COMP 2012H Final Exam - Fall 2020 - HKUST

6:10

1)

(a)

key values

0 80

1 EMPTY

2 60

3 12

4 70

5 35

6 66

7 89

8 18

9 69

(b)

= (1 + 1 + 1 + 1 + 1 + 3 + 2 + 3 + 2) / 9

= 1.67  
4.67

(c)

iii, v

2)

After the insertion of 13, 5 & 20: 13, 5, 20

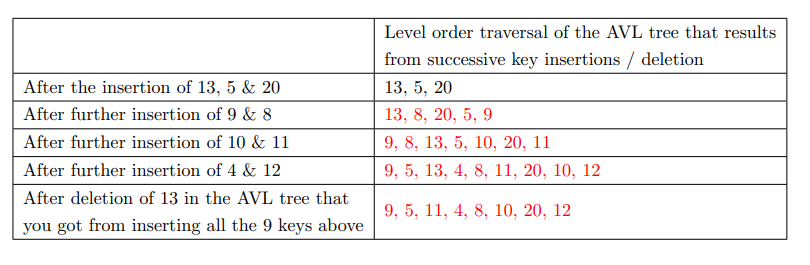
After further ins ertion of 9 & 8: 9 5 13 8 20

After further insertion of 10 & 11: 9 5 13 8 10 20 11

After further insertion of 4 & 12: 9 5 13 4 8 11 20 10 12

After deletion of 13 in the AVL tree that

you got from inserting all the 9 keys above: 9 5 20 4 8 11 10 12



3)

\*\*\* archer1 construction \*\*\*

Weapon(int)

Bow(int, int)

Bow(const Bow&) lacking Weapon(const Bow&) base class,

goes second because class member

Unit(char) goes first because this is base class constructor

Archer(char, const Bow&)

˜Bow()

˜Weapon()

\*\*\* archer2 copy-construction \*\*\*

Archer(const Archer&) unneeded, also lacking everything

Weapon(const Weapon&)

Bow(const Bow&)

Unit(const Unit&) again, this first

Archer(const Archer&)

˜Archer()

˜Unit()

˜Bow()

˜Weapon()

\*\*\* archer2 destruction \*\*\*

˜Archer()

˜Unit()

˜Bow()

˜Weapon()

~Archer()

~Bow()

~Weapon()

~Unit()

\*\*\* before main() returns \*\*\*

˜Archer()

˜Unit()

˜Bow()

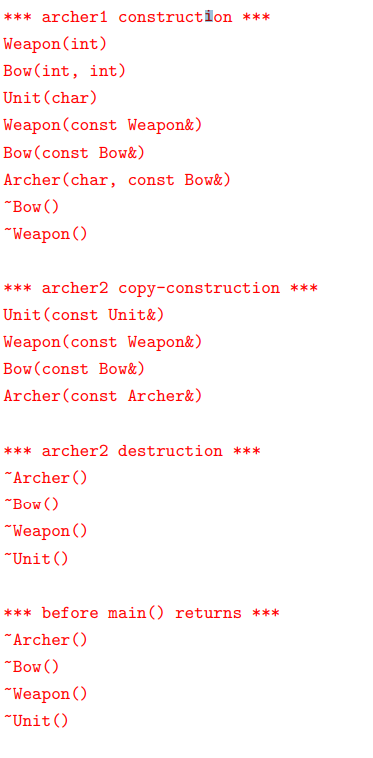
˜Weapon()

~~˜Archer()~~  archer2 already destroyed

~~˜Unit()~~

~~˜Bow()~~

~~˜Weapon()~~



4)

(a)

class LSCS {

private:

int max\_so\_far = 0

int max\_ending\_here = 0

public:

void operator()(const int& elm) {

max\_ending\_here = max\_ending\_here + elm;

if (max\_so\_far < max\_ending\_here) max\_so\_far = max\_ending\_here;

if (max\_ending\_here < 0) max\_ending\_here = 0;

}

int get\_result() {

return max\_so\_far;

}

}

(b)

LSCS lcsc;

for\_each(arr.begin(), arr.end(), lscs);

WRONG: LCSC lscs = for\_each(arr.begin(), arr.end(), lscs);

7:01

5)

(a)

Online\_Lesson::Online\_Lesson(string content, int num\_video\_on)

: Lesson(content), num\_video\_on(num\_video\_on), num\_interactions(0) {}

(b)

bool Online\_Lesson::add\_interaction(Online\_Interaction i) {

if (num\_interactions >= MAX\_NUM\_INTERACTIONS) return false;

interactions[num\_interactions++] = i;

return true;

}

(c)

bool Online\_Lesson::run() {

print();

cout << "----- Run Online Lessons -----" << endl;

cout << "Interactions: ";

for (int i = 0; i < num\_interactions; i++) {

switch (interactions[i]) {

case Online\_Interaction::YES\_NO:

cout << "YES\_NO";

break;

case Online\_Interaction::POLL:

cout << "POLL";

break;

case Online\_Interaction::CHAT:

cout << "CHAT";

break;

}

if (i < num\_interactions - 1) {

cout << ", ";

}

}

cout << endl;

cout << "Number of videos turned on: " << num\_video\_on << endl;

} cout << endl;

(d)

Course::Course(string code)

: code(code), lessons(new Lesson\*[MAX\_LESSONS]), num\_lessons(0) {}

(e)

Course::Course(const Course& course)

: Course(course.code) {

\*this = course;

}

(f)

Course::~Course() {

for (int i = 0; i < num\_lessons; i++) {

delete lessons[i];

}

delete [] lessons;

}  
okay

(g)

bool Course::addLesson(Lesson\* lesson) {

if (num\_lessons >= MAX\_LESSONS) return false;

if (typeid(\*lesson) == typeid(Online\_Lesson)) {

lessons[num\_lessons] = new Online\_Lesson(\*dynamic\_cast<Online\_Lesson\*>(lesson));

} else {

lessons[num\_lessons] = new F2F\_Lesson(\*dynamic\_cast<F2F\_Lesson\*>(lesson));

}

num\_lessons++;

return true;

}

(h)

Course& Course::operator=(const Course& course) {

if (&course != this) {

for (int i = 0; i < num\_lessons; i++) {

delete lessons[i];

}

delete [] lessons; not really necessary

code{course.code};

lessons{new Lesson\*[MAX\_LESSONS]}; not really necessary

num\_lessons{course.num\_lessons};

num\_lessons = 0;

~~for (int i = 0; i < course.num\_lessons; i++) {~~

~~if (typeid(course.lessons[i]) == typeid(Online\_Lesson)) {~~

~~lessons[i] = new Online\_Lesson(\*dynamic\_cast<Online\_Lesson\*>(course.lessons[i]));~~

~~} else {~~

~~lessons[i] = new F2F\_Lesson(\*dynamic\_cast<F2F\_Lesson\*>(course.lessons[i]));~~

~~}~~

addLesson(course.lessons[i]);

}

return \*this;

} else {

return \*this;

}

}

(i)

void Course::run() const {

cout << "Course: " << code << endl;

for (int i = 0; i < ~~course.~~num\_lessons; i++) {

lessons[i]->run();

}

}

7:41

6)

(a)

template <typename T>

BSTNode<T>\* BST::search\_helper(BSTNode<T>\* p, const T& query) const {

if (p) {

if (p->val > query) return search\_helper(p->left, query);

else if (p->val < query) return search\_helper(p->right, query);

else return p;

}

return nullptr;

}

okay

(b)

template <typename T>

BSTNode<T>\* BST::min\_helper(BSTNode<T>\* p) const {

if (p) {

BSTNode<T>\* curPtr = p;

while (curPtr->left) {

curPtr = curPtr->left;

}

return curPtr;

}

return nullptr;

}

okay

(c)

template <typename T>

void BST::list\_helper(BSTNode<T>\* p, std::ostream& os) const {

if (p) {

list\_helper(p->left, os);

os << p->val; << std::endl

list\_helper(p->right, os);

}

}

(d)

template <typename T>

void BST::insert\_helper(BSTNode<T>\* p, const T& element) {

if (root) {

if (p) {

if (p->val > element)

if (p->left) insert\_helper(p->left, element)

else {

p->left = new BSTNode<T>(element);

p->left->parent = p;

}

else

if (p->right) insert\_helper(p->right, element)

else {

p->right = new BSTNode<T>(element);

p->right->parent = p;

}

}

} else { // if root is nullptr

root = new BSTNode<T>(element); // root has no parent

}

}

okay

(e)

template <typename T>

void BST::remove\_helper(BSTNode<T>\* p) {

if (p == nullptr) return;

if (p->right) {

BSTNode<T>\* min\_node = min\_helper(p->right);

p->val = min\_node->val;

remove\_helper(min\_node); correct

} else {

BSTNode<T>\*& parent;

if (p == root) parent = root;

else if (p == p->parent->left) parent = p->parent->left;

else if (p == p->parent->right) parent = p->parent->right;

parent = p->left;

delete p;

}

}

(f)

template <typename T>

T BST::successor(const T& query) const {

BSTNode<T>\* qNode = search\_helper(root, query);

while (qNode) { but only applies to qNode

if (qNode->right) { finding min of right subtree is correct

BSTNode<T>\* min\_node = min\_helper(qNode->right);

return min\_node->val;

}

qNode = qNode->parent;

}

return query;

}

all values greater than q must be either

- in its right subtree, or

- in a parent node (the node’s right subtree must be greater than the parent node)