Lab 2. Search algorithms and data structures

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

Generated by Doxygen 1.9.8

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 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

```cpp

template<class T>
class BinTree {
public:
    Node<T>* root = nullptr;

    void insert(T object) {
        Node<T>* current = root;
        Node<T>* parent = nullptr;
        int flag_new_node = 1;
        while(current != nullptr) {
            parent = current;
            if(object.key > current->key) {
                current = current->right;
            }
            else if (object.key < current->key) {
                current = current->left;
            }
            else {
                current->add_object(object);
                return;
            }
        }
        current = new Node<T>;
        current->key = object.key;
        if(parent != nullptr) {
            current->parent = parent;
            if(current->key > parent->key) {
                parent->right = current;
            }
            else {
                parent->left = current;
            }
        }
        else {
            root = current;
        }
        current->add_object(object);
    }

    std::vector<T> find(std::string key) {
        std::vector<T> ans;
        Node<T>* current = root;
        while(current != nullptr) {
            if(key > current->key) {
                current = current->right;
            }
            else if(key < current->key) {
                current = current->left;
            }
            else {
                int k = ans.size();
                ans.resize(ans.size() + current->objects.size());
                for(int i = k; i < ans.size(); i++) {
                    ans[i] = current->objects[i-k];
                }
                break;
            }
```

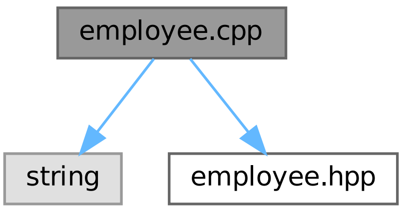
```
        }
        return ans;
    }

    ~BinTree(){
        deleteTree_rec(root);
    }

    void deleteTree_rec(Node<T>* node)
    {
        if (node != nullptr) {
            deleteTree_rec(node->left);
            deleteTree_rec(node->right);
            delete node;
        }
    }

    void printTree(Node<T>* node, int level)
    {
        for(int i = 0; i < level; i++) {
            std::cout « "     ";
        }
        std::cout « node->objects[0] « std::endl;
        if(node->left != nullptr) { printTree(node->left, level+1); }
        if(node->right != nullptr) { printTree(node->right, level + 1); }
    }
};
```

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

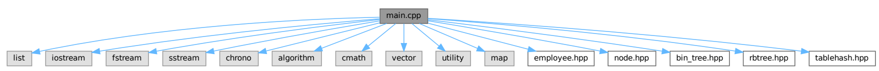
```
class Employee {
public:
    Employee(std::string, std::string, int);
    Employee();
    std::string key;
    std::string employee_name;
    std::string subdivision;
    int salary;
    bool operator>(Employee &other);
    bool operator<(Employee &other);
    bool operator>=(Employee &other);
    bool operator<=(Employee &other);
    bool operator==(Employee &other);
    friend std::ostream& operator«(std::ostream& os, const Employee& person);
};


```

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

```
template<class T>
class Node {
public:
    T object;
    std::string key;
    std::vector<T> objects;
    int color = 2;
    Node<T>* parent = nullptr;
    Node<T>* left = nullptr;
    Node<T>* right = nullptr;

    void add_object(T object) {
        objects.resize(objects.size() + 1);
        objects[objects.size() - 1] = object;
    }
};
```

4.6 rbtree.hpp

```
template<class T>
class RBTree {
public:
    Node<T>* root = nullptr;
    int elements;

    ~RBTree(){
```

```
        deleteTree_rec(root);
    }

    void deleteTree_rec(Node<T>* node)
    {
        if (node != nullptr) {
            deleteTree_rec(node->left);
            deleteTree_rec(node->right);
            delete node;
        }
    }

    void rotateLeft(Node<T>*& node)
    {
        Node<T>* child = node->right;
        node->right = child->left;
        if (node->right != nullptr)
            node->right->parent = node;
        child->parent = node->parent;
        if (node->parent == nullptr)
            root = child;
        else if (node == node->parent->left)
            node->parent->left = child;
        else
            node->parent->right = child;
        child->left = node;
        node->parent = child;
    }

    void rotateRight(Node<T>*& node)
    {
        Node<T>* child = node->left;
        node->left = child->right;
        if (node->left != nullptr)
            node->left->parent = node;
        child->parent = node->parent;
        if (node->parent == nullptr)
            root = child;
        else if (node == node->parent->left)
            node->parent->left = child;
        else
            node->parent->right = child;
        child->right = node;
        node->parent = child;
    }

    void fixInsert(Node<T>*& node)
    {
        Node<T>* parent = node->parent;
        Node<T>* grandparent = nullptr;
        if (parent != nullptr) {
            grandparent = node->parent->parent;
        }

        while (node != root && node->color == 1 && node->parent->color == 1) {
            parent = node->parent;
            grandparent = parent->parent;

            if (parent == grandparent->left) {
                Node<T>* uncle = grandparent->right;

                if (uncle != nullptr && uncle->color == 1) {
                    grandparent->color = 1;
                    parent->color = 0;
                    uncle->color = 0;
                    node = grandparent;
                } else {
                    if (node == parent->right) {
                        rotateLeft(parent);
                        node = parent;
                        parent = node->parent;
                    }
                    rotateRight(grandparent);
                    std::swap(parent->color, grandparent->color);
                    node = parent;
                }
            } else {
                Node<T>* uncle = grandparent->left;

                if (uncle != nullptr && uncle->color == 1) {
                    grandparent->color = 1;
                    parent->color = 0;
                    uncle->color = 0;
                    node = grandparent;
                } else {
                    if (node == parent->left) {
                        rotateRight(parent);
```

```
                        node = parent;
                        parent = node->parent;
                    }
                    rotateLeft(grandparent);
                    std::swap(parent->color, grandparent->color);
                    node = parent;
                }
            }
        }

        root->color = 0;
    }

    void insert(T object)
    {
        Node<T>* node = new Node<T>;
        node->add_object(object);
        node->color = 1;

        Node<T>* parent = nullptr;
        Node<T>* current = root;

        while (current != nullptr) {
            parent = current;

            if (node->objects[0].key < current->objects[0].key) {
                current = current->left;
            } else if (node->objects[0].key > current->objects[0].key) {
                current = current->right;
            } else {
                break;
            }
        }

        node->parent = parent;

        if (parent == nullptr) {
            root = node;
        } else if (node->objects[0].key < parent->objects[0].key) {
            parent->left = node;
        } else if (node->objects[0].key > parent->objects[0].key) {
            parent->right = node;
        } else {
            parent->add_object(node->objects[0]);
            delete node;
        }

        fixInsert(node);
    }

    void printTree(Node<T>* node, int level)
    {
        for (int i = 0; i < level; i++) {
            std::cout « "     ";
        }
        std::cout « node->objects[0] « " " « node->color « std::endl;

        if (node->left != nullptr) {
            printTree(node->left, level + 1);
        }

        if (node->right != nullptr) {
            printTree(node->right, level + 1);
        }
    }

    std::vector<T>* find_object(std::string key) {
        Node<T>* current = root;

        while (current != nullptr) {
            if (key > (current->objects[0]).key) {
                current = current->right;
            } else if (key < (current->objects[0]).key) {
                current = current->left;
            } else {
                return &current->objects;
            }
        }

        return nullptr;
    }
};
```

4.7 tablehash.hpp

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

    TableHash();

    ~TableHash();

    void add(T object);

    int get_count();

    int size = 100;

private:
    void resize();

    std::vector<std::vector<T» items;
    int count = 0;
    int a = 9;
};

size_t pow(int a, int power)
{
    if (power == 1)
        return a;
    if (power == 0)
        return 1;
    if (power & 1)
        return pow(a, power - 1) * a;
    return pow(a, power / 2) * pow(a, power / 2);
}

size_t hash(std::string s, int a, int m)
{
    size_t ans = 0;
    int size = s.size();
    for (int i = 0; i < size; i++)
    {
        ans += ((int)s[i] * pow(a, size - 1 - i)) % m;
    }
    ans %= m;
    return ans;
}

template<class T>
TableHash<T>::TableHash() {
    items.resize(size);
}

template<class T>
TableHash<T>::~TableHash() {
}

template<class T>
void TableHash<T>::resize() {
    size *= 2;
    std::vector<std::vector<T» new_items;
    new_items.resize(size);
    for (int i = 0; i < size / 2; i++) {
        size_t h = std::hash<std::string>{}(items[i][0].key) % size;

        while (true) {
            if (new_items[h].empty()) {
                break;
            } else {
                h = (h + 1) % size;
            }
        }

        new_items[h].resize(items[i].size());

        for (int j = 0; j < items[i].size(); j++) {
            new_items[h][j] = items[i][j];
        }
    }
    items.clear();
    items = std::move(new_items);
}

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

```
    size_t h = std::hash<std::string>{}(object.key) % size;
    bool flag = true;

    if (count == size) {
        resize();
    }

    while (true) {
        if (items[h].empty()) {
            break;
        } else {
            if (items[h][0].key == object.key) {
                break;
            } else {
                h = (h + 1) % size;
            }
        }
    }

    if (items[h].empty()) {
        count += 1;
    }

    items[h].resize(items[h].size() + 1);
    items[h][items[h].size() - 1] = object;
}

template<class T>
std::vector<T>* TableHash<T>::find(std::string key) {
    size_t h = std::hash<std::string>{}(key) % size;
    for(int i = 0; i < size; i++) {
        if (!items[h].empty()){
            if (items[h][0].key == key) {
                break;
            }
            h = (h + 1) % size;
        }
    }
    return &items[h];
}

template<class T>
int TableHash<T>::get_count()
{
    return count;
}
```

Index