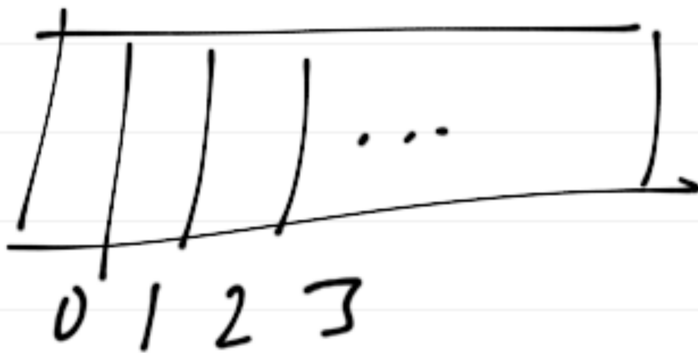
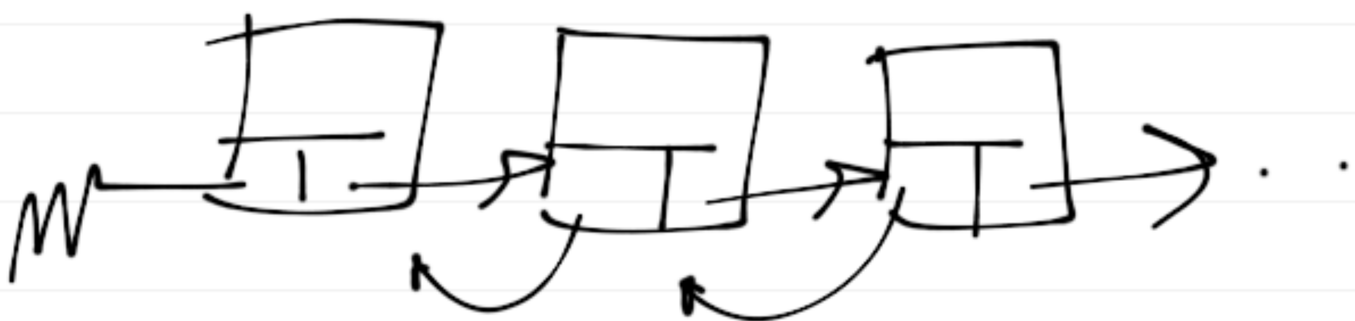
you've seen:

arrays std::vector<>

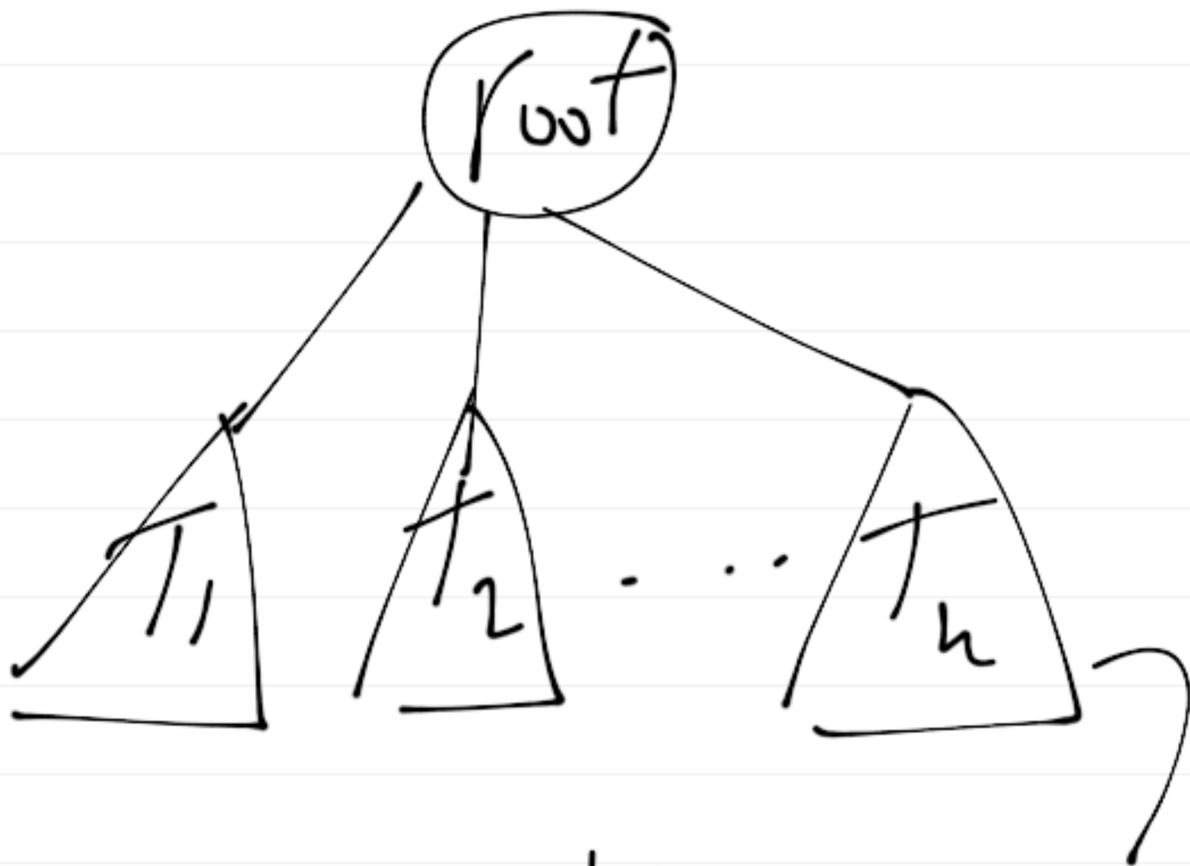


linked lists std::list<>



binary heap std::priority_queue

- now: trees (chp. 4)



e.g. Unix filesystem

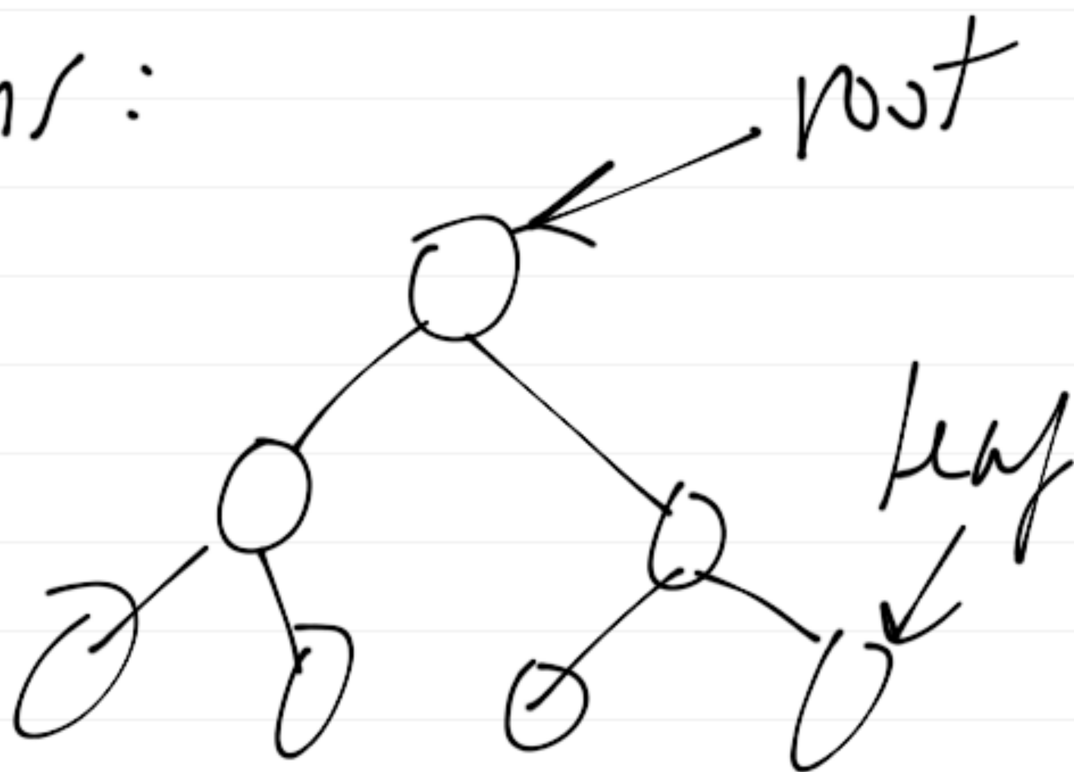
- n-ary tree: each node can have any # of children
- how to code this up?

e.g.

```
struct node_t {
    T data;
    list_t nodes;
}
```

linked list of nodes → list_t nodes

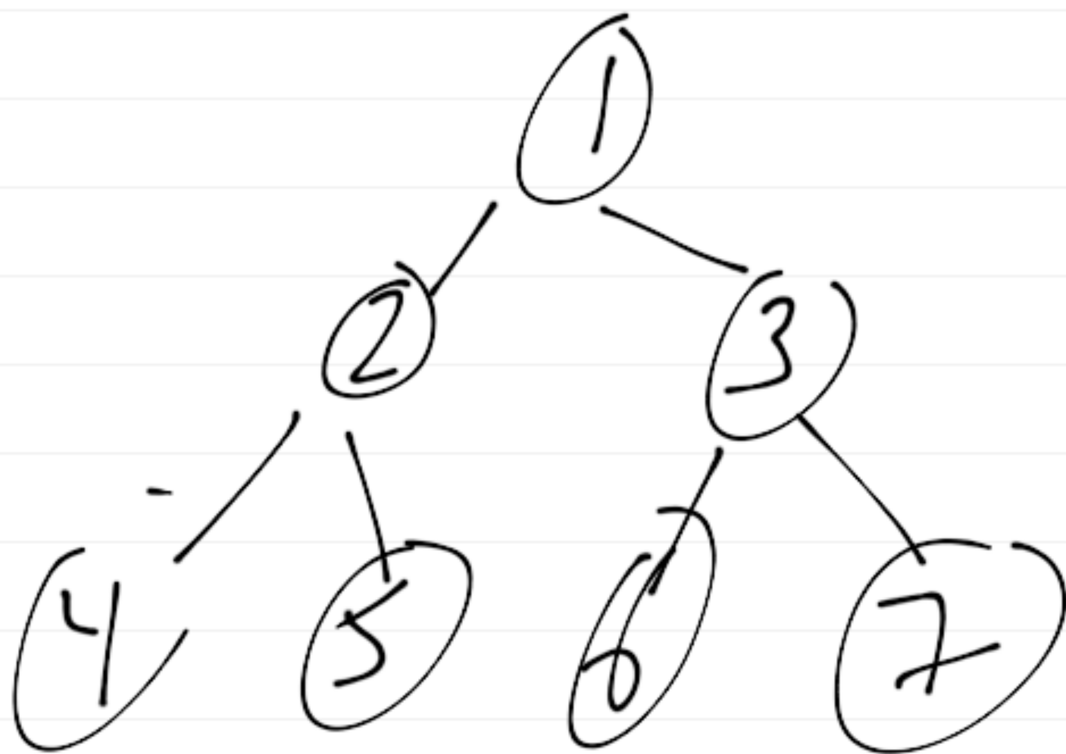
- terms:



- 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
- Property: for any node, all
  smaller items go into
  left subtree

- insertion & deletion must preserve property (insertion easy, deletion a touch trickier)
- fairly straightforward overall, but BST not always effective — tree can degenerate (into a list)

- if unbalanced (degenerate) & in worst case a list, then search for item can take $O(n)$
- if <u>balanced</u>, search for item takes $O(\lg n)$
- AVL tree = balanced BST

insertion into BST:

```
if (node == NULL)
    new node (data)
else if (data < node → data)
    insert (node → left)
else if (data > node → data)
    insert (node → right)
else ; // no-op
```

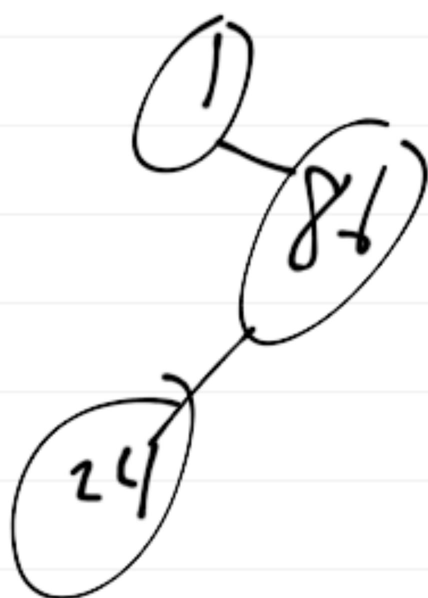
```
insert(const T & x, node_t * &t)
{
    if (t == NULL)
        t = new node_t(x,
                        NULL, NULL)
    else if (x < t->data)
        insert(x, t->left)
    else if (x > t->data)
        insert(x, t->right)
    else ; // do nothing
}
```

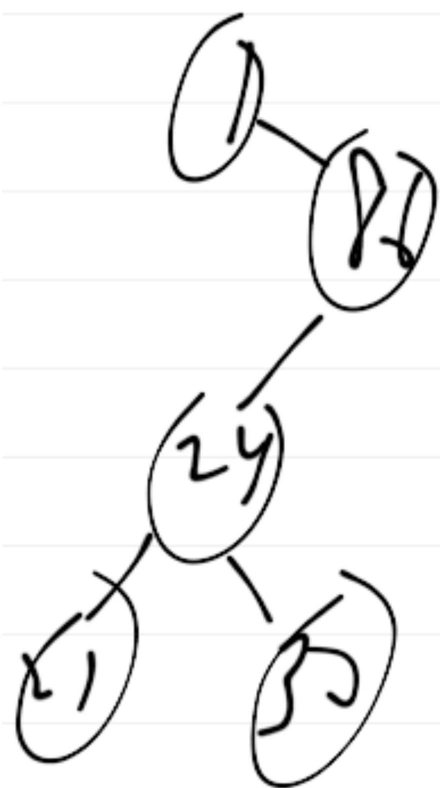
```
insert(const T& x, node_t* &t)
{
  if (t == NULL)
    t = new node_t(x, NULL, NULL)
  else if (x < t->data)
    insert(x, t->left)
  else if (x > t->data)
    insert(x, t->right)
  else; // do nothing
}
```

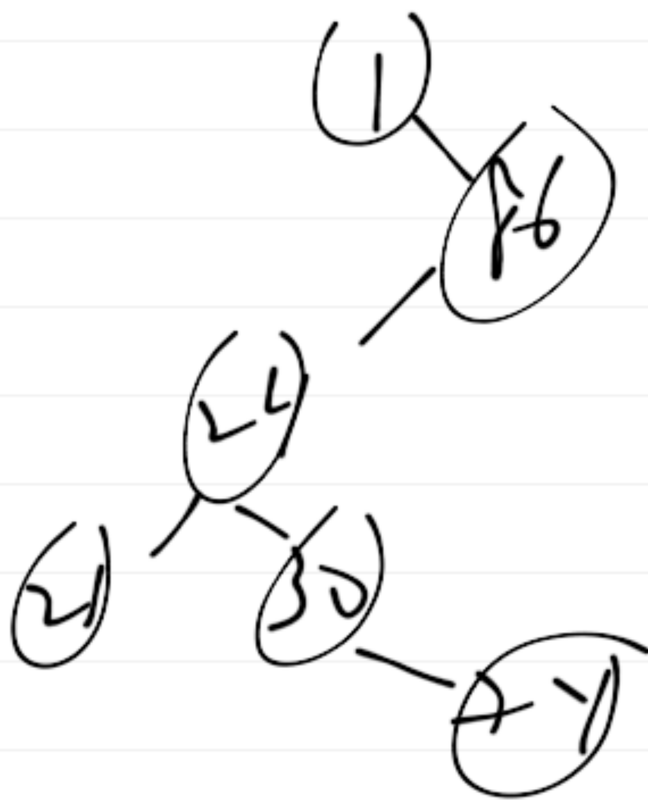
Annotations:
- T — data
- node_t* — node ptr ref
- node_t(...) — node constructor
- < — operator <
- > — operator >
- for type T

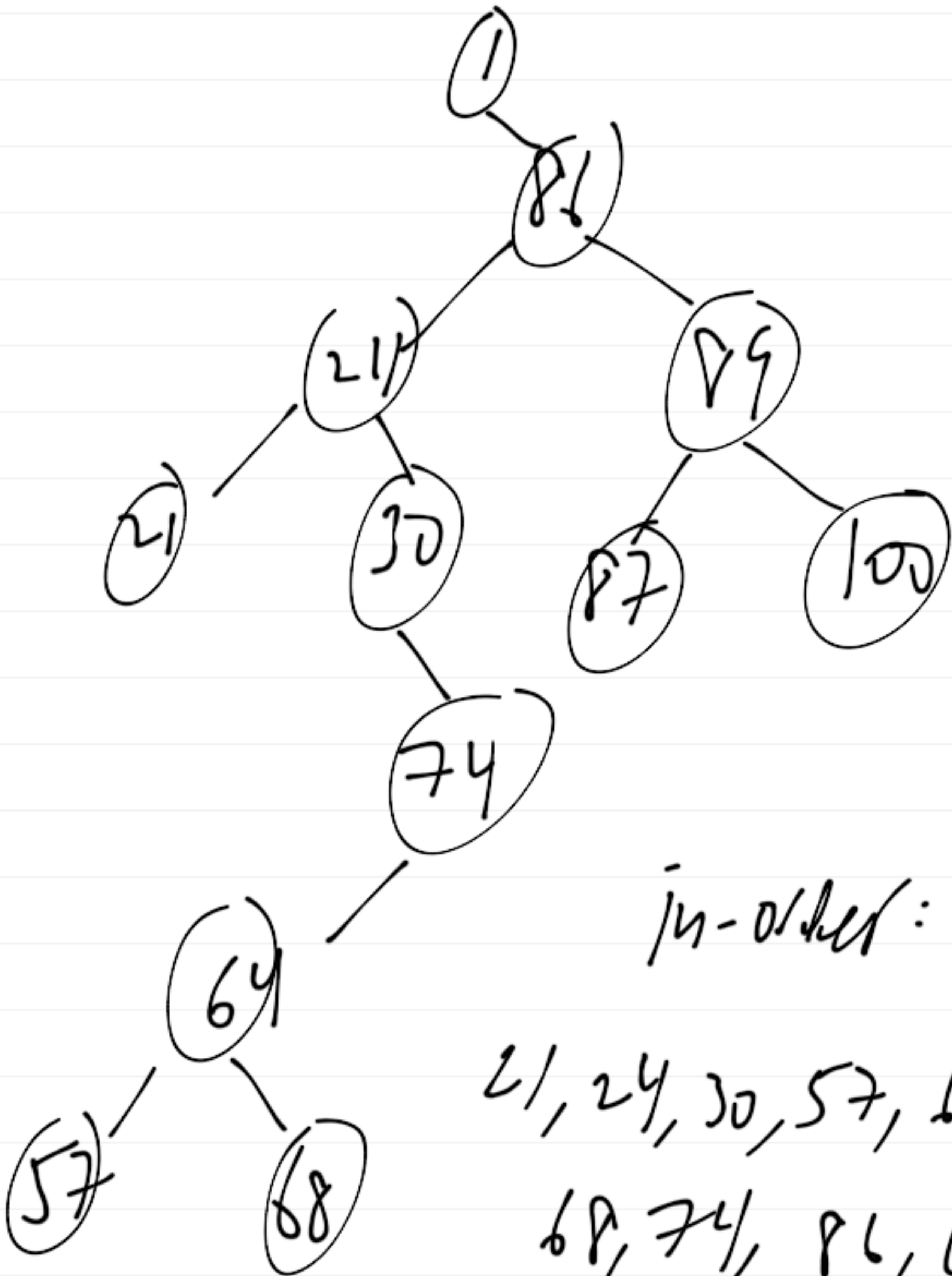
insert: 1, 86, 24, 21, 30, 74,

85, 64, 21, 68, 57, 87, 100





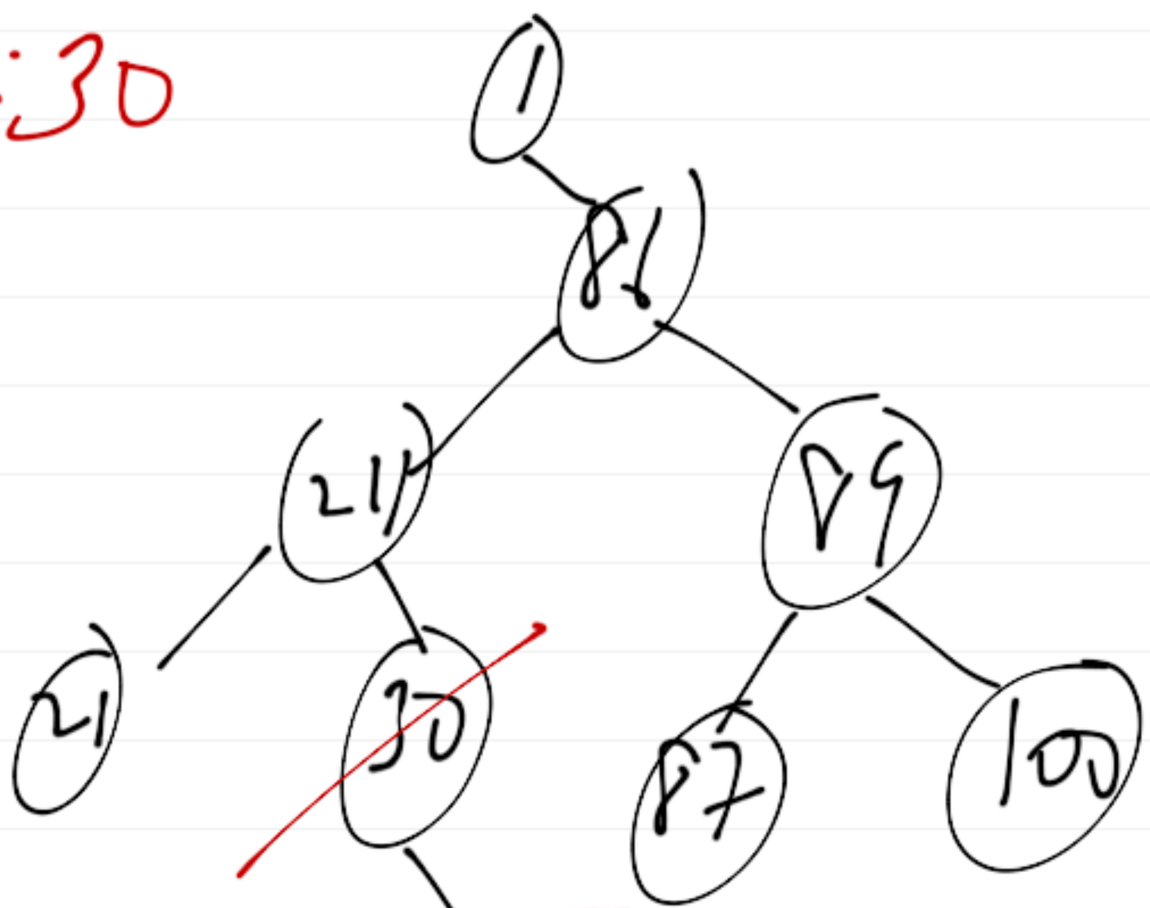




in-order:

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

erase: 30



r.subtree →

erase: data

- find data
- if found, replace with min. of right 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  tree_t & rhs));
  ~~{ // call inorder(...) }~~
```

```
in tree.cpp:
template <typename T>
std::ostream& operator<< (
    std::ostream & s, const
        tree_t <T> & rhs)
{ rhs.inorder (s, rhs.root);
  return s;
```

```
template <typename T>
void tree_t<T>::inorder(
    std::ostream & s,
    node_t * const & t) const
{
    if (t != NULL) {
        inorder(s, t->left);
        s << t->data << " ";
        inorder(s, t->right);
    }
}
```

... meanwhile back in tree.h:

```
// members (public)
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 w/ no args — overloaded private 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 *);
```

Usage:

in main.cpp

```
tree_t<pair_t *> tree;

pair_t A('c', 24);

tree.insert(&A);
```

(&A): no arg. for root