Implementation of Binary Search Trees 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*—Study of binary trees has prominent place in the training course of DSA (Data Structures and Algorithms). Their implementation in C++ however is traditionally difficult for students. To a large extent these difficulties are due not so much to the complexity of algorithms as to language complexity in terms of memory management by raw pointers – the programmer must consider too many details to ensure a reliable, efficient and secure implementation. Evolution of C++ regarded to automated resource management, as well as experience in implementation of linear lists by means of C++ 11/14 lead to an attempt to implement binary search trees (BST) via smart pointers as well. In the present paper, the authors share experience in this direction. Some conclusions about pedagogical aspects and effectiveness of the new classes, compared to traditional library containers and implementation with built-in pointers, are made.**

Keywords—abstract data structures; binary search trees; C++; smart pointers; teaching and learning

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 a 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 is the situation when an object must participate in multiple collections – each of them must provide a pointer to this object. To make everything correct it is necessary to be sure that:

- when destroying one of the pointers, take care that there are no dangling pointers or multiple deletions of the pointed object;
- when 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 valid object.

It is a must to have in mind such details to accomplish dynamic implementation of ADS (Abstract Data Structures) and often time for this exceeds time remaining to comment the structures and operations on them. Moreover, there are rare cases when these is a working implementation of a structure with carefully designed interface and methods written according to the best methodologies, but gaps can be identified in memory management only when a non-trivial situation occurs, 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, it is decided to change the content of CS2 course in DSA, include the study of smart pointers for resource management and with their help to simplify implementations of ADS to avoid explicit memory management which is widely recognized as error-prone [2].

In the work, the emphasis is on the implementation of linear structures (linked lists) and binary trees. This paper discusses only part of this work dedicated to binary search trees (BST).

The initial hypothesis is that a correct and effective implementation of BST is possible, which could relieve the work in two directions:

- operations with whole structures (trees): not having to implement copy and move semantics methods;
- shorter explanation and easier understanding of implementation of operations with elements of BST – include (insert element), search, delete.

The remaining content of the paper is as follows: Section II is a brief overview of language features for managing dynamic memory and its development. In paragraph III an implementation of Binary Search Trees (BST) is presented and compared to those based on build-in pointers. Section IV discusses effectiveness of the implemented structures and algorithms compared to the similar realization of the library container `std::set`. In section V some conclusions are made and recommendations are given for smart pointers usage in the DSA course.

II. Development of Language Features 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>.

Memory blocks allocated by 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 with using these functions 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].

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 the 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 catch all exceptions. To handle 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. The situation is similar when the function execution is terminated by premature return statement based on some condition (early return);
- **premature deletion**: An object is deleted that has some other pointer to and later that other pointer is used.
- **double deletion**: There is a possibility to re-delete the object.

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 to use "smart pointers". Their "intelligence" is expressed in the fact 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.

It is possible to 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 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 – 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). 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 are carried out. Such is the situation with passing of auto_ptr as a parameter of the function. After function completes the memory allocated in the initialization of auto_ptr variable and then passed as argument to the function will be released (at destruction of the formal parameter) and will not be given back to this variable (the actual parameter). This will result in a dangling pointer. The auto_ptr provides semantics of strict ownership. auto_ptr owns the object that holds a pointer to. Copying 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, program behavior 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 for a single object, not for 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.

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 it leads to wrong program behavior.

In the standard [3] instead of auto_ptr several different types of smart pointers are introduced (also called Resource Management Pointers) [4]. They model different aspects of resource management. The idea is not new – it formally originates from [5] and is 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 pointer semantics is you really needed, unique_ptr is a very lightweight mechanism, with no additional costs compared to the correct use of built-in pointer [4], p. 113. The class unique_ptr is designed for pointers that implement the idea of exclusive (strict) ownership, what is 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. Use of `unique_ptr` includes passing free-store allocated objects in and out of functions (rely on move semantics to make return simple and efficient).

Copying or assignment between unique pointers is impossible if ordinary copy semantics is used. However, move semantics can be used. In that case, the constructor or assignment operator transfers ownership to another unique pointer.

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);
- deallocating dynamically allocated memory for a function;
- storing pointers in the 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 this function then does not pass ownership again, the object will be destroyed at the completion of the function.

Using a unique pointer, as a member of a class may also be useful to avoid 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 – it is known that destructors are called only if any construction has been completed. So, if an exception occurs within a constructor, destructors will be executed for objects that have been already fully constructed. As a result there can be 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 it was already commented on the problems associated with their use. Often it is needed 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 shared_ptr` class 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 a 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 ->. A full range of copy and move constructions and assignments is available. 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()`).

By using shared pointers the problems with dangling pointers can be avoided. This problem arises while pointers are stored in containers.

A problem with reference-counted smart pointers is that if there is a ring of objects that have smart pointers to each other, they keep each other "alive" – they will not be 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_ptrs`, which point to the managed object but do not keep it in existence. Like raw pointers, weak pointers do not keep the pointed-to object "alive". The cycle problem is solved. However, unlike raw pointers, 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 the programmer hesitates which version of a smart pointer to use – `unique_ptr` or `shared_ptr`. The advice is to prefer `unique_ptr` by default, because later move-convert to `shared_ptr` can be done if needed. There are three main reasons for this [6]:

- 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 the options open.

In this particular case, however, it is necessary to start from the very beginning with `shared_ptr`, because being recursive by definition, binary trees that have to be implemented with smart pointers, and this cannot do without shared ownership.

III. Implementation of Binary Search Trees

Most attention in the course is given to binary search trees, so here the focus is only on the implementation. The traditional implementation interface with build-in pointers looks like this:

```
template <typename T>
class BTree {
   struct Node {
        T key;
        Node* left;
        Node* right;
        Node();
        Node(T);
   };
typedef Node* pNode;  //pNode& instead of Node*&
   pNode root;
   //........ some helper functions here ........
public:
        BTree() : root(nullptr){}
        ~BTree();
        BTree(const BTree&);
        BTree(BTree&&);
        BTree& operator =(const BTree&);
        BTree& operator =(BTree&&);
        bool insert(T);
        bool remove(T);
        void inorder(void(*)(pNode&));
        void preorder(void(*)(pNode&));
        void postorder(void(*)(pNode&));
        void breath_first(void(*)(pNode&));
        size_t height();
        Node* find(T);
};
```

Beside the special member functions methods are added to insert, search and remove elements, and various deep-first (`inorder`, `preorder`, `postorder`) and breath-first traversals. A number of additional functions are included. Their implementation is a question of interest, for example, calculating the height of the tree and, if there is enough time, balancing. For implementation of these operations, recursive algorithms are preferred because they are shorter and more intuitive. Most difficulties are met with the deletion, which is normal – the algorithm is most complex.

Since the aim is to count on the reliability, in the course it is chosen to follow the methodology for verification of object-oriented programs as proposed in [7].

In order to simplify the technical part and to focus on algorithms, implementing the operations on trees from 2013-2014, it is decided to choose implementation with smart pointers. The initial expectation is that it is possible to avoid all methods of copy and move semantics, destructors for nodes and whole trees.

The interface of smart pointers implementations with which the work is started is the following:

```
template <typename T>
class Tree {
  struct Node {
         T key;
         shared_ptr<Node> left;
         shared_ptr<Node> right;
         Node():key(), left(), right(){}
         Node(T x):key(x),left(), right(){}
  };
  shared_ptr<Node> root;
//...
public:
  Tree():root(){}
  ~Tree();
  Tree(Tree&&) = default;
  Tree& operator =(Tree&&) = default;
  Tree(const Tree&);
  Tree& operator =(const Tree&);
  bool push(T);
  bool remove(T);
  void inorder();
  shared_ptr<Node> find(T x) {
         return find(x, root);
  }
  void breath_first();
  size_t height(){
         return height(root);
  }
};
```

Because of recursive algorithms that are used for each operation two functions had to be written – one private, with additional parameter the node from which to start. So public method is very short and just calls the corresponding private method that implements the algorithm. For example the public method for deleting:

```
template <typename T>
bool Tree<T>::remove(T x){
         return remove(x, root);
}
```

calls the private method `remove(T, shared_ptr<Node>&)` where the second parameter is the root of the tree:

```
template <typename T>
bool Tree<T>::remove(T x, shared_ptr<Node>& p) {
   if(p && x < p->key)
        return remove(x, p->left);
   else if(p && x > p->key)
        return remove(x, p->right);
        else if(p && p->key == x) {
           if(!p->left)
                p = p->right;
           else if(!p->right) p = p->left;
                else {
        shared_ptr<Node> q = p->left;
        while(q->right) q=q->right;
        p->key = q->key;
        remove(q->key, p->left);
        }
        return true; }
        return false;}
```

We note that the code for this method is 37% shorter than the code for the corresponding raw pointers implementation (due mainly to the fact that there is no need to call `delete`). In addition readability of code is improved. For inserting a node there is no difference between the amounts of code – both methods have 16 rows.

For educational purposes, all operations with a single tree run normally, but when a larger tree is tested, a "stack overflow" error appears during automatic tree destruction at the end of the program. With a standard size of 1 MB stack error occurs even for destruction of a tree of 29,000 integers. Because of recursive links, a situation arises 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 demolition of the structure. So the choice is to add a destructor, instead of increasing stack size from the settings of the linker. The decision is not to work for efficiency and chose the easiest option – using the method for deletion. As such, the destructor looks like this:

```
~Tree(){
        while (root) remove(root->key);
}
```

As for the implementation of special member functions, defaulting of move constructor and move assignment operator works and it is not needed to implement move semantics, but copy semantics requires to write appropriate methods, because it is needed to copy the entire tree structure, so as to obtain a true copy of the tree, not just tree, which contains the same elements.

Comparing the overall implementation of trees with raw pointers, the conclusion is that smart pointers give short and easy to understand code without apparent loss of efficiency (Table 1).

IV. Performance Evaluation

In order to evaluate the efficiency of smart pointers implementation an experiment is carried out in which times for typical operations with binary trees, implemented with and without smart pointers, are compared.

Three conversions are compared: traditional row pointer implementation, new smart pointer and library implementation `std::set` (Table 1). Note that `std::set` is typically implemented in libraries as a red-black tree. This may adversely affect time for generating the tree (for coloring and balance), but improves search speed.

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 trees. The first operation "Add element" reads all strings from the file and stores them in the relevant tree. For each tree, the text file is opened and read again. For unbalanced versions, a tree with height of 38 is obtained.

In testing for search and remove elements another file is used, which records 10,000 strings that are found in the tree. The algorithm makes search and remove operations for exactly these elements.

TABLE I. Test Results for Binary Trees

Operations	*Binary Search Tree Implementations*		
	Row Pointers	Smart Pointers	`std::set`
Add element	438	453	156
Search	31	32	15
Remove	47	46	32

Note: time in milliseconds

The results show that there is practically no difference in performance between implementation of operations with build-in and smart pointers, which is a good argument to continue to study smart pointers in the course DSA. Some surprise is the time for `std::set` in operations creating structure (adding operation), which is three times better. Apparently, extra time for coloring and balancing the tree is offset by the lower height of the red-black tree – `std::set` for these input data theoretically the tree can get a height of 12, and as mentioned before, the tree in our implementations has height 38. For the same reasons, search time in our implementations is 2 times worse, and time for removing elements – 1.5 times worse library implementation.

V. Conclusion

The initial hypothesis regarding the implementation of BSTs with smart pointers is proven partially. It is not possible to do the work entirely without implementation of methods of copy and move semantics, but their code turns out to be short, clear and easily understandable for students. Moreover, move semantics can be provided by defaulted move constructors and assignment operators. It is considered that the second part of the hypothesis, namely the shorter and clearer implementation of basic operations with data structures is fully achieved. In addition, smart pointer versions do not require user-defined exception handling.

Since there is not enough empirical data, the advantage of this way of teaching DSA cannot be proved yet, but even without holding a strictly formal pedagogical experiment, it can be stated that results of students tests, homework and exams are comparable to those demonstrated by their colleagues trained in previous years under the old program.

Implementation of ADS with smart pointers is more clear and concise, but requires spending time to study in addition 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 the presented 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`. While working with students during the school year some difficulties are met in debugging of programs related to discovery of logical errors in memory management, most often connected with its release.

It is appropriate to add an intermediate output (operator `cout`) in 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 actual programming the opinion of Stroustrup should be mentioned, 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" [4], p. 114. The results of comparative tests also show that library containers are sufficiently effective. In order 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, as future professionals 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] ISO/IEC. (2011). *International Standard ISO/IEC 14882:2011(E) Information technology – Programming languages – C++* (3rd ed.)

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

[5] 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

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

[7] 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