TND004: lab 3

- Binary search trees supporting bi-iterators
  - Extend the `BinarySearchTree` class -- sec. 4.3 of course book

Note: the code in the slides has simplifications
Be watchful when implementing

`std::map`

sorted by key

```
map<string, int> table;
…
for (auto &p: table) { //C++14
    cout << p.first << "  " << p.second;
}
for (auto &[key, value]: table) { //C++17
    cout << key << "  " << value;
}
```

Key: word	Value: counter
obligations	*3*
pauses	1
permutations	2
pressures	5
quebec	2
…	

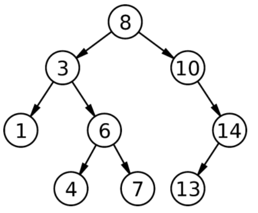
Often libraries implement `std::map` as a binary search tree -- *red-black tree*

```
…
auto it = find(table.begin(), table.end(), "pressures");
while (it != table.begin()) {
    cout << it->word;
    it--;
}
```

Recall TNG033/Lab3

Lab 3

```
BST<int> T; //binary search tree storing ints

T.insert(8);
…

BST<int>::Iterator it{};

//Display the keys in increasingly order  -- inorder
for(it = T.begin(); it != T.end(); it++) {
    cout << *it << " ";
}

for (auto key : T) {
    cout << key << " ";
}
```

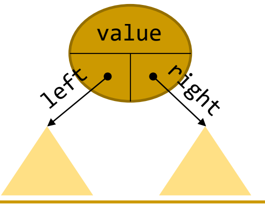


- Create a template class for binary search trees storing items of a generic type `T`
  - Requirement: there is an `operator<=` for type `T`
- Create a class for (bi-directional) iterators which perform an in-order traversal of the tree
- To use the binary search tree class to represent a data structure similar to `std::map`
  - Create a frequency table of words – TNG033: lab3/exerc3

Binary search trees: implementation

```
template <typename Comparable>
class BST
{
    class Node {
    public:
        Node(const Comparable &v, Node *l = nullptr, Node *r = nullptr)
            : value{ v }, left{ l }, right{ r } { }

        Comparable value;
        Node *left, *right;  //left and right sub-trees
    };

  public:
    …
  private:
    Node *root;
};
```

Template class



Template classes: member functions should be implemented in the header file

Exercise 2: to add iterators

```
template<typename Comparable>
class BST
{
    public:
        class Iterator {
        public:
            Iterator();
            Comparable& operator*() const;
            Comparable* operator->() const;
            bool operator==(const Iterator &it) const;
            bool operator!=(const Iterator &it) const;
            Iterator& operator++(); //pre-increment
            Iterator& operator--(); //pre-decrement
            …
        private:
            Node *current;
            Iterator(Node *p = nullptr) : current{ p } { };
        };
        …
        Iterator begin() const;
        Iterator end() const
        …
};
```

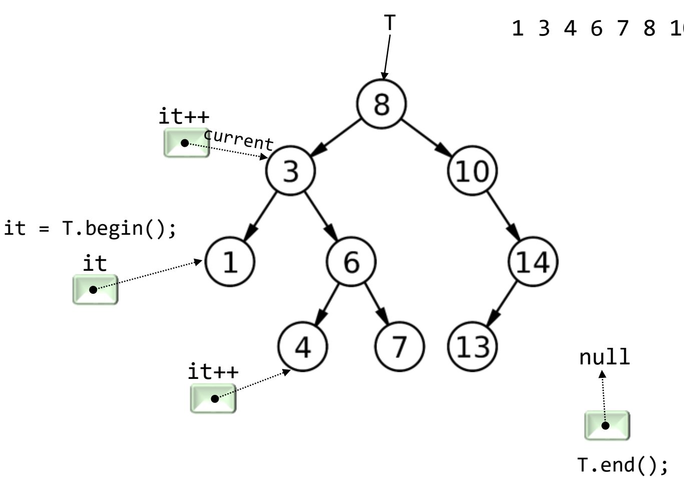
Iterator object holds internaly a pointer to a node


```
BST<int>::Iterator it{ };

for (it = T.begin(); it != T.end(); it++)
    cout << *it << endl;
```

In-order traversal

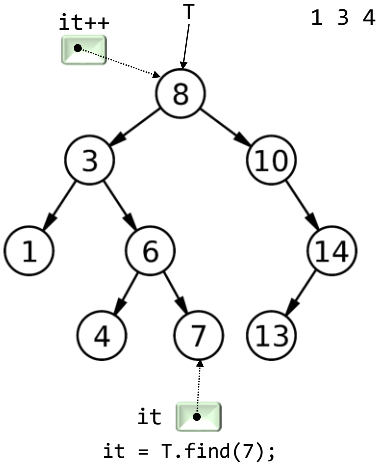
Inorder:

1 3 4 6 7 8 10 13 14



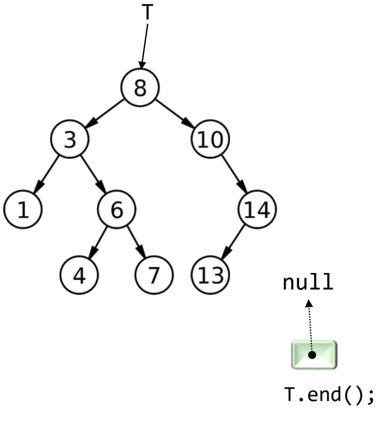
```
auto it = find(7);
it++;
```

Inorder:
1 3 4 6 7 8 10 13 14



T.end()

```
BST::Iterator end() const
{
  return Iterator(nullptr);
}
```

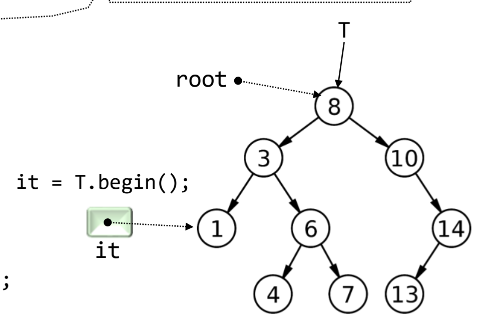


T.begin()

- The smallest value stored in the tree is in the left most node

```
//Return pointer to node storing min
Node* findMin(Node* t) const
{
    while (t->left)
        t = t->left;

    return t;
}
```

While there is a left-subtree, descend to the left




```
Iterator begin() const
{
    if (isEmpty()) return end();

    return Iterator(findMin(root));
}
```

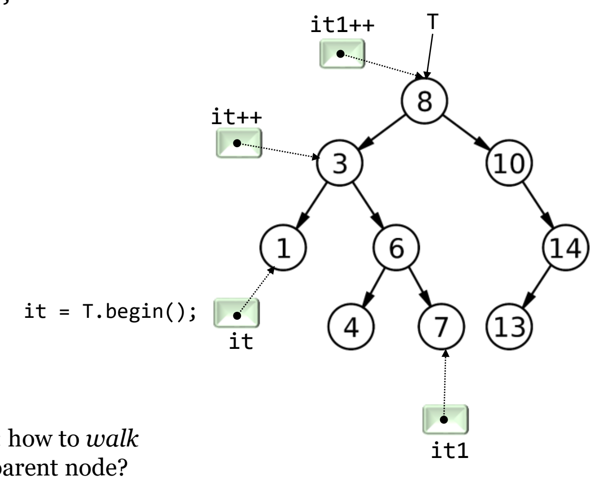

9

Inorder:

`1 3 4 6 7 8 10 13 14`

```
for(auto it = T.begin(); it != T.end(); it++)
    cout << *it << endl;
```

```
auto it1 = find(7);
it1++;
```



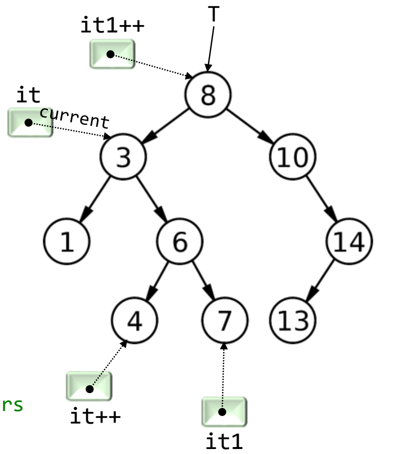
Problem: how to *walk up* to the parent node?

10

BiIterator::operator++()

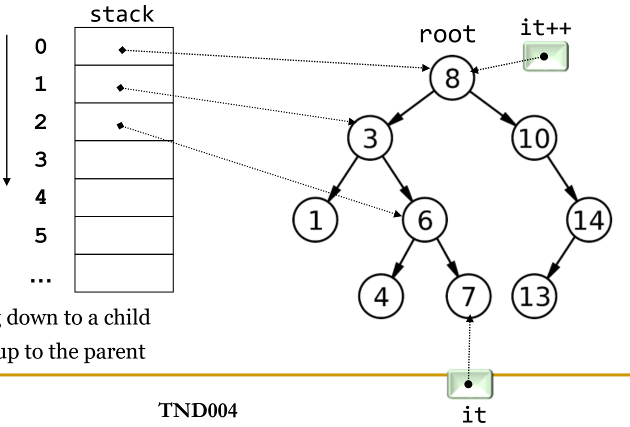
Inorder:
1 3 4 6 7 8 10 13 14

```
BST::Iterator& operator++()
{
     current = find_successor(current);

     return *this;
}


Node* find_successor(Node* t)
{
    if (t != nullptr && t->right)
    {
        return findMin(t->right);
    }
    else //successor is one of the ancestors
    {
        ? ? ? ?
    }
}
```



How to *“walk up”*?

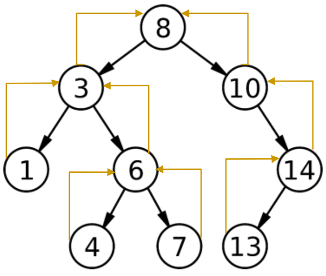
- **Solution 1**
  - Each `Iterator` instance stores a stack
  - Pointers along the way from `root` down to `it` are stored in the stack
- **Disadvantage**: not efficient solution in terms of space usage



push, when moving down to a child
pop, when moving up to the parent

How to “*walk up*”?

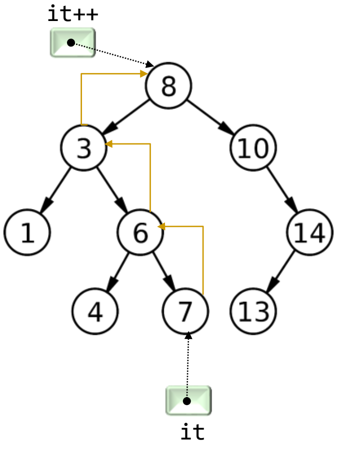
- **Solution 2** -- to be implemented in lab 3
  - Every node stores an extra pointer to its parent node
- **Disadvantage**: extra space required at each node



The parent pointer has to be updated during insertion and removal of a node

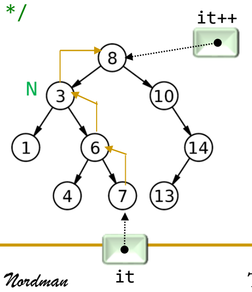
BiIterator::operator++()

```
Iterator& operator++()
{
    current = find_successor(current);

    return *this;
}


Node* find_successor(Node* t)
{
    if (t != nullptr && t->right)
    {
        return findMin(t->right);
    }
    else //successor is one of the ancestors
    {
        ? ? ? ?
    }
}
```



How to "*walk up*"?

```
Node* find_successor(Node* t)
{
    if (t !== nullptr && t->right != nullptr)  //t has a right sub-tree
    {
        return findMin(t->right);
    }
    else  //successor is one of the ancestors
    {
        /*
        Algorithm: climb up using the parent pointer until
        one finds a node N which is left child of its parent
        The parent of N is the successor of node t
        */
    }
}
```

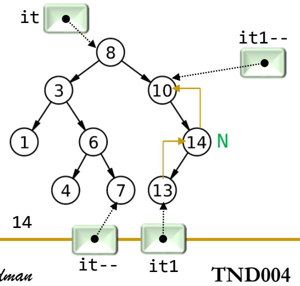



```
BiIterator& operator++()
{
    current = find_successor(current);

    return *this;
}
```


15

How to "*walk up*"?

```
Node* find_predecessor(Node* t)
{
    if (t !== nullptr && t->left != nullptr)  //t has a left sub-tree
    {
        return findMax(t->left);
    }
    else  //predecessor is one of the ancestors
    {
        /*
        Algorithm: climb up using the parent pointer until
        one finds a node N which is right child of its parent
        The parent of N is the predecessor of node t
        */
    }
}
```




```
BiIterator& operator--()
{
    current = find_predecessor(current);

    return *this;
}
```

Inorder:

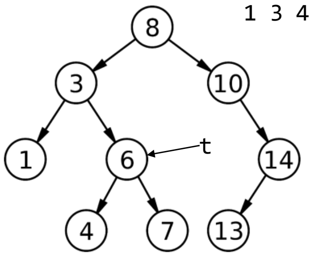
1 3 4 6 7 8 10 13 14

16

Lab 3: exercise 1

- Add a parent pointer to each node
- Re-write the code to insert/remove a node from the tree
- Write a function to find the predecessor and sucessor of a value x

```
std::pair< Comparable, Comparable> find_pred_succ(const Comparable &x) const;
```



Inorder:

`1 3 4 6 7 8 10 13 14`

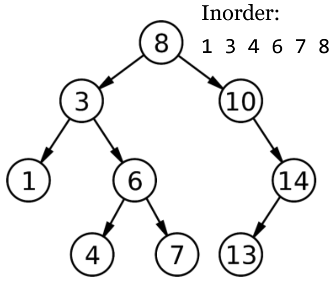
6 is in the tree

```
(a,b) = find_pred_succ(6);
```

```
a ← 4   //largest value smaller than 6
b ← 7   //smallest value larger than 6
```

```
a ← findMax(t->left);
b ← findMin(t->right);
```


Exercise 1: `find_pred_succ`



Inorder:

`1 3 4 6 7 8 10 13 14`

12 is not in the tree

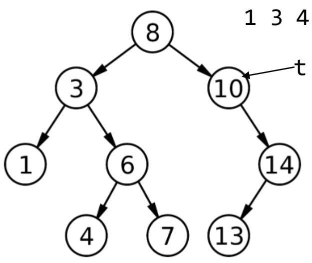
```
(a,b) = find_pred_succ(12);
```

```
a ← 10   //largest value smaller than 12
b ← 13   //smallest value larger than 12
```

	$12 \in [a, b]$
`find_pred_succ(12, a, b, root);`	
`t = root, 12, a = −∞, b = +∞`	$12 \in [-\infty, +\infty]$
`t = t->right, 12, a = 8, b = +∞`	$12 \in [8, +\infty]$
`t = t->right, 12, a = 10, b = +∞`	$12 \in [10, +\infty]$
`t = t->left, 12, a = 10, b = 14`	$12 \in [10,14]$
`t = t->left, 12, a = 10, b = 13`	$12 \in [10,13]$

Lab 3: exercise 1

```
std::pair< Comparable, Comparable> find_pred_succ(const Comparable &x) const;
```



Inorder:

1 3 4 6 7 8 10 13 14

10 is in the tree

```
(a,b) = find_pred_succ(10);
```

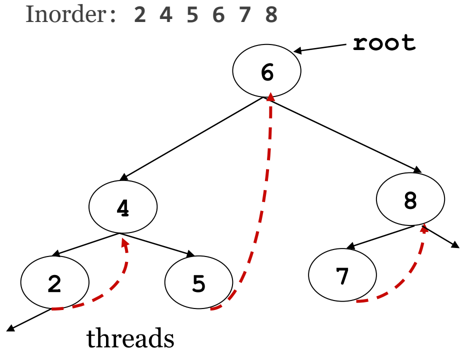
```
a ← 8   //largest value smaller than 10
b ← 13  //smallest value larger than 10
```

```
a ← //find it while descending the tree
b ← findMin(t->right);
```


How to "*walk up*"?

- **Solution 3** -- threaded tree
  - If right sub-tree is empty then use the **right** link to point to the (inorder) successor -- `right` null pointers are used
- **Disadvantage**: *insert* and *remove* are more complicated to program

Inorder: **2 4 5 6 7 8**



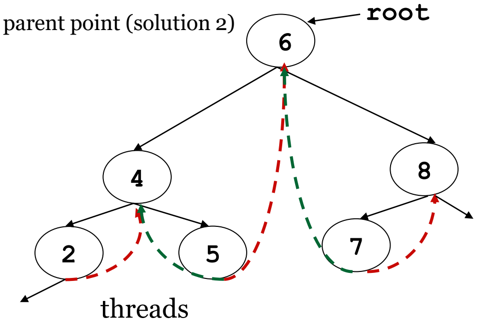
Flag indicating if pointer right is used as a thread

```
Node* find_successor(Node* t)
{
    if (t == nullptr) return nullptr;

    if (t->r_thread == false)
        return findMin(t->right);
    return t->right;
}
```

Threaded Tree

Advantages:

- Two flags are needed in each node to distinguish a **thread** from a usual link *parent →child node* -- can be encoded with 1 byte
- No stack or recursion is needed to perform inorder traversal or find successor/predecessor of a node
- Faster then climb up with parent point (solution 2)



How do we insert and remove from a threaded tree?