Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2021-09-08

Linked list

Aim

In this laboration you will learn to handle dynamic memory and pointer structures. Specifically you will see how constructors, destructors and operators may help you in keeping track of pointers and allocated memory.

Reading instructions

- Recursion
- Pointers
- Inner classes/structs
- The five important
  - Copy constructor (deep copy to a new object)
  - Assignment operator (deep copy to an existing object)
  - Move constructor (quick move from a dying object to a new object)
  - Move assignment (quick move from a dying object to an existing object)
  - Destructor (deep deallocation)
- *Optional*: Random generation (`random_device`, `default_random_engine`, `uniform_int_distribution`, etc.)

Assignment: Singly linked sorted list

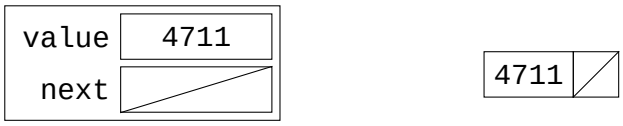
In this assignment you will create a straight singly linked list. Just like the previous lab, the aim of this lab is not to make a program, but instead a module that can be used by other programmers.

The list will be represented by a class. You need functions to handle the list in a correct way in all situations, this includes copying and assigning between lists, as well as functions to insert, remove, iterate items in the list. The list should of course be free of memory leaks no matter what.

Internally, as an inner class (or struct), you will have a class object (or struct object) representing a link in the list. A link will keep track of the value stored in this position, and the next link in the list. The chain of links will build the list.

As each link will keep track of the next link, the list class need to only keep track of the first link in the list. The list class should have at least functions for insertion and removal of items in the list, printing of the entire list and accessing an element at a certain index. You may add functionality as you see fit as long as they are sensible.

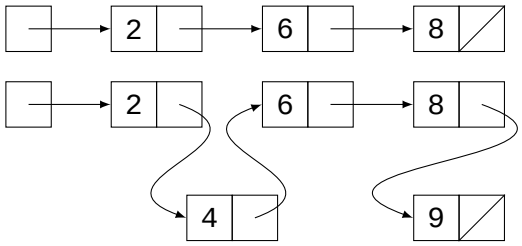
Figure 1 demonstrates the memory layout of a link with value `4711` and how we usually draw it in a simplified form. Drawing the pointer structure and performing operations step by step on the drawing is a good way of debugging the list.



Figur 1: A link in memory, in detail and in simplified form

An important aspect of the list as a program module and as an abstraction is the interface. The programmer that use the list should not (and need not) know how the list is built internally. The link class and any functions pertaining to it should thus be stashed away and be inaccessible to the programmer.

You should start by planning your insert and remove operations and draw how they will work on paper. It is suitable to start with an empty list, insert 5, 3, 9, 7, remove them again (in that order), and in each step think of which pointer variables that need modifications. demonstrates an example of how we draw a list after insertion of 6, 2 and 8 and another example when 9 and 4 have been inserted.



Figur 2: List after insertion of 6, 2, 7 and then 9, 4

All functions you write should be tested. The test cases should cover normal as well as exceptional cases. Insertion first may for example need special treatment in the code, and should thus be tested more careful. More such cases are likely to exist, but may vary with solution.

All operations on your list must of course work on all special cases of a list, including empty lists. Copying and assigning from list to list should work, and must create deep copies (changes in the copy will not affect the original). Your test program will help you ensure that this works properly.

You list should also have support for the move-operations (move constructor and move assignment). It is sufficient to simply move pointers around for these functions.

Note: Your header file may only contain declarations; no implementations.

Requirements

Below are all the minimum requirements for the lab. The list class must:

- Always be sorted,
- Have an insert function that adds a value to the list (in sorted order),
- Have a remove function that removes a value (either by index or by value),
- Have a print function that prints all the values in the list (the formatting must be readable),
- Have an at function that return the value stored at a specified index,
- Support all special member functions correctly (destructor, copy constructor, move constructor, copy assignment operator and move assignment operator),
- Have no memory leaks, under any circumstances.

At least one of these functions must be *recursive* and at least one of them must be *iterative*.

The link/node class must be an *inner class (or struct)* of the list class.