std::list internals

Basic operations

```
iterator insert(iterator pos, T value);
iterator erase(const_iterator pos);
void push_front(T value);
void push_back(T value);
```

Note: std::list does not implement operator[] (because this operator must work in O*(1), because of standard)

Implementation

First try

```
template <class T>
class list {
    using value_type = T;

    struct Node {
        T value;
        Node* next;
        Node* prev;
    };

    size_t sz;
    Node* head;
    Node* tail;
};
```

Note a few optimizations:

1. We will make a node (FakeNode), that is **connected to the last and the first element of the list**
2. Notice, that FakeNode must hold a **fictional element of type T**. That's why we use inheritance and rewrite struct Node.

```
struct BaseNode {
    BaseNode* next;
    BaseNode* prev;
};

struct Node : BaseNode {
    T value;
};
```

Also **note**, that in a default c-tor We **don't** want to allocate anything and we want begin() == end(). That's why we hold BaseNode instead of BaseNode*.

```
// default c-tor
list() : sz{0}, fakeNode{&fakeNode, &fakeNode} {}
```

Final implementation

```
template <class T>
class list {
    using value_type = T;

    struct BaseNode {
        BaseNode* next;
        BaseNode* prev;
    };

    struct Node : BaseNode {
        T value;
    };

    size_t sz;
    BaseNode FakeNode;

public:
    // default c-tor
    list() : sz{0}, fakeNode{&fakeNode, &fakeNode} {}
};
```

list::sort()

A few **notes**:

1. Under the hood, it is **in-place merge sort** (makes sense, cause it is just nodes)
2. It uses O(log N) for memory (blocks of 2ˆk, 2ˆ(k - 1), . . . , 2; therefore we need O(log N) pointers for blocks), however log N = const ;).

list::splice()

`list::splice(const_iterator pos, iterator first, iterator last);`

Goal: want to move a range of elements into the list

Note:

1. If we split one list into another, it works in **O(last - first)**, because we need to recalculate the size of the list
2. If it is the same list, than it is **O(1)** (you also give a **list&& other** to splice) (Note: if the ranges overlap, it is **UB**).

Problems which are solved with lists

Problem 1

Check if a list is cycled

Solution 1

The rabbit and turtle.

We are Done!

std::map internals

Basic operations

```
pair<it, bool> insert(const pair<Key, Value>& kv);
size_type erase(Key);
Value& at(Key);
Value& operator[](Key);
iterator find(Key)
size_t count(Key);
bool contains(Key);
```

Note:

1. const map can't use *operator[]* (no const version, cause it creates value, if it does not exist)

Implementation

```
template <class Key, class Value, class Compare = std::less<Key>>
class map {
    struct BaseNode {
        BaseNode* parent;
        BaseNode* left;
        BaseNode* right;
    };

    struct Node : BaseNode {
        pair<const Key, Value> kv;
        bool red;
    };
};
```

Note:

1. FakeNode will be the root of the tree, and the tree will be the left subtree.

2. We also need to hold pointers to begin() (*the leftest child*) and end() (points to fakeNode).
3. FakeNode is cycled with itself.
4. We hold pair<**const** Key, Value> in Node, because we don't want to be able to change the value of key via iterator (because then we would need to know about it and rebalance the tree)

Note also a few side-effects (in both cases, there is a missing const in key slot)

1. The object is copied every iteration:

```
for (const <Key, Value>& v : mp) {
    // some code...
}
```

2. Dangling reference (CE since C++26):

```
const pair<int, std::string>& first() {
    // ...
    return *mp.begin();
}
```

Exception safety:

list

In list the exception safety is **strict** and easy to implement;

map

Also **strict** exception safety

Insert: comparator can throw, but it throws when nothing is done, so we have nothing to rebuild. When we do rotations, we are in a safe zone (no throws).

Erase: analogous to insert