Implementation of ADS Linked List Via Smart Pointers

Ivaylo Donchev, Emilia Todorova
Department of Information Technologies, Faculty of Mathematics and Informatics
St Cyril and St Methodius University of Veliko Turnovo
Veliko Turnovo, Bulgaria

***Abstract*—Students traditionally have difficulties in implementing abstract data structures (ADS) in C++. To a large extent, these difficulties are due to language complexity in terms of memory management with raw pointers – the programmer must take care of too many details to provide reliable, efficient and secure implementation. Since all these technical details distract students from the essence of the studied algorithms, we decided to use in the course in DSA (Data Structures and Algorithms) an automated resource management, provided by the C++ standard ISO/IEC 14882:2011. In this work we share experience of using smart pointers to implement linked lists and discuss pedagogical aspects and effectiveness of the new classes, compared to the traditional library containers and implementation via built-in pointers.**

Keywords—abstract data structures; C++; smart pointers; teaching

I. Introduction

From the C language we know that pointers are important but are a source of trouble. One reason to use pointers is to have reference semantics outside the usual boundaries of scope [1]. However, it can be quite difficult to ensure that the life of the pointer and the life of the object to which it points will coincide, especially in cases where multiple pointers point to the same object. Such a situation we have as if an object must participate in multiple collections – each of them must provide a pointer to this object. To make everything correct we need to ensure:

- When destroying one of the pointers, take care that there are no dangling pointers or multiple deletions of the pointed object;
- When you destroying the last reference to an object, to destroy the very object in order not to allow resource leaks;
- Do not allow null-pointer dereference – a situation in which a null pointer is used as if it points to a real object.

We must have in mind such details if we want to accomplish dynamic implementation of ADS and often the time for this exceeds the time remaining to comment the structures and operations on them. Moreover, there are rare cases where we have a working implementation of a structure with carefully designed interface and methods written according to the best methodologies, but we identify gaps in the management of memory only when the fall in non-trivial situations such as copying large structures, transfer of items from one structure to another, or destruction of a large recursive structure. For each class representing ADS the programmer must also provide characteristic operations as well as correctly working copy and move semantics, exception handling, construction and destruction. This requires both time and expertise in programming at a lower level. The teacher will have to choose between emphasizing on language-specific features and quality of implementation or to compromise with them and to spend more time on algorithms and data structures. In an attempt to escape from this compromise, we decided to change the content of our CS2 course in DSA, and include the study of smart pointers for resource management and with their help to simplify implementations of ADS, and avoid explicit memory management which is widely recognized as error-prone [2].

Our initial hypothesis was that a correct and effective implementation is possible, which could relieve our work in two directions:

- Operations with whole structures: not having to write destructors, copy and move constructors and copy and move assignment operators;
- Shorter and easier to understand implementation of operations with elements of structures – include (insert element), search, delete.

II. Development of Language Tools for Dynamic Memory Management

Before introducing of `new` and `delete` for work with dynamic memory, inherited from the C language functions `malloc, calloc, realloc` and `free` are used, which are still available in C++ by including the header file `<cstdlib>`.

```
Data * d = (Data *)  malloc(sizeof Data);
// ...
free(d);
```

Memory blocks allocated through these functions are not necessarily compatible with those returned by `new`, so each must be handled with its own set of functions or operations. The problems here are related to unnecessary type conversions and error-prone size calculations (with `sizeof`).

Introduction of `new` and `delete` operators simplifies the syntax, but does not solve all problems. Especially in applications that manipulate complicated linked data structures, it may be difficult to identify the last use of an object. Mistakes

lead to either duplicate de-allocations and possible security holes, or memory leaks [2]. We illustrate this with an example: Let p1, p2, p3 and p4 are pointers to objects of the class Person.

```
vector<Person*> family { p1, p2, p3, p4 };
vector<Person*> kids { p3, p4 };
//...
delete p3;
print(family);//family contains dangling ptr
if (kids.empty()) return 0;     //early return
//...
delete p1;
delete p2;
delete p3;        // double deletion
delete p4;
```

The two vectors – family and kids contain pointers to shared objects – p3 and p4. Deleting the object pointed to by p3 leads to the emergence of "dangling" pointers in the two vectors because they cannot "understand" that the referred object is deleted. All the potential problems with locally defined naked pointers include:

- **Leaked objects**: memory allocation with new can cause (though rarely) an exception which is not handled. It is also possible function execution to be terminated by another raised exception and the allocated with new memory to remain unreleased (it is not exceptions safety). Avoiding such resource leak usually requires that a function catches all exceptions. To handle the deletion of the object properly in case of an exception, the code becomes complicated and cluttered. This is a bad programming style and should be avoided because it is also error prone. Similar situation we have when function execution is terminated by premature return statement based on some condition (early return);
- **Premature deletion**: we delete an object that has some other pointer to and later use that other pointer.
- **Double deletion**: we are not insured against an attempt to re-delete an object (in the example with vectors the one pointed by p3).

One way to circumvent these problems is to simply use a local variable, instead of a pointer, but if we insist to use pointer semantics, the usual approach to overcome such problems is the use of "smart pointers". Their "intelligence" is expressed in that they "know" whether they are the last reference to the object and use this knowledge to destroy the object only when its "ultimate owner" is to be destroyed. We can consider that a "smart pointer" is RAII (Resource Acquisition Is Initialization) modeled class that manages dynamically allocated memory. It provides the same interfaces that ordinary pointers do (*, ->). During its construction it acquires ownership of a dynamic object in memory and deallocates that memory when goes out of scope. In this way, the programmer does not need to care himself for the management of dynamic memory.

For the first time the standard C++98 introduces a single type of smart pointer – auto_ptr which provides specific and focused transfer-of-ownership semantics. auto_ptr is most charitably characterized as a valiant attempt to create a unique_ptr before C++ had move semantics. auto_ptr is now deprecated, and should not be used in new code. It works well in trivial situations:

```
int main(){
   try {
          auto_ptr<X> ap1(new X(1122));
          // _div() throws exception
          cout << _div(5, 0) << endl;
          ap1->print();
   }
   catch (exception& e){
          cerr << e.what() << endl;
   }
}
```

Template auto_ptr holds a pointer to an object obtained via new and deletes that object when it itself is destroyed (such as when leaving block scope). Function _div() returns the quotient of its arguments and causes an exception at zero divisor. Thus, in main() an exception occurs and the operator ap1->print() will not be executed, but still the memory that ap1 manages will be properly released. This is due to the stack unwinding, which occurs in exception processing – all local objects defined in the try block are destroyed, the destruction of ap1 releases the associated memory for the object of class X. Here auto_ptr is "smart" enough, but it appears that the problems entailed outweigh the benefit from it:

- copying and assignment among smart pointers transfers ownership of the manipulated object as well. That is, by default move assignment and move construction is carried out. Such is the situation with passing of auto_ptr as a parameter of the function:

```
void foo(auto_ptr<X> ap2){
   ap2->print();
}
int main(){
   auto_ptr<X> ap1(new X(1122));
   foo(ap1);
   ap1->print();  //oops! ap1 is empty
}
```

After completion of foo() the memory allocated in the initialization of ap1 and then passed to ap2 will be released (at the destruction of ap2) and will not be given back to ap1. This will result in an error when trying to use the contents of ap1 (it is already a dangling pointer).

We have a similar result in the following situations:

```
auto_ptr<X> ap3(ap1);     //move construction
ap1->print();             //oops! ap1 is empty
auto_ptr<X> ap4;
ap4 = ap3;                //move assignment
ap3->print();             //oops! ap3 is empty
```

In constructing ap3 it acquires the resource managed by ap1. This is called *copy elision*. In some cases this is a very useful technique (eg to avoid unnecessary copying when the function returns local object by value – compilers do this automatically).

The auto_ptr provides a semantics of strict ownership. auto_ptr owns the object it holds a pointer to. Copying an auto_ptr copies the pointer and transfers ownership to the destination. If more than one auto_ptr owns the same object at the same time, the behavior of the program is undefined.

- auto_ptr can not be used for an array of objects. When auto_ptr goes out of scope, delete runs on its associated memory block. This works if we have a single object, not an array of objects that must be destroyed with delete [].
- because auto_ptr does not provide shared-ownership semantics, it can not even be used with Standard Library containers like vector, list, map.

Although auto_ptr is now officially deprecated by the standard ISO/IEC, 2011 [4], in Visual Studio 2013 can have declarations like:

```
auto_ptr<vector<int>> apv {new vector<int>{ 1
    2, 3, 4, 5 } };
vector<auto_ptr<int>> v;
```

The reason for this is the famous backward compatibility feature of C ++.

Practice shows that to overcome (or at least limit) problems as described above it is not sufficient to use only one "smart pointer" class. Smart pointers can be smart in some aspects and carry out various priorities, as they have to pay the price for such intelligence [1], p. 76. Note that even now, with several types of smart pointers their misuse is possible and programming of wrong behavior.

In the standard (ISO/IEC, 2011) instead of auto_ptr several different types of smart pointers are introduced (also called Resource Management Pointers) [5], modeling different aspects of resource management. The idea is not new – it formally originates from [6] and was originally implemented in the Boost library and only in 2011 became a part of the Standard Library. The basic, top-level and general-purpose smart pointers are unique_ptr and shared_ptr. They are defined in the header the file <memory>.

Unfortunately, excessive use of new (and pointers and references) seems to be an escalating problem. However, when you really need pointer semantics, unique_ptr is a very lightweight mechanism, with no additional costs compared to the correct use of built-in pointer [5], p. 113. The class unique_ptr is designed for pointers that implement the idea of exclusive (strict) ownership, what was intended auto_ptr to do. It ensures that at any given time only one smart pointer may point to the object. As a result, an object gets destroyed automatically when its unique_ptr gets destroyed. However, transfer of ownership is permitted. This class is particularly useful for avoiding leak of resources such as missed delete calls for dynamic objects or when exception occurs while an object is being created. It has much the same interface as an ordinary pointer. Operator * dereferences the object to which it points, whereas operator -> provides access to a member if the object is an instance of a class or a structure. Unlike ordinary pointers, smart pointer arithmetic is not possible, but specialists consider this an advantage, because it is known that pointer arithmetic is a source of trouble. unique_ptr uses include passing free-store allocated objects in and out of functions (rely on move semantics to make return simple and efficient):

```
// make Person object and give it to a
   unique_ptr
unique_ptr<Person> make_Person(
   const string & name, int year)
{
   // ... check Person, etc. ...
   return unique_ptr<Person>{new Person{name,
   year}};
}
// ..................
auto pp = make_Person("Ivaylo", 1971);
pp->print();
```

For such situations std::move() will be automatically executed for the return value (under the new rules in C++11). Copying or assignment between unique pointers is impossible if we use the ordinary copy semantics. However, we can use the move semantics. In that case, the constructor or assignment operator transfers the ownership to another unique pointer.

The typical use of unique_ptr includes:

- ensuring safe use of dynamically allocated memory through the mechanism of exceptions (exception safety);
- transfer of ownership of dynamically allocated memory to function (via parameter);
- returning dynamically allocated memory – the function returns a pointer to the allocated memory (unique_ptr) ;
- storing pointers in a container.

A point of interest is the situation when unique_ptr is passed as a parameter of a function by rvalue reference, created by std::move(). In this case the parameter of the called function acquires ownership of unique_ptr. If then this function does not pass ownership again, the object will be destroyed at its completion:

```
template <typename T>
void f(unique_ptr<T> x)
{
   cout << *x << endl;
}
int main()
{
 unique_ptr<string> up{new string{"Ivaylo"}};
 f(move(up));     // up became empty
 if (up) cout << *up << endl;
   else cout << "empty pointer" << endl;
}
```

Using a unique pointer, as a member of a class may also be useful for avoiding leak of resources. By using unique_ptr, instead of built-in pointer there is no need of a destructor because the object will be destroyed while destroying the member concerned. In addition unique_ptr prevents leak of resources in case of exceptions which occur during initialization of objects – we know that destructors are called

only if any construction has been completed. So, if an exception occurs within the constructor, destructors will be executed for objects that have been already fully constructed. As a result we can get outflow of resources for classes with multiple raw pointers, if the first construction with new is successful, but the second fails.

Simultaneous access to an object from different points in the program can be provided through ordinary pointers and references, but we already commented on the problems associated with their use. Often we have to make sure that when the last reference to an object is deleted, the object itself will be destroyed as well (which usually implies garbage collection operations – to deallocate memory and other resources).

The class `shared_ptr` implements the concept of shared ownership. Many smart pointers can point to the same object, and the object and its associated resources are released when the last reference is destroyed. The last owner is responsible for the destroying. To perform this task in more complex scenarios auxiliary classes `weak_ptr`, `bad_weak_ptr`, `enable_shared_from_this` are provided.

The class `shared_ptr` is similar to a pointer with counter of the number of sharings (reference counter), which destroys the pointed object when this counter becomes zero. Imagine `shared_ptr` as a structure of two pointers – one to the object and one to the counter of sharings.

Shared pointer can be used as an ordinary pointer – to assign, copy and compare, to have access to the pointed object via the operations `*` and `->`. We have a full range of copy and move constructions and assignments. Comparison operations are applied to stored pointers (usually the address of the owned object or `nullptr` if none). `shared_ptr` does not provide index operation. For `unique_ptr` a partial specialization for arrays is available that provides `[]` operator, along with `*` and `->`. This is due to the fact that `unique_ptr` is optimized for efficiency and flexibility. Access to the elements of the owned by `shared_ptr` array can be provided through the indices of the internal stored pointer, encapsulated by `shared_ptr` (and accessible through the member function `get()`).

We already discussed the problems with dangling pointers, which arise while build-in pointers are stored in containers. Now we will show how the use of `shared_ptr` avoids them. Consider the same situation with vectors of `Person` objects – `family` and `kids`:

In the function `main()` we have 4 shared pointers, to manipulative dynamic objects of `Person`:

```
auto sp1=make_shared<Person>("Ivaylo", 1971);
auto sp2=make_shared<Person>("Doroteya", 1977);
auto sp3=make_shared<Person>("Victoria", 2002);
auto sp4=make_shared<Person>("Peter", 2009);
```

and two vectors of such pointers in which objects are duplicated:

```
vector<shared_ptr<Person>> sp_family{sp1,
   sp2, sp3, sp4};
vector<shared_ptr<Person>> sp_kids{sp3, sp4};
```

There is a single copy of each object of `Person`. The number of references to the children is 3 - one in each vector and the one of `sp3` (or `sp4`).

The name change

```
sp3->set_name("Victoria Doncheva");
```

immediately affects both vectors. Release of `sp3` by `reset()` does not lead to destruction of the object `Person {"Victoria", 2002}`, in opposit to build-in pointers.

Of course, if you like, you can always make a mess. If you initialize a build-in pointer with the owned by `shared_ptr` internal pointer, and then deallocate memory by this raw pointer:

```
Person* p = sp3.get();
delete p;
```

A problem with reference-counted smart pointers is that if there is a ring, or cycle, of objects that have smart pointers to each other, they keep each other "alive" – they will not get deleted even if no other objects are pointing to them from "outside" the ring. Such a situation often occurs in implementations of recursive data structures. C++11 includes a solution: "weak" smart pointers: these only "observe" an object but do not influence its lifetime. A ring of objects can point to each other with `weak_ptr`s, which point to the managed object but do not keep it in existence. Like raw pointers, the weak pointers do not keep the pointed-to object "alive". The cycle problem is solved. However, unlike raw pointers, the weak pointers "know" whether the pointed-to object is still there or not and can be interrogated about it, making them much more useful than a simple raw pointer would be.

In practice often happens a situation when we hesitate which version of a smart pointer to use – `unique_ptr` or `shared_ptr`. The advice is to prefer `unique_ptr` by default, and we can always later move-convert to `shared_ptr` if needed. There are three main reasons for this [7]:

- try to use the simplest semantics that are sufficient;
- a `unique_ptr` is more efficient than a `shared_ptr`. A `unique_ptr` does not need to maintain reference count information and a control block under the covers, and is designed to be just about as cheap to move and use as a raw pointer;
- starting with `unique_ptr` is more flexible and keeps your options open.

In our case, however, we had from the very beginning to start with `shared_ptr`, because being recursive by definition, the data structures that we tried to implement with smart pointers can not do without shared ownership.

III. Implementation of Lists

In the course in Data Structures and Algorithms (DSA) we use dynamically implemented singly linked and doubly linked lists and based on them specializations for other ADS – stack, queue, deque. We develop a template class `List` with an interface similar to the following:

```
//singly linked list with built-in pointers
```

```
template <typename T>
class List {
private:
   struct Node {
      T key;
      Node* next;
      Node():key(),next(nullptr){}
      Node(T x):key(x),next(nullptr){}
   };
   Node* front;       //first element
public:
   List():front(nullptr){}   //default constructor
   List(T x):front(new Node(x)){}
   //initializer list constructor
   List(initializer_list<T>);
   ~List();           //destructor
   List(const List&); //copy constructor
   List(List&&);      //move constructor
   //copy assignment
   List& operator =(const List&);
   List& operator =(List&&); //move assignment
   bool push_front(T);       //add to the top
   bool push_back(T);        //add to the bottom
   T& operator [](int);      //index operator
   size_t size();     //the length of the list
   bool find(T);      //search for element;
   Node* find_ref(T); //reference to element
   bool empty(){ return front == nullptr; }
   bool remove(T);
};
```

In addition, students develop on their own methods to insert a node in any location; to search and insert an element in a way to keep the list sorted; to exchange places of elements; to insert an element before and after a node; to merge two lists and more.

Since we count on the reliability, in the course we try to follow the methodology for verification of object-oriented programs as proposed in [3]. Correct implementation of all methods requires multiple checks; catching any exceptions; tracking the number of references to a node. Our current practice shows that students encounter the greatest difficulties in removing items from the list and the most common mistake is to forget a `delete` operator in any branch of the algorithm. So in fact an element is excluded from the list, but the occupied memory is not released – a typical example of a memory leak. Other typical logic errors are skipping a special case such as an attempt to delete an item from an empty list or when the element to be deleted is the first in the list.

In order to simplify the technical part and to focus on algorithms, implementing the operations on lists from 2013-2014, we went to implementation with smart pointers. Our initial expectation was that it was possible to avoid all methods of copy and move semantics, destructors for nodes and list, release of memory when deleting nodes and exception handling related to the construction of a list and its nodes. We relied on simplified syntax in the implementation of operations.

We started with the realization of the template class with the following interface:

```
template <typename T>
class List {
    class  Node {
    public:
        T key;
        shared_ptr<Node> next;
        Node():key(), next(){}
        Node(T x):key(x), next(){}
    };
    shared_ptr<Node> top;
    shared_ptr<Node> bottom;
public:
    List():top(), bottom(make_shared<Node>()){}
    List(T x):top(make_shared<Node>(x)),
              bottom(make_shared<Node>()){
       top->next = bottom;}
    List(initializer_list<T>);
    bool push_front(T);
    bool push_back(T);
    operator bool(){return top!=nullptr;}
    shared_ptr<Node> find(T)
    bool remove(const T&);
    T& operator[](size_t);
};
```

Unlike the interface of `std::forward_list`, we added a feature inserting elements at the end (the method `push_back`) and aiming a more effective implementation of this, we used a fictitious node `bottom` as a sentinel.

We will show the advantage of using shared pointers through the method `remove` to delete element with a key `x`:

```
template <typename T>
bool List<T>::remove(const T& x) {
   if(!top) return false;
   if(top->key == x) {
          top = top->next;
          return true;
   }
   for(auto p=top; p->next; p=p->next)
          if(p->next->key == x) {
                 if(p->next == bottom)
                 bottom = p;
                 p->next = p->next->next;
          }
   return true;
}
```

It is seen that the code with shared pointers differs from that with build-in pointers only by avoiding `delete` several times to release occupied by the deleted node memory. The code of the other methods is sufficiently clear and concise, for example adding a new element to the beginning of the list looks like this:

```
template <typename T>
bool List<T>::push_front(T x) {
   auto p = make_shared<Node>(x);
   if(!p) return false;
   p->next = top ? top : bottom;
   top = p;
   return true;
}
```

With automatic type deduction and factory function `make_shared` (row 2) we even avoid explicit type declaration for smart pointer `p` and do not use `new`, instead:

```
shared_ptr<Node> p { new Node{ x } };
```

For educational purposes all operations with a single list ran normally, but when we tested a larger list (100000 strings), we got a "stack overflow" error during the automatic destruction of the list at the end of the program. Because of the recursive links a situation occurs where one node keeps "alive" the whole structure. This on one hand requires a large stack, and on the other – can lead to significant delays in the demolition of the structure. So we decided to add a destructor, instead of increasing the stack size from the settings of the linker:

```
template<typename T>
List<T>::~List() {
   while (top != bottom)
        top = top->next;
}
```

Here, again, we don't use `delete` to release the memory occupied by each node, but instead just sequentially shift the first element until we reach the end of the list. This causes automatic execution of a destructor for each node, managed by shared pointer, as there will be no more references to it.

Further, when working with two or more lists, we encountered problems with copy assignment and copy construction. Both operations performed shallow copying and we had to add a copy constructor and copy assignment operator to evoke correct actions for deep copying. Their code proved to be with complexity equivalent to the version with naked pointers, so in this case we could not save the students the technical details.

The situation with move semantics proved to be analogous – the lack of user-defined move constructor and move-assignment operator results in that after the transfer of ownership the pointer members of the object (list) on the right are not reset to its initial state, so we implemented these methods as well, but as seen from the code below, the implementation is quite trivial and does not burden the students:

```
template<typename T>
List<T>::List(List<T>&&other):
   top(move(other.top)),
   bottom(move(other.bottom)) {
          other.top = nullptr;
          other.bottom = nullptr;
   }
```

The reason that compiler-generated move semantics methods don't work is that the complex types, such as our list, often define one or more of the special member functions themselves, and this can prevent other special member functions from being automatically generated. This problem we solved in another way, without implementation of the corresponding methods, but passed to compiler that supports explicitly defaulted and deleted functions – Microsoft Visual C++ Compiler Nov 2013 CTP (CTP_Nov2013).

Then the declarations

```
List(List&&) = default;
List& operator =(List&&) = default;
```

provided smooth operation of the automatically generated move constructor and move-assignment operator. Unfortunately we found that this approach does not work with copy semantics.

Similar difficulties were encountered with the implementation of Doubly Linked List. Here is a part of its interface:

```
template <typename T> class List {
    class Node {
    public:
           T key;
           shared_ptr<Node> next;
           weak_ptr<Node> prev;
           Node():key(),next(), prev(){}
           Node(T x):key(x), next(), prev(){}
    };
    shared_ptr<Node> front;
    shared_ptr<Node> back;
public:
    List():front(), back(){}
    List(initializer_list<T>);
    bool push_front(T);
    bool push_back(T);
//...
};
```

As here are bidirectional links, in order not to duplicate them and make the structure "indestructible", for those in the opposite direction we use a weak pointer. And for this list we can state that implementation of operations has the same or less complexity than the version with built-in pointers.

We will comment on another issue, connected not so much with the lists as with the syntax rules in C++11 and implementation of initializer list constructor. If you try to initialize a list with another using the syntax for uniform initialization:

```
List <int> L2 {L1};
```

If we have templatized initializer list constructor, the compiler will consider this as a call to this constructor with an argument initializating list of one element of type `List<int>`, not as a call to the copy constructor. That would cause unexpected behavior. One option for dealing with the problem is definition of specialization for initializer list constructor for lists:

```
List(initializer_list<List<T>>);
```

The other option is simply to use function syntax:

```
List<int> L2(L1);
```

In conclusion we can assert that although our initial idea to avoid implementation of all special member functions was not completely accomplished, these methods, as well as all operations with lists can be implemented more concisely and clearly than their respective analogues in the build-in pointers implementation. Furthermore, by using smart pointers we implemented a complete "no naked new " policy, respecting the recommendation of [5], p. 64 that avoiding naked `new` and naked `delete` makes code far less error-prone and far easier to

keep free of resource leaks. From this perspective, we consider reasonable study of smart pointers in the course of DSA.

IV. Performance Evaluation

In order to evaluate the efficiency of smart pointers implementation we carried out an experiment in which we compare the times for typical operations with lists, implemented with and without smart pointers.

Three implementations of Singly Linked Lists with library `std::forward_list` and our realization of Doubly Linked List with smart pointers with library equivalent `std::list` we compared (Table 1). The same data is used in the experiment: 100'000 randomly generated unique strings of length of 20 stored in a text file. They are used to construct lists by adding elements to the beginning for the one-directional linked versions and at the end of bi-directional linked lists.

TABLE I. Test Results

Operations	**List Implemantations**					
	Singly Linked Lists				***Doubly Linked Lists***	
	C-style	Row Pointers	Smart Pointers	std::forward_list	Smart Pointers	std::list
Add node	78	109	109	78	125	63
Traverse	14 078	14 703	30 578	19 829	31 829	14 546
Delete node	21 594	21 625	143 703	107 515	153 812	78 172

Note: Time in milliseconds

The first operation "Add element" reads all strings from the file and stores them in the relevant list. For each list the text file is opened and read again.

Traversing accomplishes 10000 searches for an element not contained in the list: that is complete pass over all the nodes.

The test of deletion is deliberately made so as to require multiple traverse – check if each element meets the set criterion (comparison of strings) and if so, the key of this element is passed as argument to the deleting function. This function each time searches the element from the beginning of the list and deletes only the first hit. 59 996 elements of all 100 000 are deleted.

The results show a negligibly small difference in performance between the implementation without classes (C-style), and implementation using classes and raw pointers. Only the "add element" operation is 28% slower. Time difference between single linked lists and bi-directional linked lists implemented with smart pointers is inessential. This was expected because the test algorithms traverse lists only in one direction. The advantage of bi-directional linked list is only visible in comparison with library implementations. The library template class `forward_list` is inferior in efficiency to our raw pointer implementation for traverse operation by 26%, and removing elements is nearly 5 times slower. Implementation of smart pointers has significantly weaker results – traverse is 2 times slower, and removing elements – 6 times compared to raw pointers. Adding elements shows no difference in performances. Our version of bidirectional linked list with smart pointers proved to be twice slower than library version `std::list` for all operations.

V. Conclusion

Our initial hypothesis regarding the implementation of lists with smart pointers was proven partially. We could not do entirely without implementation of methods of copy and move semantics, but their code turned out to be short, clear and easily understandable for students. Moreover, move semantics in our case can be provided by defaulted move constructors and assignment operators. We consider the second part of the hypothesis, namely the shorter and clearer implementation of the basic operations with data structures for fully achieved. In addition, smart pointer versions do not require user-defined exception handling.

Since we do not have enough empirical data, we cannot prove the advantage of this way of teaching DSA yet, but even without conducting a strictly formal pedagogical experiment, we can confirm that the results of students tests, homework and exams are comparable to those demonstrated by their colleagues trained in previous years under the old program.

The implementation of ADS with smart pointers is more clear and concise, but requires spending time to study in additionally templates and essential elements of the STL, though not in detail. This could be facilitated by reorganizing CS1 course Programming Fundamentals, where to underlie learning C++11/14 and STL. Note that for our implementations it is not needed even to know the full interface for work with smart pointers. In most situations the interface of build-in pointers is sufficient plus function `make_shared` and possibly member function `reset`. In our work with the students during the school year we met difficulties in debugging of programs related to discovery of logical errors in memory management, most often connected with its release. We found that it is appropriate to add an intermediate output (operator `cout`) in the destructors as of DSA, as of the elements held in them (if they are of user-defined types). In this way it is easy to detect situations where objects remain undestroyed.

Regarding the applicability of smart pointers in the actual programming will mention the opinion of Stroustrup, that they "are still conceptually pointers and therefore only my second choice for resource management – after containers and other types that manage their resources at a higher conceptual level" [5], p. 114. The results of our comparative tests also show that library containers are sufficiently effective and can join the opinion of Stroustrup. Furthermore, anyway, to learn smart pointers it is necessary to get into STL. On one hand it is better to teach students how to use its efficient and reliable containers. On the other hand though, we train professionals and they must be able to independently implement such containers – to develop creative thinking. It is therefore not a bad idea to do so with smart pointers as well – one more opportunity provided by the STL.

References

[1] Josuttis, N. M. (2012). *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional; 2nd edition (April 9, 2012).

[2] Boehm, H. & Spertus, M. (2009). *Garbage Collection in the Next C++ Standard*. Proceedings of the 2009 international symposium on Memory

management, pp. 30-38. ACM New York. doi>10.1145/1542431.1542437

[3] Todorova, M., Kanev, K. (2012). *Educational framework for verification of object-oriented programs*, in Proceedings of the 2012 Joint International Conference on Human-Centered Computer Environments, ACM, New York, pp. 23-27

[4] ISO/IEC. (2011). *International Standard ISO/IEC 14882:2011(E) Information technology – Programming languages – C++* (3rd ed.)

[5] Stroustrup, Bj. (2013). *The C++ Programming Language, 4th Edition*. Addison-Wesley Professional; 4th edition (May 19, 2013)

[6] Dimov, P., Dawes, B. & Colvin, G. (2003). *A Proposal to Add General Purpose Smart Pointers to the Library Technical Report*. C++ Standards Committee Papers. Document number: N1450=03-0033 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1450.html

[7] Sutter, H. (2013). *Sutter's Mill. GotW #89 Solution: Smart Pointers*. http://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers/