Московский Авиационный Институт
(Национальный Исследовательский Университет)

Кафедра: 806 «Вычислительная математика и программирование»
Факультет: «Информационные технологии и прикладная математика»
Дисциплина: «Объектно-ориентированное программирование»

# Лабораторная работа №8.

Группа: М8О – 104Б-16
Студент: Чекушкин Денис Игоревич
Преподаватель: Поповкин Александр Викторович
Вариант: №18

Москва
2017

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

• Знакомство с параллельным программированием в C++.

ЗАДАНИЕ

Используя структуры данных, разработанные для лабораторной работы No6 (контейнер первого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класса-контейнера .

Необходимо разработать два вида алгоритма:

• Обычный, без параллельных вызовов.

• С использованием параллельных вызовов. В этом случае, каждый рекурсивный вызов сортировки должен создаваться в отдельном потоке.

Для создания потоков использовать механизмы:

• future

• packaged_task/async

Для обеспечения потоко-безопасности структур данных использовать:

• mutex

• lock_guard

**Теория**

**Параллельные вычисления** — способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно). Термин охватывает совокупность вопросов параллелизма в программировании, а также создание эффективно действующих аппаратных реализаций. Теория параллельных вычислений составляет раздел прикладной теории алгоритмов.

Существуют различные способы реализации параллельных вычислений. Например, каждый вычислительный процесс может быть реализован в виде процесса операционной системы, либо же вычислительные процессы могут представлять собой набор потоков выполнения внутри одного процесса ОС. Параллельные программы могут физически исполняться либо последовательно на единственном процессоре — перемежая по очереди шаги выполнения каждого вычислительного процесса, либо параллельно — выделяя каждому вычислительному процессу один или несколько процессоров (находящихся рядом или распределённых в компьютерную сеть).

**Быстрая сортировка относится к алгоритмам «разделяй и властвуй».**

Алгоритм состоит из трёх шагов:

1. Выбрать элемент из массива. Назовём его опорным.
2. *Разбиение*: перераспределение элементов в массиве таким образом, что элементы

меньше опорного помещаются перед ним, а больше или равные после.

3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один или отсутствуют элементы.

btree.cpp

```
template <class T>
Btree<T>::Btree()
{
m_root=nullptr;
m_size=0;
}
template <class T>
Iterator<BTreeItem<T>, T> Btree<T>::begin() const
{
    return Iterator<BTreeItem<T>, T>(m_root);
}
template <class T>
Iterator<BTreeItem<T>, T> Btree<T>::end() const
{
    std::shared_ptr<T> cur=m_root;
    return Iterator<BTreeItem<T>, T>(cur);
}
template <class T>
std::shared_ptr<T> Btree<T>::front() const
{
 return m_root->getFigure();
}
template <class T>
unsigned int Btree<T>::size() const
{
 return m_size;
}
template <class T>
void insert_helper(std::shared_ptr<BTreeItem<T>> m_root,const std::shared_ptr<T>&
figure)
{
 std::shared_ptr<BTreeItem<T>> node = m_root;
```

```
      if(*figure == *(node->getFigure())){ std::cout<<*figure<<"\n"; std::cout<<*(node->getFigure())<<"\n"; std::cout<<"БЫЛО"<<"\n";}

      if(*figure < *(node->getFigure())) {

            if(node->getLeft()==nullptr)

                  {

                  std::shared_ptr<BTreeItem<T>> newl=std::make_shared<BTreeItem<T>>(figure);

                  node->setLeft(newl);

                  }

            else

                  {

   if(*figure == *(node->getFigure())){ std::cout<<*figure<<"\n"; std::cout<<*(node->getFigure())<<"\n"; std::cout<<"БЫЛО"<<"\n";return;}

                  std::shared_ptr<BTreeItem<T>> left = node->getLeft();

                  insert_helper(left,figure);

                  }

            }

      else if(*figure > *(node->getFigure()))  {

            if(node->getRight()==nullptr)

                  {

                  std::shared_ptr<BTreeItem<T>> newr=std::make_shared<BTreeItem<T>>(figure);

                  node->setRight(newr);

                  }

            else

                  {

   if(*figure == *(node->getFigure())){ std::cout<<*figure<<"\n"; std::cout<<*(node->getFigure())<<"\n"; std::cout<<"БЫЛО"<<"\n";return;}

                  std::shared_ptr<BTreeItem<T>> right = node->getRight();

                  insert_helper(right,figure);

                  }

      }

   }


   template <class T>

   void Btree<T>::bstInsert(const std::shared_ptr<T>& figure)

   {

   m_size++;
```

```
   if(m_root==nullptr)
    {
    m_root = std::make_shared<BTreeItem<T>>(figure);
    }
   else
   {insert_helper(m_root,figure);
   }
   }
   template <class T>
   void rinsert_helper(std::shared_ptr<BTreeItem<T>> m_root,const std::shared_ptr<T>&
figure)
   {
    std::shared_ptr<BTreeItem<T>> node = m_root;
      if(*figure < *(node->getFigure())) {
            if(node->getRight()==nullptr)
                  {
                  std::shared_ptr<BTreeItem<T>>
newr=std::make_shared<BTreeItem<T>>(figure);
                  node->setRight(newr);
                  }
            else
                  {
                  std::shared_ptr<BTreeItem<T>> right = node->getRight();
                  rinsert_helper(right,figure);
                  }
            }
      else if(*figure > *(node->getFigure()))  {
            if(node->getLeft()==nullptr)
                  {

                  std::shared_ptr<BTreeItem<T>>
newl=std::make_shared<BTreeItem<T>>(figure);
                  node->setLeft(newl);
                  }
            else
                  {
                  std::shared_ptr<BTreeItem<T>> left = node->getLeft();
                  rinsert_helper(left,figure);
                  }
```

```
            }
    }
    template <class T>
    void Btree<T>::bstRInsert(const std::shared_ptr<T>& figure)
    {
    m_size++;
    if(m_root==nullptr)
     {
     m_root = std::make_shared<BTreeItem<T>>(figure);
     }
    else
    {rinsert_helper(m_root,figure);
    }
    }


    template <class T>
    void remove_helper(std::shared_ptr<BTreeItem<T>> & item, const std::shared_ptr<T>&
figure)
    {
        std::shared_ptr<BTreeItem<T>> repl = nullptr, parent = nullptr, tmp = item;
        while (tmp != nullptr && !(*(tmp->getFigure()) == *figure))
        {
            parent = tmp;
            if (*figure < *(tmp->getFigure()))
            {
                tmp = tmp->getLeft();
            }
            else
            {
                tmp = tmp->getRight();
            }
        }
        if (tmp == nullptr)
        {
            return;
        }
```

```
if (tmp->getLeft() != nullptr && tmp->getRight() == nullptr)
{
    if (parent != nullptr)
    {
        if (parent->getLeft() == tmp)
            parent->setLeft(tmp->getLeft());
        else
            parent->setRight(tmp->getRight());
    }
    else
        item = tmp->getLeft();

}
else if (tmp->getLeft() == nullptr && tmp->getRight() != nullptr)
{
    if (parent != nullptr)
    {
        if (parent->getLeft() == tmp)
            parent->setLeft(tmp->getLeft());
        else
            parent->setRight(tmp->getRight());
    }
    else
        item = tmp->getRight();

}

else if (tmp->getLeft() != nullptr && tmp->getRight() != nullptr)
{
    repl = tmp->getRight();

    if (repl->getLeft() == nullptr) {
        tmp->setRight(repl->getRight());}

    else
    {
        while (repl->getLeft() != nullptr)
```

```
            {
                parent = repl;
                repl = repl->getLeft();
            }

            parent->setLeft(repl->getRight());
        }
 tmp->setRoot(repl->getFigure());

    }
    else
    {
        if (parent != nullptr)
        {
            if (parent->getLeft() == tmp)
                parent->setLeft(nullptr);
            else
                parent->setRight(nullptr);
        }
        else
            item = nullptr;
    }
    return;
}

template <class T>
void Btree<T>::bstRemove(const std::shared_ptr<T>& figure)
{
 m_size--;
    remove_helper(m_root, figure);
}
template <class T>
void print2(const std::shared_ptr<BTreeItem<T>> &node)
{
    if(node == nullptr) return;
    print2(node->getLeft());
    std::cout << *(node->getFigure())<< std::endl;
    print2(node->getRight());
```

```
}
template <class B>
std::ostream& operator<<(std::ostream& os, const Btree<B> &btree)
{
std::shared_ptr<BTreeItem<B>> item=btree.m_root;
print2(item);
return os;
}
template <class T>
void Btree<T>::sort()
{
std::cout<<"Not par"<<std::endl;
 sortHelper(*this, false);
}
template <class T>
void Btree<T>::sortParallel()
{
std::cout<<"par"<<std::endl;
 sortHelper(*this, true);
}
template <class T>
void Btree<T>::sortHelper(Btree<T>& b, bool isParallel)
{
 if (b.size() <= 1){
 return;
 }
 Btree<T> left;
 Btree<T> right;
 std::shared_ptr<T> mid = b.front();
 b.bstRemove(b.front());
 while (b.size() > 0)
 {
    std::shared_ptr<T> item = b.front();
    b.bstRemove(b.front());
    if (item->area() < mid->area()){
           left.bstInsert(item);
    }
    else{
```

```
            right.bstInsert(item);
        }
    }

    if (isParallel)
    {
        std::future<void> leftFu = sortParallelHelper(left);
        std::future<void> rightFu = sortParallelHelper(right);

        leftFu.get();
        rightFu.get();
    }
    else
    {
        sortHelper(left, isParallel);
        sortHelper(right, isParallel);
    }
    b.bstInsert(mid);
    while (left.size() > 0)
    {
        b.bstRInsert(left.front());
        left.bstRemove(left.front());
    }
    while (right.size() > 0)
    {
        b.bstRInsert(right.front());
        right.bstRemove(right.front());
    }
}
template <class T>
std::future<void> Btree<T>::sortParallelHelper(Btree<T>& q)
{
    auto funcObj = std::bind(&Btree<T>::sortHelper, this, std::ref(q), true);
    std::packaged_task<void()> task(funcObj);
    std::future<void> res(task.get_future());
    std::thread th(std::move(task));
    th.detach();
```

```
std::cout<<"thread_id:"<<std::this_thread::get_id()<<std::endl;
 return res;
}
```

Выводы: в данной лабораторной работе я получил навыки работы с параллельным программированием. Добавил два алгоритма быстрой сортировки — обычная и с параллельными вызовами.

https://github.com/israelcode/oop/tree/master/sem2/lab8