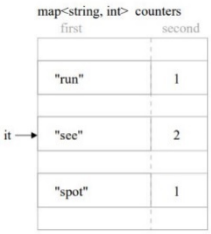
Quick sort:

```
int quickSort(vector<double>& array, int start, int end) {
  int swaps = 0;
  if(start < end) {
    int pIndex = partition(array, start, end, swaps);

    //after each call one number(the PIVOT) will be at its final position
    swaps += quickSort(array, start, pIndex-1);
    swaps += quickSort(array, pIndex+1, end);
  }

  return swaps;
}
```

std::map<key_type, value_type> var_name

std::pair< key_type, value_type>

Map search, insert and erase => 0(logn)



For maps, the [] operator searches the map for the pair containing the key (string) s.

– If such a pair containing the key is **not** there, the operator:
  1. creates a pair containing the key and a default initialized value,
  2. inserts the pair into the map in the appropriate position, and
  3. returns a reference to the value stored in this new pair (the second component of the pair).

  This second component may then be changed using operator++.
– If a pair containing the key **is** there, the operator simply returns a reference to the value in that pair.

std::map<std::string, int>::const_iterator it;

it->first it -> second

m.find(key) m.insert(std::make_pair(key, value));

std::pair<map<key_type, value_type>::iterator, bool>

`void erase(iterator p)` — erase the pair referred to by iterator `p`.

`void erase(iterator first, iterator last)` — erase all pairs from the map starting at `first` and going up to, but not including, `last`.

`size_type erase(const key_type& k)` — erase the pair containing key `k`, returning either 0 or 1, depending on whether or not the key was in a pair in the map

map < string, vector<int> > :: const_iterator p;

typedef map < string, vector<int> > map_vect;

map_vect :: const_iterator p;

set:

for (set<string>::iterator p = words.begin(); p!= words.end(); ++p) cout << *p << endl; s.insert(*it)

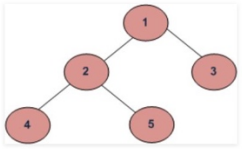
size_type set<key>::erase(const Key& x);

void set<key>::erase(iterator p);

void set<key>::erase(iterator first, iterator last);

insert find 复杂度同 map

```
template <class T>
class TreeNode {
public:
  TreeNode() : left(NULL), right(NULL) {}
  TreeNode(const T& init) : value(init), left(NULL), right(NULL) {}
  T value;
  TreeNode* left;
  TreeNode* right;
};

template <class T> class ds_set;

// -----------------------------------------------------------------------
// TREE NODE ITERATOR CLASS
template <class T>
class tree_iterator {
public:
  tree_iterator() : ptr_(NULL) {}
  tree_iterator(TreeNode<T>* p) : ptr_(p) {}
  tree_iterator(const tree_iterator& old) : ptr_(old.ptr_) {}
  ~tree_iterator() {}
```



Example Tree

Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

```
void Rope::destroy_tree(Node* p)
{       if (!p) return;
        destroy_tree(p->left);
        //move the iterator before delete the Node
        Node* tmp = p->right;
    delete p;
    destroy_tree(tmp);}
Node* Rope::copy_tree(Node* p){
        if (p == NULL) return NULL;
        else{
                Node* copy = new Node;
                copy -> weight = p->weight;
                copy -> value = p->value;
                copy -> left = copy_tree(p->left);
                if (copy->left != NULL){
                        copy->left->parent = copy;
                }
        copy -> right = copy_tree(p->right);
        if (copy->right != NULL){
                        copy->right->parent = copy;
                }
                return copy;
        }
}
```

```
rope_iterator& rope_iterator::operator++()
{
    if (ptr_->right != NULL)
    { // find the leftmost child of the right node
      ptr_ = ptr_->right;
      while (ptr_->left != NULL)
        {
            ptr_ = ptr_->left;
        }
    }
    else
    { // go upwards along right branches...  stop after the first left
      while (ptr_->parent != NULL && ptr_->parent->right == ptr_)
        {
            ptr_ = ptr_->parent;
        }
      ptr_ = ptr_->parent;
    }
    return *this;
}
```

```
template <class T>
const T& FindLargestInTree(TreeNode<T>* root){
  while(root->right){
    root = root->right;
  }
  return root->value;
}

template <class T>
TreeNode<T>* FindSmallestInRange(const T& a, const T& b, TreeNode<T>* root){
  if(!root){
    return NULL;
  }

  T best_value = FindLargestInTree(root);
  TreeNode<T>* ret = FindSmallestInRange(a, b, root, best_value);
  if(best_value >= b){
    return NULL;
  }
  return ret;
```

```
template <class T>
TreeNode<T>* FindSmallestInRange(const T& a, const T& b, TreeNode<T>* root, T& best_value){
```

Solution:

```
  if(!root){
    return NULL;
  }

  TreeNode<T>* left_subtree = FindSmallestInRange(a,b,root->left,best_value);
  TreeNode<T>* right_subtree = FindSmallestInRange(a,b,root->right,best_value);
  if(root->value > a && root->value < best_value){
    best_value = root->value;
    return root;
  }
  else if(left_subtree && left_subtree->value == best_value){
    return left_subtree;
  }
  else if (right_subtree){
    return right_subtree;
  }
  return NULL;

}
```

Tree sort:

```
template <class T>
std::vector<T> TreeSort(TreeNode<T>* root){
  std::vector<T> ret;
  const T& smallest = FindSmallestInTree(root);
  const T& largest = FindLargestInTree(root);

  ret.push_back(smallest);
  TreeNode<T>* find = FindSmallestInRange(ret[ret.size()-1],largest,root);
  while(find){
    ret.push_back(find->value);
    find = FindSmallestInRange(ret[ret.size()-1],largest,root);
  }
  ret.push_back(largest);
  return ret;
}
```

在 string 查找 char:

```
int EInString(const std::string& str, int index){
```

Solution:

```
  if(index>str.length()){
    return 0;
  }
  if(str[index]=='E'){
    return 1 + EInString(str,index+1);
  }
  return EInString(str,index+1);
}
```

	Vector	Singled-linked list	STL list
size	√ O(1)	√ O(n)	O(1)
push_back	√ O(1)	√ O(n)	O(1)
push_front		√ O(1)	O(1)
pop_back	√ O(1)	√ O(n)	O(1)
pop_front		√ O(1)	O(1)
erase	√ O(n)	√ O(1)	O(1)
insert	√ O(n)	√ O(1)	O(1)

```
class Node {
 public:
  Node(int v) : value(v), parent(NULL) {}
  int value;
  Node* parent;
};
```

```
unsigned int v_it, ret_it;
for(v_it=0,ret_it=0 ; v_it<v.size(); v_it++){
  if(i< v[v_it].size()){
    ret[i][ret_it] = v[v_it][i];
    ret_it++;
  }
}
```

```
int count_odd(const TreeNode<int> *p) {
  if (p == NULL) return 0;
  //if (p->value % 2 == 1)
  // return 1 + count_odd(p->left) + count_odd(p->right);
  //else
  //return count_odd(p->left) + count_odd(p->right);
  return (p->value % 2) + count_odd(p->left) + count_odd(p->right);

}
```

```
iterator find(const T& key_value, TreeNode<T>* p) {
  // Implemented in Lecture 17
  if (p == NULL) {
    return end();
  } else if ( p->value == key_value ) {
    return iterator(p);
  } else if ( p->value > key_value ) {
    return find (key_value, p->left);
  } else {
    return find (key_value, p->right);
  }
}
```

```
int erase(T const& key_value, TreeNode<T>* &p) {
  if (!p) return 0;

  // look left & right
 if (p->value < key_value)
    return erase(key_value, p->right);
  else if (p->value > key_value)
    return erase(key_value, p->left);

  // Found the node.  Let's delete it
  assert (p->value == key_value);
  if (!p->left && !p->right) { // leaf
    delete p;
    p=NULL;
    this->size_--;
  } else if (!p->left) { // no left child
    TreeNode<T>* q = p;
    p=p->right;
    assert (p->parent == q);
    p->parent = q->parent;
    delete q;
    this->size_--;
  } else if (!p->right) { // no right child
    TreeNode<T>* q = p;
    p=p->left;
    assert (p->parent == q);
    p->parent = q->parent;
    delete q;
    this->size_--;
  } else { // Find rightmost node in left subtree
    TreeNode<T>* q = p->left;
    while (q->right) q = q->right;
    p->value = q->value;
    // recursively remove the value from the left subtree
    int check = erase(q->value, p->left);
    assert (check == 1);
  }
  return 1;
}
```

```
std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*& p, TreeNode<T>* the_parent) {
  if (!p) {
    p = new TreeNode<T>(key_value);
    p->parent = the_parent;
    this->size_++;
    return std::pair<iterator,bool>(iterator(p,this), true);
  }
  else if (key_value < p->value)
    return insert(key_value, p->left, p);
  else if (key_value > p->value)
    return insert(key_value, p->right, p);
  else
    return std::pair<iterator,bool>(iterator(p,this), false);
}
```

```
template <class T>
void breadth_first(TreeNode<T> *p) {
  std::vector<TreeNode<T>*> current;
  std::vector<TreeNode<T>*> next;

  // handle an empty tree
  if (p != NULL) {
    current.push_back(p);
  }

  // loop over all levels of the tree
  while (current.size() > 0) {
    // loop over all elements on this level
    for (int x = 0; x < current.size(); x++) {
      std::cout << " " << current[x]->value;  // the "do something" part of this traversal
      if (current[x]->left != NULL)
        next.push_back(current[x]->left);
      if (current[x]->right != NULL)
        next.push_back(current[x]->right);
    }
    // switch to the next level, empty next vector to receive following level
    current = next;
    next.clear();
  }
}
```

```
void printPostorder(struct Node* node)

{

    if (node == NULL)

        return;

    printPostorder(node->left);

    printPostorder(node->right);

    cout << node->data << " ";

}

void printInorder(struct Node* node)

{

    if (node == NULL)

        return;

    printInorder(node->left);

    cout << node->data << " ";

    printInorder(node->right);

}

void printPreorder(struct Node* node)

{

    if (node == NULL)

        return;

    cout << node->data << " ";

    printPreorder(node->left);

    printPreorder(node->right);

}
```