Take-Home Exam 2

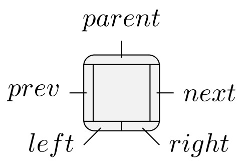
(Deadline: 23th of June 2023, Friday, 23:59)

Introduction

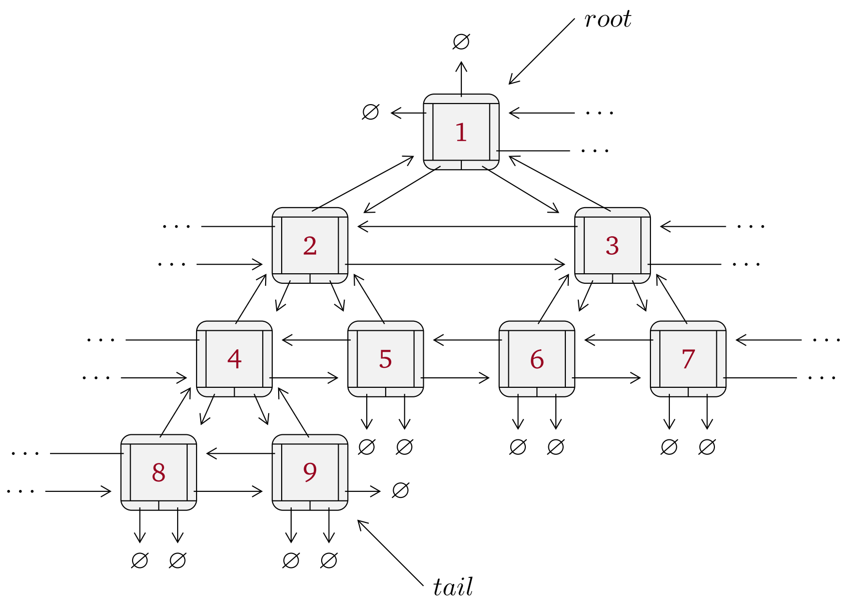
In this THE, you will implement a data structure that is essentially a mixture of a nearly complete binary tree and a linked list. We named this data structure the *l-tree*. The structure has the following properties:

- Its shape is a nearly complete (pseudo-complete) binary tree with the *traditional heap shape*. (We are just talking about the shape. There is no heap-order in this tree.)
- It is implemented with *binary tree nodes containing child and parent links*, rather than with an array.
- It also maintains a *doubly linked list* on its nodes, representing the traditional 1 to n labelling of the nodes.

Internally, an l-tree contains nodes which host five pointers: (a) $parent/left/right$ pointers for the binary tree structure, and (b) $prev/next$ pointers for the linked list structure. Visually, we display these nodes as follows:



The access to the structure is obtained through two pointers: a $root$ and a $tail$. Below is an example l-tree of 9 elements where the nodes are labelled with their traditional heap-shape indices:



Handle Logic

When an item is pushed into the l-tree, the pointer to its newly created node is returned. This pointer acts as a *handle* for the item and should remain associated with the item unless the item is popped. In other words, you are NOT allowed to change the value stored in the nodes as a result of any operation.

Task

You are given a partially implemented class template named `LTree` and you are expected to complete its missing functionality. The partial implementation is below:

```
template<typename T>
class LTree
{
    friend class diag; // For testing.
    public:
        class Node
        {
            friend class LTree;
            friend class diag; // For testing.

            private:
                const T mValue;
                Node *mParent, *mLeft, *mRight, *mPrev, *mNext;

                // Constructs a new node with all links equal to null.
                Node(const T &value) : mValue(value), mParent(nullptr),
                                       mLeft(nullptr), mRight(nullptr),
                                       mPrev(nullptr), mNext(nullptr) {}

            public:
                Node *parent() { return mParent; }
                Node *prev() { return mPrev; }
                Node *next() { return mNext; }
                Node *left() { return mLeft; }
                Node *right() { return mRight; }
                const T & get() { return mValue; }
        };

    private:
        Node *mRoot, *mTail;

        static void destroy(Node *node) { // For use in destructor.
            if (node != nullptr) {
                destroy(node->left());
                destroy(node->right());
                delete node;
            }
        }

    public:
        // Constructs an empty l-tree.
        LTree() : mRoot(nullptr), mTail(nullptr) {}

        // The l-trees cannot be copied.
        LTree(const LTree &) = delete;
        LTree & operator=(const LTree &) = delete;

        Node *root() const { return mRoot; }
        Node *tail() const { return mTail; }
        ~LTree() { destroy(mRoot); }

        // *** IMPLEMENT THESE FUNCTIONS ***.
        Node *pushBack(const T & value);
        void popBack();
        void exchange(Node *a, Node *b);
        void split(LTree **outLeft, LTree **outRight);
};
```

The missing functionality is as follows:

- `Node *pushBack(const T & value);`
  Creates a new node containing `value` as its value in the l-tree. This new node should be placed at the required position to keep the classical pseudo-complete heap-shape of the structure. The pointer to this new node should be returned.

- `void popBack();`
  Pops the tail of the l-tree. The popped node should be deleted. This method will always be called on a non-empty l-tree.

- `void exchange(Node *a, Node *b);`
  Swaps two *distinct* nodes in the l-tree. The nodes are guaranteed to belong to the l-tree instance. Note that the nodes themselves are swapped in the tree/list structure, not their values.

- `void split(LTree **outLeft, LTree **outRight);`
  Splits the l-tree into three:

  (a) The left subtree, which is supposed to be returned as a new l-tree. The pointer of the new l-tree is supposed to be returned via the output parameter `outLeft`.

  (b) The right subtree. This is similarly constructed as a new l-tree and returned in `outRight`.

  (c) The root of the tree, which will remain in the original l-tree instance.

  The method will always be called on a non-empty l-tree. Notice that, after the split, the original instance and the two resulting instances should be valid l-trees in their own rights, with all their links consistent. Since this is a split, you are not supposed construct any new nodes.

Efficiency

All your methods should be reasonably well-implemented and complete in $O(\log n)$ time where n is the number of nodes in the l-tree. A portion of the points will come from the efficiency aspect of your code. You are also expected to avoid memory leaks by properly cleaning unneeded nodes via `delete`. However, we will not specifically test for memory leaks in this THE.

Arrays (and alike) Forbidden

You are *forbidden* from using:

- Arrays.
- Any data structure from any library that can hold multiple items such as `vector`, `set`, `map`, etc.

You can use basic functions from the standard library such as `std::swap`.

Submission

Submission is through ODTÜClass. We will provide you a CMake project archive that contains the partial implementation of the l-tree in a file named *LTree.tpp* and a stub file named *the2.tpp* for your implementation. We expect a single updated *the2.tpp* from you as part of the submission.

Your code will be compiled on `g++` with the options: `-std=c++2a -w`. However, we suggest using the following options when compiling your code yourself: `-std=c++2a -Wall -Wextra -Wpedantic`.

Diagnostics Utilities

In the project archive, there is a header named *diag.tpp* which contains some useful functions to test your code:

- `LTree<unsigned> * diag::range(unsigned n);`
  Returns a new l-tree of unsigneds. The returned tree contains n nodes whose values are set from 1 to n representing their traditiona labels.

- `template<typename T> void diag::check(const LTree<T> *t);`
  Checks whether the given l-tree has consistent links. Being consistent does not mean that your l-tree is correct; however, it means that it is a valid l-tree in its own right. If the check fails, an exception is thrown with an appropriate error message.

- `template<typename T> void diag::print(const LTree<T> *t);`
  Prints the given l-tree to the standard output by visiting its node in their linked list order. It also performs a consistency check prior to printing.

Grading

You will submit your file through a VPL item in ODTÜClass. The item will have an "Evaluate" feature which you can use to auto-grade your submission. (We will make it ready in a few days.) To get full marks, you need to implement all functions and your implementations should adhere to the description above. You may get partial points if you implement a subset of the functions. Also note that your code is supposed to work within the reasonable time/memory limits in place.

The grade from the auto-grader is not final. We may do additional black-box and white-box evaluations and adjust your grade. Solutions that do not reasonably attempt to solve the given task as described may lose points.

No late submissions are allowed. *Please, start now!* The amount of code you need to write is not large, however, it involves a lot of thinking and edge cases.

As always, you are expected to do this THE in academic integrity!

Example Code

```
#include "LTree.tpp"
#include "the2.tpp"
#include "diag.tpp"

int main()
{
    auto t = diag::range(9);
    diag::print(t); // 1 2 3 4 5 6 7 8 9

    auto node99 = t -> pushBack(99);
    diag::print(t); // 1 2 3 4 5 6 7 8 9 99

    t -> pushBack(100);
    t -> pushBack(101);
    diag::print(t); // 1 2 3 4 5 6 7 8 9 99 100 101

    t -> popBack();
    t -> popBack();
    diag::print(t); // 1 2 3 4 5 6 7 8 9 99

    auto node5 = node99 -> parent();
    auto node6 = node5 -> next();
    auto node2 = node5 -> parent();
    auto node4 = node2 -> left();

    t -> exchange(node4, node99);
    diag::print(t); // 1 2 3 99 5 6 7 8 9 4

    t -> exchange(node99, node2);
    diag::print(t); // 1 99 3 2 5 6 7 8 9 4

    t -> exchange(node99, node2);
    diag::print(t); // 1 2 3 99 5 6 7 8 9 4

    t -> exchange(node5, node6);
    diag::print(t); // 1 2 3 99 6 5 7 8 9 4

    LTree<unsigned> *l, *r;

    t -> split(&l, &r);

    diag::print(t); // 1
    diag::print(l); // 2 99 6 8 9 4
    diag::print(r); // 3 5 7
    delete t;
    delete l;
    delete r;

    return 0;
}
```

Hints

- One fundamental difficulty in this task is to make sure that all links are consistently updated. Notice that there are two kinds of links: (a) the tree links and (b) the list links. Both kinds are bidirectional; if you get one link right, you should be able to correct its opposing link without much effort.
- You also need to keep an eye on the *root* and *tail* pointers while you are doing modifications.
- We suggest that you write some utility functions to simplify your code. You can freely add global functions in *the2.tpp*; however, due to the way the class is declared, these functions cannot access the private fields of the l-tree or the nodes. If you want to write a function that can access the private fields, you can write it as a lambda expression inside a member function. The following is an example of a lambda expression of that kind:

  ```
  auto swapChildren = [] (const LTree<T>::Node *node) -> LTree<T>::Node *
  {
      std::swap(node->mLeft, node->mRight);
      return node->mLeft;
  };

  myNewLeftChild = swapChildren(myNode);
  ```

Here are some more tips about the individual functions (in supposed difficulty order):

- `void popBack();`
  This is the easiest of the functions and we suggest that you attempt this the first.

- `Node *pushBack(const T & value);`
  The main difficulty here is to locate the parent node under which the new node is going to be placed. You will need to do some navigation through the tree starting from the tail. The direction of the links should give you clue on where you can find the aforementioned parent node. You may need to write code for a few different general cases in this one, in the addition to the edge cases.

- `void split(LTree **outLeft, LTree **outRight);`
  The main modification in this function is to the list structure and it essentially boils down to connecting the right set of nodes with another right set (on both sides of the split). Another difficulty is to figuring out the new tails for both sides. One side keeps the original tail and you have to figure out which one.

- `void exchange(Node *a, Node *b);`
  This may actually be the most difficult piece. There are four main things you need to work through:
  - Swapping of the nodes' links.
  - Updating the *root* and *tail* appropriately.
  - Take care of the situation when the swapped nodes are neighbors (in the linked list or in the tree). This is probably the most confusing part of the storry.
  - Making sure the opposing links are corrected. Combined with the item above, this part may also become confusing.

  Working through the most basic ones may get some partial points.

Good luck!