
00119
00120         for (int j = 0; j < items[i].size(); j++) {
00121             new_items[h][j] = items[i][j];
00122         }
00123     }
00124     items.clear();
00125     items = std::move(new_items);
00126 }
00127
00128 template<class T>
00129 void TableHash<T>::add(T object) {
```

```
00130     size_t h = std::hash<std::string>{}(object.key) % size;
00131     bool flag = true;
00132 
00133     if (count == size) {
00134         resize();
00135     }
00136 
00137     while (true) {
00138         if (items[h].empty()) {
00139             break;
00140         } else {
00141             if (items[h][0].key == object.key) {
00142                 break;
00143             } else {
00144                 h = (h + 1) % size;
00145             }
00146         }
00147     }
00148 
00149     if (items[h].empty()) {
00150         count += 1;
00151     }
00152 
00153     items[h].resize(items[h].size() + 1);
00154     items[h][items[h].size() - 1] = object;
00155 }
00156 
00157 template<class T>
00158 std::vector<T>* TableHash<T>::find(std::string key) {
00159     size_t h = std::hash<std::string>{}(key) % size;
00160     for(int i = 0; i < size; i++) {
00161         if (!items[h].empty()){
00162             if (items[h][0].key == key) {
00163                 break;
00164             }
00165             h = (h + 1) % size;
00166         }
00167     }
00168     return &items[h];
00169 }
00170 
00171 template<class T>
00172 int TableHash<T>::get_count()
00173 {
00174     return count;
00175 }
```

Index

add
    TableHash< T >, 14
add_object
    Node< T >, 8

BinTree< T >, 5
    deleteTree_rec, 6
    find, 6
    insert, 6
    printTree, 7

deleteTree_rec
    BinTree< T >, 6
    RBTree< T >, 11

Employee, 7
employee.cpp, 16

find
    BinTree< T >, 6
    TableHash< T >, 14
find_object
    RBTree< T >, 11
fixInsert
    RBTree< T >, 11

get_count
    TableHash< T >, 14

insert
    BinTree< T >, 6
    RBTree< T >, 11

linear_search
    main.cpp, 18

main.cpp, 17
    linear_search, 18

Node< T >, 8
    add_object, 8

printTree
    BinTree< T >, 7
    RBTree< T >, 12

RBTree< T >, 10
    deleteTree_rec, 11
    find_object, 11
    fixInsert, 11
    insert, 11
    printTree, 12
    rotateLeft, 12
    rotateRight, 12
rotateLeft
    RBTree< T >, 12
rotateRight
    RBTree< T >, 12

TableHash< T >, 13
    add, 14
    find, 14
    get_count, 14