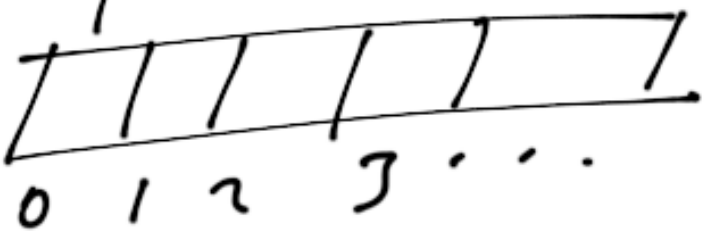
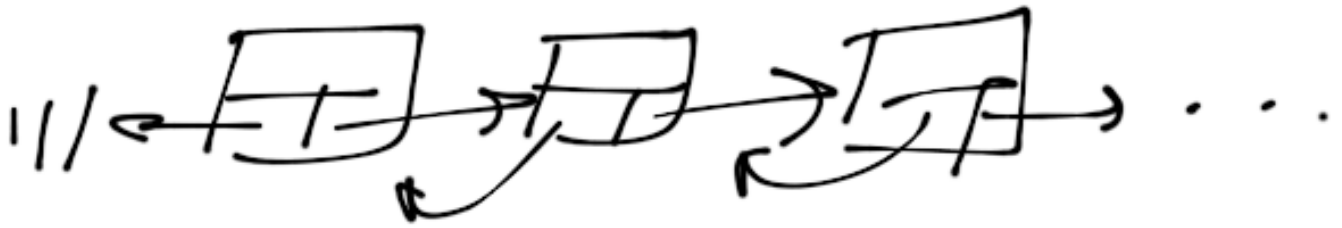
You've seen:

arrays — std::vector<>

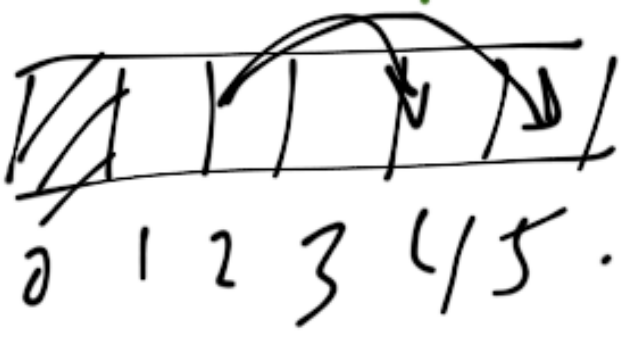


linked lists — std::list<>



binary heap (priority queue)

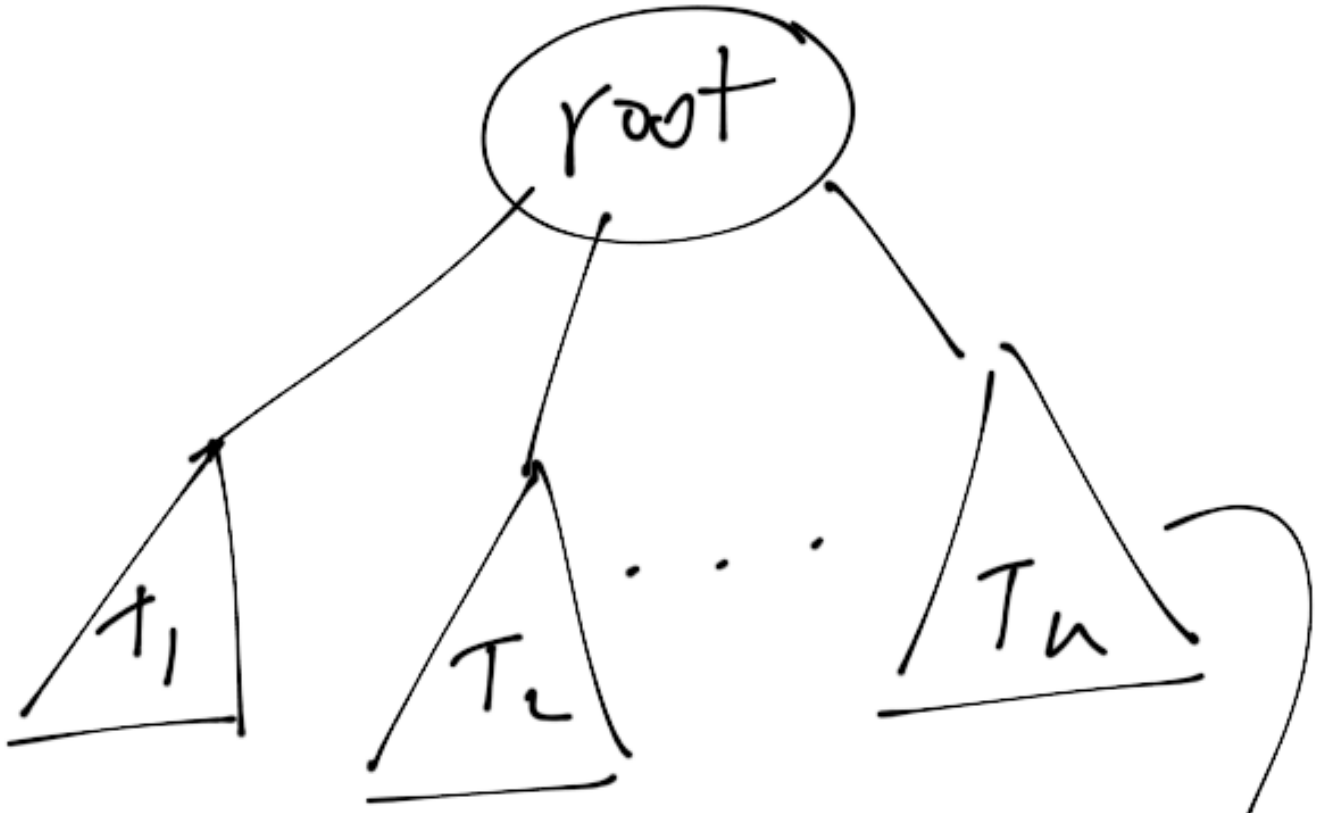
std::priority_queue<>



array acting as tree

- in general (Chp. 4)

Trees



n-ary tree

sub-tree

e.g. Unix file system tree

- n-ary tree: each node can have any # of children

- how to code this up?

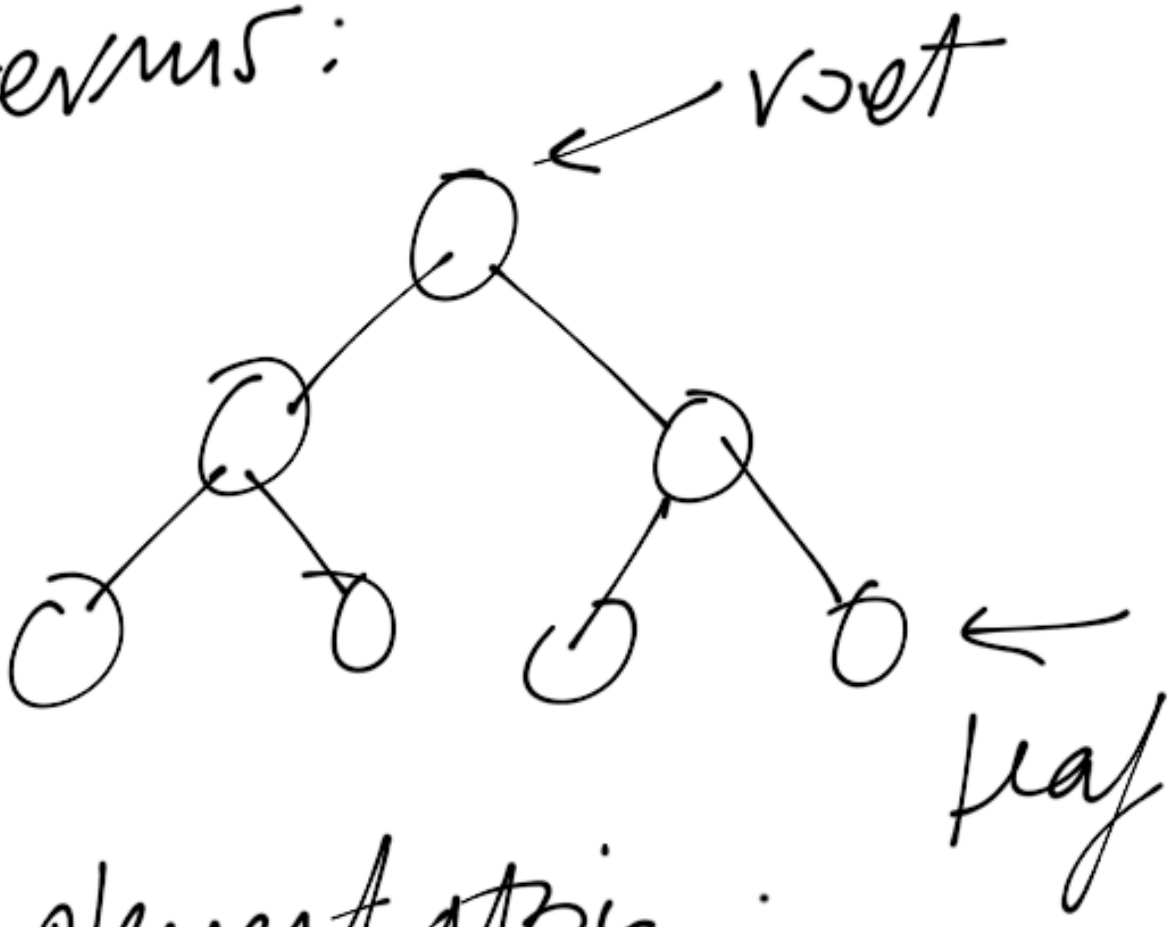
e.g. struct node_t ?

linked list of nodes

T data

list_t nodes

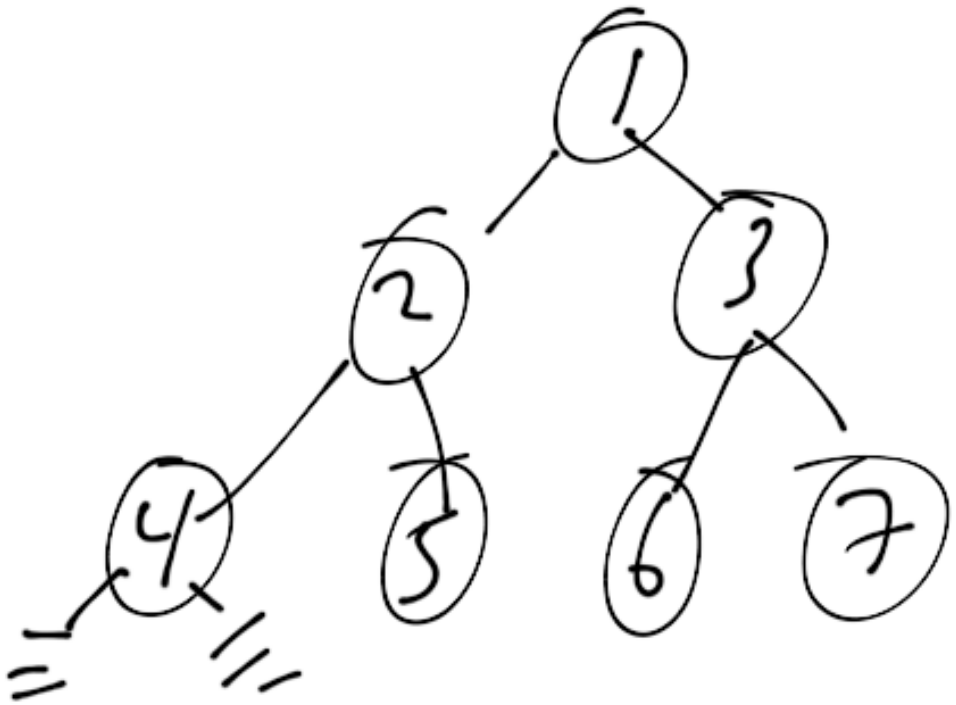
- terms:



- implementation:
(binary tree)

```
struct node_t {
    T data;
    node_t *left, *right;
}
```

- traversals :



pre-order : root, left, right

1, 2, 4, 5, 3, 6, 7

post-order : left, right, root

4, 5, 2, 6, 7, 3, 1

in-order : left, root, right

4, 2, 5, 1, 6, 3, 7

- in-order traversal:
  what we want to do
  to get sorted order
  given a binary search
  tree (BST)

- BST: maintain sorted
  order.

  BSP: for any node
  all smaller items go
  into left subtree

- insertion & deletion must preserve property

(insertion easy
deletion a touch trickier)

- pretty simple overall for BST, but not always effective — tree can degenerate (into a list)

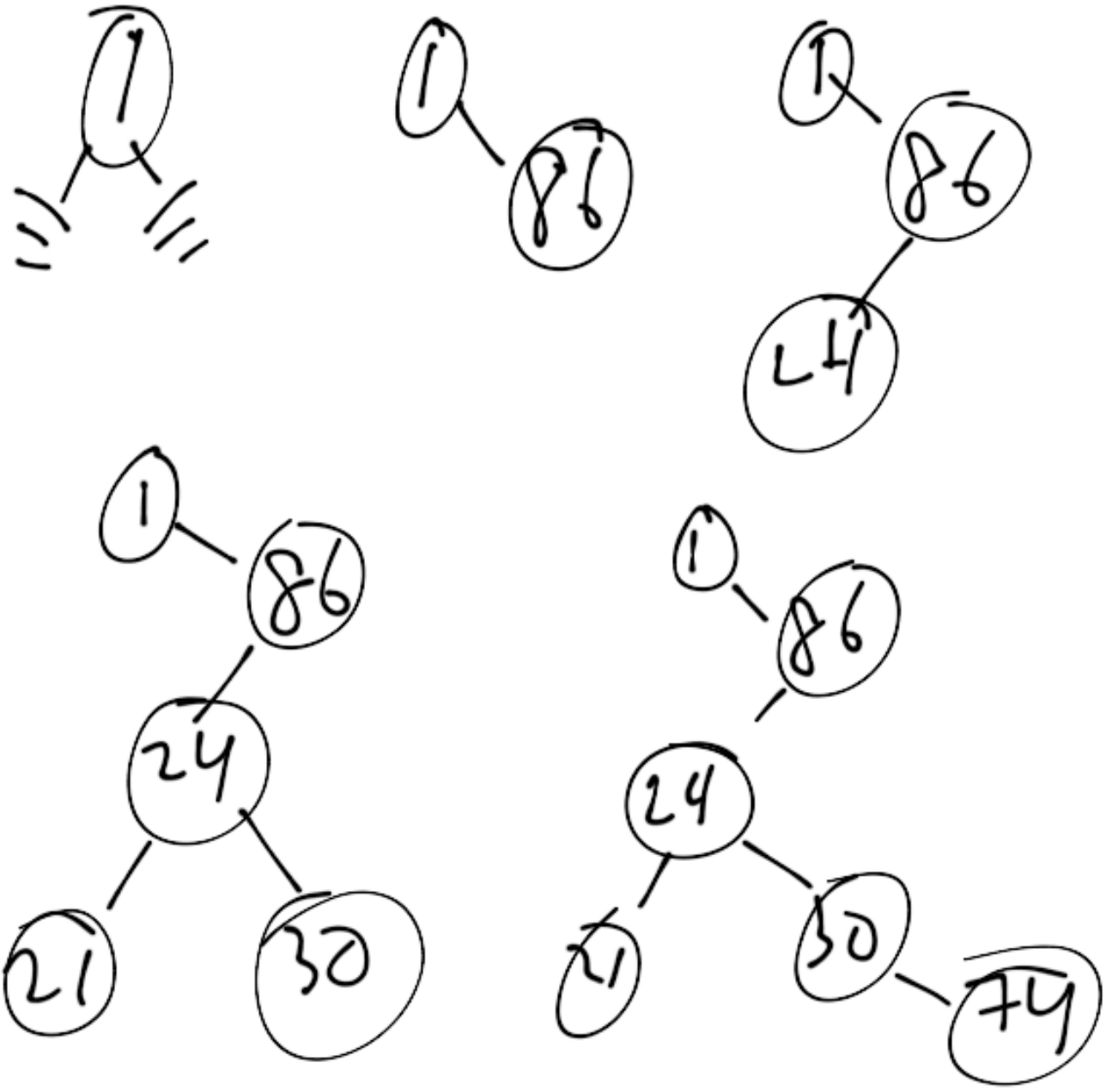
- if balanced, $O(\log n)$ search
- AVL trees: maintain balance, BST do not

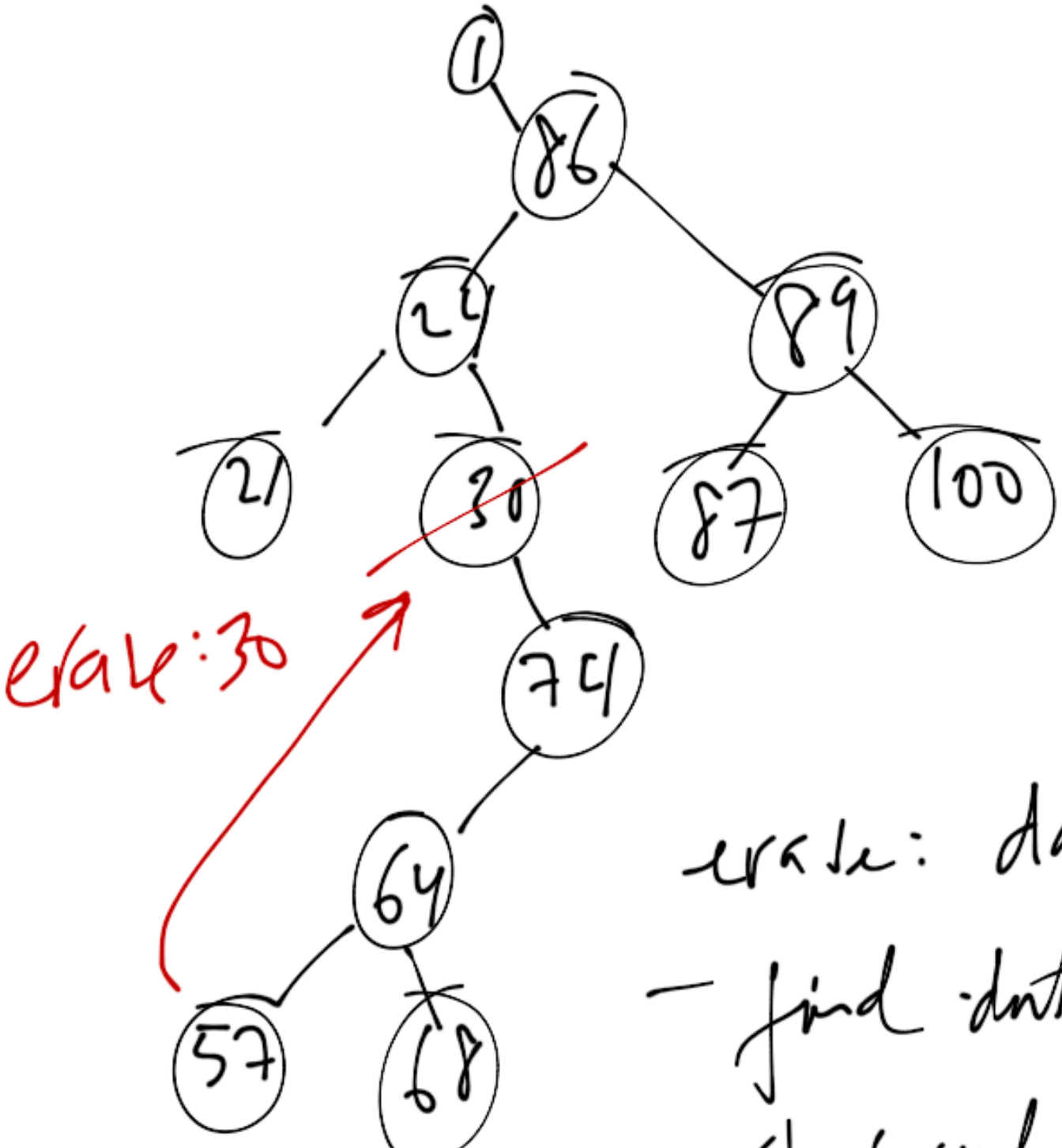
- insertion (BST):

```
if (data < node->data)
    insert(node->left)
else
    insert(node->right)
```

```
if (node == NULL)
    new node(data)
```

insert: 1, 86, 24, 21, 30,
74, 89, 64, 21, 68,
57, 87, 100





erase: data

- find data
- if found,

replace with min of r. subtree

```
template <typename T>
class tree_t {
    private:
    struct node_t
    {
        T data;
        node_t *left, *right;
        node_t(const T& d=T(),
  node_t *l=NULL, node_t *r=NULL)
: data(d), left(l), right(r) {}
};
```

```
public:
friend std::ostream<< <>
    (std::ostream& s,
     const true_t& rhs);
// call inorder(...)
void inorder(std::ostream& s,
        node_t* const& t)
    const;
```

```
// members (public)
bool empty() const;
bool contains(const T& x)
                const;
void insert(const T& x);
void erase(const T& x);
void clear() { clear(root); }
```

public member with no arguments

overloaded private member w/ root

```
private:
node_t *root;
bool contains(const T&,
              node_t *)
              const;
void insert(const T&,
            node_t *);
void erase(const T&,
           node_t **);
void clear(node_t *);
```