







Executing a breadth-first search for the shortest path from root to leaf on a balanced binary search tree is most likely going to be faster BUT ALSO REQUIRE MORE ADDITIONAL MEMORY than a depth-first search on the same tree.

Maintaining the Red-Black property for a BINARY SEARCH TREE, ensures that the data balanced and elements can be accessed in O(log n) time

When the same member function is implemented in more than one class within the inheritance hierarchy, the virtual keyword on the parent class function indicates that the derived class function should be used if it is available.

Depth-first quickly investigate leaf nodes, but if it has made “incorrect” branch decision early in the search, it will take a long time to work back to the point and go down the right branch.

Breadth-first find the solution node with shortest path to root node if there are multiple solution nodes, but it is memory-intensive cuz it must store worst case number of nodes on each level doubles

Preorder: Visit the root – Traverse the left sub-tree – Traverse the right sub-tree

Inorder: Traverse the left sub-tree – Visit the root – Traverse the right sub-tree

Postorder: Traverse the left sub-tree – Traverse the right sub-tree – Visit the root

vector<T> v; list<T> l; queue<T> q; set<type, compare> s;

priority\_queue<T, container<T>, comparison<T> > q1;

map<key\_type, value\_type, key\_compare> m;

pair<first\_type, second\_type> p(fir, sec)

v.push\_back(value) & v.pop\_back() O(1)

v.insert(itr, value) & v.erase(itr) & v.erase(begin, end) O(n)

l.insert(itr, value) & l.erase(itr) & l.erase(begin, end) O(1)

l.push\_front/back(value) & l.pop\_front/back(value) O(1)

l.remove(value) & l.reverse() & l.merge(l2) O(n)

l.sort() & l.sort(comparison) O(n log n)

q.front/back() & q.pop/size/empty() O(1)

q1.top/size/empty() O(1) Return the “biggest” element/Return current # of elements/Return true if q1 is empty

q1.push(value) & q1.pop() O(log n) Add value to q1/Remove biggest value

s.find/count(key) & s.insert(itr, key) & s.insert(key) O(log n) Return itr pointing to key in s or s.end() if not/Return # of items equal to key in s

Return itr pointing to key/Return a pair p.first is an itr pointing to key; p.second is true was not already in s

s.size/empty/begin/end() O(1)

m[key] & m.find(key) & m.insert(pair) O(log n) Return value stored for key/Return itr pointing to key-value pair or m.end() if not in m

m.size/empty/begin/end() O(1)

p.first/second() O(1) typedef tree\_iterator<T> iterator;

template <class T> template <class T>

class tree\_iterator { class ds\_set(

public: tree\_iterator() : ptr\_(NULL) {} public: ds\_set() : root\_(NULL), size\_(0) {}

tree\_iterator(TreeNode<T>\* p) : ptr\_(p) {} ds\_set(const ds\_set<T>& old) : size\_(old.size\_) {

tree\_iterator(const tree\_iterator& old) : ptr\_(old.ptr\_) {} root\_ = this->copy\_tree(old.root\_); }

~tree\_iterator() {} ~ds\_set() { this->destroy\_tree(root\_); root\_ = NULL; }

tree\_iterator& operator=(const tree\_iterator& old) ds\_set& operator=(const ds\_set<T>& old) {

{ ptr\_ = old.ptr\_; return \*this; } if (&old != this) { this->destroy\_tree(root\_);

const T& operator\*() const { return ptr\_->value; } root\_ = this->copy\_tree(old.root\_); size\_ = old.size\_;}

bool operator==(const tree\_iterator& r) { return ptr\_ == r.ptr\_; } return \*this;}

bool operator!=(const tree\_iterator& r) { return ptr\_ != r.ptr\_; } TreeNode<T>\* copy\_tree(TreeNode<T>\* old\_root) {}

private: TreeNode<T>\* ptr\_; }; iterator find(const T& key\_value, TreeNode\* p) {}

std::pair<iterator, bool> insert(const T& key\_value, TreeNode<T>\*& p) {}

int erase(T const& key\_value, TreeNode<T>\* &p) {}

private: TreeNode<T>\* root\_; int size\_;