template <class T>

**class TreeNode** {

public:

TreeNode() : left(NULL), right(NULL), parent(NULL) {}

TreeNode(const T& init) : value(init), left(NULL), right(NULL), parent(NULL) {}

T value;

TreeNode\* left;

TreeNode\* right;

TreeNode\* parent; // to allow implementation of

};

**template <class T> class ds\_set**;

template <class T>

class tree\_iterator {

public:

tree\_iterator() : ptr\_(NULL), set\_(NULL) {}

tree\_iterator(TreeNode<T>\* p, const ds\_set<T> \* s) : ptr\_(p), set\_(s) {}

tree\_iterator(const tree\_iterator& old) : ptr\_(old.ptr\_), set\_(old.set\_) {}

~tree\_iterator() {}

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

const T& operator\*() const { return ptr\_->value; }

friend bool operator== (const tree\_iterator& lft, const tree\_iterator& rgt)

{ return (lft.set\_ == rgt.set\_ && lft.ptr\_ == rgt.ptr\_); }

friend bool operator!= (const tree\_iterator& lft, const tree\_iterator& rgt)

{ return (lft.set\_ != rgt.set\_ || lft.ptr\_ != rgt.ptr\_); }

tree\_iterator<T> & operator++() {

if (ptr\_->right != NULL) { // find the leftmost child of the right node

ptr\_ = ptr\_->right;

while (ptr\_->left != NULL) { ptr\_ = ptr\_->left; }

} else {

while (ptr\_->parent != NULL && ptr\_->parent->right == ptr\_) { ptr\_ = ptr\_->parent; }

ptr\_ = ptr\_->parent; }

return \*this; }

tree\_iterator<T> operator++(int) {

tree\_iterator<T> temp(\*this);

++(\*this);

return temp; }

tree\_iterator<T> & operator--() {

qif (ptr\_ == NULL){

assert( set\_ != NULL);

ptr\_ = set\_ -> root\_;

while ( ptr\_ -> right != NULL) { ptr\_ = ptr\_ -> right; }}

else if ( ptr\_ -> left != NULL ){

ptr\_ = ptr\_ -> left;

while ( ptr\_ -> right != NULL) { ptr\_ = ptr\_ -> right; }

}else {

while ( ptr\_ -> parent != NULL && ptr\_ -> parent -> left == ptr\_ ){

ptr\_ = ptr\_ -> parent;xs }

ptr\_ = ptr\_ -> parent; }

return \*this;}

tree\_iterator<T> operator--(int) {

tree\_iterator<T> temp(\*this);

--(\*this);

return temp; }

private:

TreeNode<T>\* ptr\_;

const ds\_set<T>\* set\_;};

**template <class T>**

**class ds\_set {**

public:

ds\_set() : root\_(NULL), size\_(0) {}

ds\_set(const ds\_set<T>& old) : size\_(old.size\_) {

root\_ = this->copy\_tree(old.root\_,NULL); }

~ds\_set() {

this->destroy\_tree(root\_);

root\_ = NULL;}

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

if (&old != this) {

this->destroy\_tree(root\_);

root\_ = this->copy\_tree(old.root\_,NULL);

size\_ = old.size\_;}

return \*this; }

typedef tree\_iterator<T> iterator;

friend class tree\_iterator<T>;

int size() const { return size\_; }

bool operator==(const ds\_set<T>& old) const { return (old.root\_ == this->root\_); }

void accumulate ( T & string ) const {

ds\_set<T>::iterator itr;

for ( itr = this -> begin(); itr != this -> end(); itr ++ ){

string += \*itr; }

}

iterator find(const T& key\_value) { return find(key\_value, root\_); }

std::pair< iterator, bool > insert(T const& key\_value) { return insert(key\_value, root\_, NULL); }

int erase(T const& key\_value) { return erase(key\_value, root\_); }

// OUTPUT & PRINTING

friend std::ostream& operator<< (std::ostream& ostr, const ds\_set<T>& s) {

s.print\_in\_order(ostr, s.root\_);

return ostr;}

void print\_as\_sideways\_tree(std::ostream& ostr) const {

print\_as\_sideways\_tree(ostr, root\_, 0);}

iterator begin() const {

if (!root\_) return iterator(NULL,this);

TreeNode<T>\* p = root\_;

while (p->left) p = p->left;

return iterator(p,this);}

iterator end() const { return iterator(NULL,this);}

bool sanity\_check() const {

if (root\_ == NULL) return true;

if (root\_->parent != NULL) {

return false;}

return sanity\_check(root\_);}

private:

TreeNode<T>\* root\_;

int size\_;

// PRIVATE HELPER FUNCTIONS

TreeNode<T>\*  **copy\_tree**(TreeNode<T>\* old\_root, TreeNode<T>\* the\_parent) {

if (old\_root == NULL)

return NULL;

TreeNode<T> \*answer = new TreeNode<T>();

answer->value = old\_root->value;

answer->left = copy\_tree(old\_root->left,answer);

answer->right = copy\_tree(old\_root->right,answer);

answer->parent = the\_parent;

return answer; }

void **destroy\_tree**(TreeNode<T>\* p) {

if (!p) return;

destroy\_tree(p->right);

destroy\_tree(p->left);

delete p; }

iterator find(const T& key\_value, TreeNode<T>\* p) {

if (!p) return end();

if (p->value > key\_value)

return find(key\_value, p->left);

else if (p->value < key\_value)

return find(key\_value, p->right);

else

return iterator(p,this);}

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);}

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);

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;

int check = erase(q->value, p->left);

assert (check == 1);}

return 1;}

void print\_in\_order(std::ostream& ostr, const TreeNode<T>\* p) const {

if (p) {

print\_in\_order(ostr, p->left);

ostr << p->value << "\n";

print\_in\_order(ostr, p->right); } }

void print\_as\_sideways\_tree(std::ostream& ostr, const TreeNode<T>\* p, int depth) const {

if (p) {

print\_as\_sideways\_tree(ostr, p->right, depth+1);

for (int i=0; i<depth; ++i) ostr << " ";

ostr << p->value << "\n";

print\_as\_sideways\_tree(ostr, p->left, depth+1);}}

bool sanity\_check(TreeNode<T>\* p) const {

if (p == NULL) return true;

if (p->left != NULL && p->left->parent != p) {

return false;}

if (p->right != NULL && p->right->parent != p) {

return false; }

return sanity\_check(p->left) && sanity\_check(p->right);

} };

#endif

template <class T>

void breadth\_first(TreeNode<T> \*p) {

//Iterative solution

std::list<TreeNode<T>\*> current\_row;

std::list<TreeNode<T>\*> next\_row;

if(p){current\_row.push\_back(p);}

while(!current\_row.empty()){

//Proccess the current row

while(!current\_row.empty()){

TreeNode<T>\*& tmp = current\_row.front();

std::cout << tmp->value << " ";

if(tmp->left){next\_row.push\_back(tmp->left);}

if(tmp->right){

next\_row.push\_back(tmp->right);}

current\_row.pop\_front(); //Advance the current "queue"}

std::cout << std::endl;

//Make the next row the current row

current\_row = next\_row;

next\_row.clear();}}

15.1 What is the order notation for the number of operations in our non-linear word search algorithm, assuming s = search word length, and h \* w = dimensions of the board? O(w•h•8^s)

15.2 order notation for the running time of Quicksort? Running time: O(n log n) Memory: O(1)

16.1Which of the following statements is least true? **If you can code something with fewer keystrokes it is always better software.**

16.2 Which of the following is false for the STL pair class? **The first item in an STL pair is always const and cannot be changed**

16.3 Which of the following is true for the STL map iterators? **None of the above (hint, data is accessed in the….)**

17.1 Which of the following statements is false? **Any valid C++ type can be used as the key(first) part of an STL map.**

17.2 what is the order natation to look up student X’s grade in course?

**O(logs + k)**

17.3 what is the order notation to make a list of all students who have taken course Y? **O(s + k)**

17.4 Which of the following statement is false about a binary search tree holding the integer 1-10? **If 7 is the parent of 5, then 5 is the right child of 7 if 1**

**If '5' is not in the left subtree of '7', then '7' must be in the right subtree of '5'.**

18.1 How many exactly balanced binary search trees exist with the numbers 4.5 9.8 3.5 … **1, 7!**

18.2 Which of the following statements about STL container types is true? **A program that uses an STL set can easily be changed to use an STL map instead, with no performance impact.**

18.3 order notation of find\_smailest function we just wrote,assuming the tree has n nodes in it and a height h. **O(h)**

18.4 assuming the tree has n nodes in it and a height h?

**O(n)**

18.5 What is the height (# of levels) of the binary search tree that has post order traversal: 1 2 3 4 5 6 7. **7**

19.1 What is the post-order traversal of this tree? **1 3 2 5 7 6 4**

19.2 what is the height of the binary search tree that has pre-order traversal 1 2 3 4 5 6 7? **7**

19.3 what is the traversal order of the destroy\_tree function we wrote earlier?

**Poster order**

19.4 What is the sum of the last 4 elements in a breadth first traversal of an exactly balanced binary search tree with the elements 1-7? **16**

19.5 **running time: best: O(n) Average: O(n) Worst:O(n) Memory usuage: Best O(1) Average: O(n) Worst: O(n)**

19.6 Which of the following statements about tree iterator is false? **If the tree iterator is pointing at the node containing the last element in sorted order, that node must be a leaf node.**

19.7 Consider a BPlusTree node with b = 5 and keys: "ant", "bear", "bee", "cod";

**The fourth child (children[3])**

19.8 Consider a BPlusTree node with b = 4. What is the maximum number of nodes that be on level i? **b^(i)**

20.1 Which of these statements is true about B+ trees? **) B+ trees are a good choice when getting a node takes a long time, but things are fast once the node is in memory.**

20.2 with the tree with n nodes, fot the recursive tree height algorithm, ….? **Running time:O(n) O(n) O(n) memory:O(log n) O(log n) O(n)**

20.3 for the tree with n nodes, for a breadth-first shortest path to leaf node algorithm…? **RT: O(1) O(n) O(n) MU: O(1) O(n) O(n)**

20.4 For either version of the tree iterator operator++ function, for a balanced tree with n elements what is the order notation for the worst case call to operator++….? **S\_wor: O(logn) S\_avg: O(1) total: O(n)**

20.5 Which of the following statements about tree iterators is false?

**If the tree iterator is pointing at the node containing the last element in sorted order, that node must be a leaf node.; When we……..**

21.1: **2,4&6**

21.2 Which of the following statement about operator overloading is true? **If a member function takes in as an argument a second object of the class type, it has access to the private member variable of both the “this” object and the argument object.**

21.3 is false? **You can overload operators for every symbol on your**

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

Until all nodes are traversed −

Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

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

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

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

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

Step 3 − Visit root node.

void pre\_order\_traversal(struct node\* root) {

if(root != NULL) {

printf("%d ",root->data);

pre\_order\_traversal(root->leftChild);

pre\_order\_traversal(root->rightChild);}}

void inorder\_traversal(struct node\* root) {

if(root != NULL) {

inorder\_traversal(root->leftChild);

printf("%d ",root->data);

inorder\_traversal(root->rightChild);}}

void post\_order\_traversal(struct node\* root) {

if(root != NULL) {

post\_order\_traversal(root->leftChild);

post\_order\_traversal(root->rightChild);

printf("%d ", root->data);}}