Задание 3

Выполнил:	Захаревич Николай Сергеевич
Имя входного файла:	Консоль
Имя выходного файла:	Консоль

Реализовать бинарное дерево поиска (https://en.wikipedia.org/wiki/Binary$_s earch_t ree$), $.STL$.

Формат входного файла

Входной файл содержит описание операций с деревом, их количество не превышает 100. В каждой строке находится одна из следующих операций: • insert x — добавить в дерево ключ x. Если ключ x есть в дереве, то ничего делать не надо • delete x — удалить из дерева ключ x. Если ключа x в дереве нет, то ничего делать не надо • exists x — если ключ x есть в дереве выведите «true», если нет «false» • next x — выведите минимальный элемент в дереве, строго больший x, или «none» если такого нет • prev x — выведите максимальный элемент в дереве, строго меньший x, или «none» если такого нет В дерево помещаются и извлекаются только целые числа, не превышающие по модулю 109

Формат выходного файла

Выведите последовательно результат выполнения всех операций exists, next, prev. Следуйте формату выходного файла из примера.

Пример

Консоль	Консоль
insert 2 insert 5 insert 3 exists 2 exists 4 next 4 prev 4 delete 5 next 4 prev 4	true false 5 3 none 3

lab3.cpp

```
#include <iostream>
#include <vector>

template<class T, class Compare = std::less<T>>
class BST {
public:
  struct Node {
    T value_;
    int height_;
    int diff_;
    Node *left = nullptr;
    Node *right = nullptr;
    Node *parent = nullptr;

    Node() {
      height_ = 1;
```

```
    diff_ = 0;
  }
  Node(T value) : value_(value) { Node(); }
  Node(const Node &other) {
    value_ = other.value_;
    height_ = other.height_;
    diff_ = other.diff_;
  }
};

class iterator {
private:
  Node *pointer_;
public:
  iterator() { pointer_ = nullptr; }
  iterator(Node *pointer) : pointer_(pointer) {}
  iterator(const iterator *other) : pointer_(other -> pointer_) {}
  iterator(const iterator &other) : pointer_(other.pointer_) {}

  iterator &operator=(const iterator &other) { this -> iterator(); new (this) itera
  iterator &operator=(const std::nullptr_t &other) { pointer_ = nullptr; return *t

  bool operator!=(iterator const& other) const { return pointer_ != other.pointer_
  bool operator==(iterator const& other) const { return pointer_ == other.pointer_
  T operator*() const { return pointer_ -> value_; }

  iterator& operator++() { pointer_ = next_(pointer_, pointer_-> value_); return *
  iterator operator++(int) const { iterator temp(this); ++(*this); return temp; }
  iterator& operator--() { pointer_ = prev_(pointer_, pointer_-> value_); return *
  iterator operator--(int) const { iterator temp(this); --(*this); return temp; }

  Node *getPointer() const { return pointer_; }
};

typedef const iterator const_iterator;

BST();
BST(const BST &other);
BST &operator=(const BST &other);
~BST();

template<class InputIt>
void assign(InputIt first, InputIt second) {
  BST<T, Compare>::iterator it = begin_;
  for (InputIt i = first; i != second; ++i, ++it) {
    it = new Node(*i);
  }
  while (root_ -> diff_ > 2 || root_ -> diff_ < 2) {
    refresh_(root_);
  }
}
```

```
  iterator begin() { return begin_; }
  iterator end() { return end_; }
  const_iterator begin() const { return begin_; }
  const_iterator end() const { return end_; }

  void insert(const T &value);
  void remove(const T &value);

  void print() const;

  bool exists(const T &x);
  size_t size() const;
private:
  Node *root_;
  size_t size_;
  iterator begin_;
  iterator end_;
  Compare comp;

  void correctHeight(Node *node);
  void smallLeft(Node *node);
  void smallRight(Node *node);
  void bigLeft(Node *node);
  void bigRight(Node *node);
  void refresh_(Node *node);
  Node *find_(Node *node, const T &value);
  static Node *next_(Node *node, const T &value);
  static Node *prev_(Node *node, const T &value);
  void insert_(Node *node, const T &value, Node *parent);
  void remove_(Node *node, const T &value);
  void print_(Node *node) const;
  Node *copy_(Node *dest, Node *parent, Node *src);
};

template<class T, class Compare>
BST<T, Compare>::BST() {
  root_ = nullptr;
  size_ = 0;
  begin_ = nullptr;
  end_ = nullptr;
}

template<class T, class Compare>
BST<T, Compare>::BST(const BST &other) {
  root_ = copy_(root_, nullptr, other.root_);
  size_ = other.size_;
  begin_ = other.begin_;
  end_ = other.end_;
}

template<class T, class Compare>
BST<T, Compare> &BST<T, Compare>::operator=(const BST &other) {
```

```
  this -> ~BST();
  new (this) BST(other);
  return *this;
}

template<class T, class Compare>
BST<T, Compare>::~BST() {
  delete [] root_;
}

template<class T, class Compare>
void BST<T, Compare>::insert(const T &value) {
  insert_(root_, value, nullptr);
}

template<class T, class Compare>
void BST<T, Compare>::remove(const T &value) {
  Node *node = find_(root_, value);
  remove_(node, value);
}

template<class T, class Compare>
bool BST<T, Compare>::exists(const T &value) {
  Node *node = find_(root_, value);
  return node != nullptr;

}

template<class T, class Compare>
size_t BST<T, Compare>::size() const {
  return size_;
}

template<class T, class Compare>
void BST<T, Compare>::print() const {
  std::cout << "size = " << size_ << "\n";
  print_(root_);
}

template<class T, class Compare>
void BST<T, Compare>::correctHeight(Node *node) {
  node -> height_ = std::max(
      node -> left == nullptr ? 0 : node -> left -> height_,
      node -> right == nullptr ? 0 : node -> right -> height_
  ) + 1;

  node -> diff_ = (node -> left == nullptr ? 0 : node -> left -> height_) -
          (node -> right == nullptr ? 0 : node -> right -> height_);
}

template<class T, class Compare>
void BST<T, Compare>::smallLeft(Node *node) {
```

```
  Node *temp = node -> right;

  node -> right = temp -> left;
  if (temp -> left != nullptr) {
    temp -> left -> parent = node;
  }

  temp -> left = node;
  if (node -> parent == nullptr) {
    root_ = temp;
    temp -> parent = nullptr;
  } else {
    if (node == node -> parent -> left)
      node -> parent -> left = temp;
    else node -> parent -> right = temp;
    temp -> parent = node -> parent;
  }
  node -> parent = temp;

  correctHeight(node);
  correctHeight(temp);
}

template<class T, class Compare>
void BST<T, Compare>::smallRight(Node *node) {
  Node *temp = node -> left;

  node -> left = temp -> right;
  if (temp -> right != nullptr) {
    temp -> right -> parent = node;
  }

  temp -> right = node;
  if (node -> parent == nullptr) {
    root_ = temp;
    temp -> parent = nullptr;
  } else {
    if (node == node -> parent -> left)
      node -> parent -> left = temp;
    else node -> parent -> right = temp;
    temp -> parent = node -> parent;
  }
  node -> parent = temp;

  correctHeight(node);
  correctHeight(temp);
}

template<class T, class Compare>
void BST<T, Compare>::bigLeft(Node *node) {
  smallRight(node -> right);
  smallLeft(node);
```

```
}

template<class T, class Compare>
void BST<T, Compare>::bigRight(Node *node) {
  smallLeft(node -> left);
  smallRight(node);
}

template<class T, class Compare>
void BST<T, Compare>::refresh_(Node *node) {
  if (node == nullptr)
    return;

  correctHeight(node);

  if (node -> diff_ == -2) {
    node -> right -> diff_ == 1 ? bigLeft(node) : smallLeft(node);
  }

  if (node -> diff_ == 2) {
    node -> left -> diff_ == -1 ? bigRight(node) : smallRight(node);
  }

  correctHeight(node);
  refresh_(node->parent);
}

template<class T, class Compare>
Node *BST<T, Compare>::find_(Node *node, const T &value) {
  if (node == nullptr) {
    return node;
  }
  if (node -> value_ == value)
    return node;

  comp(node -> value_, value) ? find_(node -> right, value) : find_(node -> left, va
}

template<class T, class Compare>
Node *BST<T, Compare>::next_(Node *node, const T &value) {
  Node *ret = nullptr;
  while (node != nullptr) {
    if (value >= node -> value_) {
      node = node -> right;
    } else {
      ret = node;
      node = node -> left;
    }
  }
  return ret;
}
```

```
template<class T, class Compare>
Node *BST<T, Compare>::prev_(Node *node, const T &value) {
  Node *ret = nullptr;
  while (node != nullptr) {
    if (value <= node -> value_) {
      node = node -> left;
    } else {
      ret = node;
      node = node -> right;
    }
  }
  return ret;
}

template<class T, class Compare>
void BST<T, Compare>::insert_(Node *node, const T &value, Node *parent) {
  if (node != nullptr && value == node -> value_)
    return;

  if (parent == nullptr && node == nullptr) {
    root_ = new Node(value);
    begin_ = root_;
    end_ = root_;
    size_++;
    refresh_(node);
    return;
  }

  if (node == nullptr) {
    node = new Node(value);
    comp(node -> value_, parent -> value_) ? parent -> left = node : parent -> right
    node -> parent = parent;
    if (begin_.getPointer() == parent && node == parent -> left) {
      begin_ = node;
    } else if (end_.getPointer() == parent && node == parent -> right) {
      end_ = node;
    }
    size_++;
    refresh_(node);
    return;
  }

  comp(node -> value_, value) ? insert_(node -> right, value, node) : insert_(node ->

}

template<class T, class Compare>
void BST<T, Compare>::remove_(Node *node, const T &value) {
  if (node == nullptr)
    return;

  if (node -> left == nullptr && node -> right == nullptr) {
```

```
    if (node -> parent == nullptr) {
      root_ = nullptr;
      begin_ = nullptr;
      end_ = nullptr;
      return;
    }

    node == node -> parent -> right ? node -> parent -> right = nullptr : node -> pa
    if (begin_.getPointer() == node) {
      begin_ = node -> parent;
    }
    if (end_.getPointer() == node) {
      end_ = node -> parent;
    }
    refresh_(node->parent);
    return;
  }

  if (node -> left != nullptr && node -> right != nullptr) {
    Node *nextNode = next_(node -> right, value);
    node -> value_ = nextNode -> value_;
    remove_(nextNode, value);
    size_--;
    return;
  }

  if (node -> left != nullptr) {
    if (node -> parent == nullptr) {
      root_ = node -> left;
      node -> left -> parent = nullptr;
      if (end_.getPointer() == node) {
        end_ = root_;
      }
      return;
    }
    if (node == node -> parent -> right) {
      node -> parent -> right = node -> left;
      node -> left -> parent = node -> parent;
      if (end_.getPointer() == node) {
        end_ = node -> left;
      }
    } else {
      node -> parent -> left = node -> left;
      node -> left -> parent = node -> parent;
    }

  } else {
    if (node -> parent == nullptr) {
      root_ = node -> right;
      node -> right -> parent = nullptr;
      if (begin_.getPointer() == node) {
        begin_ = root_;
```

```
      }
      return;
    }
    if (node == node -> parent -> right) {
      node -> parent -> right = node -> right;
      node -> right -> parent = node -> parent;
    } else {
      node -> parent -> left = node -> right;
      node -> right -> parent = node -> parent;
      if (end_.getPointer() == node) {
        end_ = node -> right;
      }
    }
  }
  size_--;
  refresh_(node -> parent);
}

template<class T, class Compare>
void BST<T, Compare>::print_(Node *node) const {
  if (node == nullptr)
    return;
  std::cout << node -> value_ << " - value; " << node -> height_ << " - height; " <<
  print_(node->left);
  print_(node->right);
}

template<class T, class Compare>
Node *BST<T, Compare>::copy_(Node *dest, Node *parent, Node *src) {
  if (src == nullptr) {
    return nullptr;
  }
  dest = new Node(*src);
  dest -> parent = parent;
  dest -> left = copy_(dest -> left, dest, src -> left);
  dest -> right = copy_(dest -> right, dest, src -> right);
  return dest;
}

int main() {
  BST<int, std::less<int>> tree, tree2;
  std::string q;
  int x;
  BST<int, std::less<int>>::iterator b, e;

  tree.insert(10);
  tree.insert(20);
  tree.insert(15);
  tree.insert(14);

  for (BST<int, std::less<int>>::iterator it = tree.begin(); it != tree.end(); ++it)
    std::cout << *it << " ";
```

```
    }

    return 0;
}
```