CS225 Assignment 6

Group 18: Xu Ke, Wang Dingkun, Qian shuaicun, Hua Bingshen

March 2021

1 Assignment 6 Exercise 1

1.1

Please see the code in the attached file.

1.2

Please see the code in the attached file.

2 Assignment 6 Exercise 2

2.1

Please see the code in the attached file.

2.2

In this algorithm, the part of searching delete value is same, so we just need to compare the merge and re-insert part. Obviously, in merge part the time complexity is $O(h)$, but in re-insert part, the time complexity will be $O(nh)$ for this method will go through every elements under the deleted node. So the merge method is much better than re-insert.

3 Assignment 6 Exercise 3

3.1 the iterative implementation

3.1.1 one possible implementation (normal but complex)

For the AVL tree, the iterative implementation of insert and delete operation can be achieved shown as the following code, which is conventional but very complex:

```
/* the iterative insertion function */
template<class T> void AVL<T>::insert(const T& item)
{
        if(header->parent == NULL)
        {
                insertLeaf(item,header,header->parent);
                header->left = header->parent;
                header->right = header->parent;
                return header->parent;
        }//insert at the root
        else
        {
                Link parent = header, child = header->parent,ancestor = NULL;
                while(child!=NULL)
```

```
			{
				parent = child;
				if(child->balanceFactor !='=')
					ancestor = child;
				if(compare(item,child->item))
					child = child->left;
				else
					child = child ->right;
			}
			if(compare(item,parent->item))
			{
				insertLeaf(item,parent,parent->left);
				fixAfterInsertion(ancestor,parent->left);
				if(header->left == parent)
					header->left = parent->left;
				return parent->left;
			}//insert at the left side of the parent
			else
			{
				insertLeaf(item,parent,parent->right);
				fixAfterInsertion(ancestor,parent->right);
				if(header->right == parent)
					header->right = parent->right;
				return parent->right;
			}//insert at the right side of the parent
	}//the tree is not empty
}//	insert

template<class T> void AVL<T>::insertLeaf(const T& item,Link parent,Link& child)
{
	child = new tree_node;
	child->balanceFactor = '=';
	child->isHeader = false;
	child->item = item;
	child->left = NULL;
	child->right = NULL;
	child->parent = parent;
	node_count++;
}//insertLeaf

template<class T> void AVL<T>::fixAfterInsertion(Link ancestor,Link inserted)
{
	Link root = header->parent;
	T item = inserted->item;
	if(ancestor == NULL)
	{
		if(compare(item ,root->item))
			root->balanceFactor = 'L';
		else
			root->balanceFactor = 'R';
		adjustPath(root,inserted);
	}//Case 1:all ancestor have balance factor ''=
	else if((ancestor->balanceFactor =='L'&& !compare(item,ancestor->item)) ||
		(ancestor->balanceFactor =='R' && compare(item,ancestor->item)))
	{
		ancestor->balanceFactor = '=';
```

```
            adjustPath(root, inserted);
    }//Case 2:insert at the child tree opposite to the balance factor of the ancestor
    else if(ancestor->balanceFactor == 'R' && !compare(item, ancestor->right->item))
    {
            ancestor->balanceFactor = '=';
            rotateLeft(ancestor);
            adjustPath(ancestor->parent, inserted);
    }//Case 3:insert at the right child tree of the right child tree of the ancestor
    else if(ancestor->balanceFactor == 'L'&& compare(item, ancestor->left->item))
    {
            ancestor->balanceFactor = '=';
            rotateRight(ancestor);
            adjustPath(ancestor->parent, inserted);
    }//Case 4:insert at the left child tree of the left child tree of the ancestor
    else if(ancestor->balanceFactor == 'L' && !compare(item, ancestor->left->item))
    {
            rotateLeft(ancestor->left);
            rotateRight(ancestor);
            adjustLeftRight(ancestor, inserted);
    }//Case 5:insert at the right child tree of the left child tree of the ancestor
    else
    {
            rotateRight(ancestor->right);
            rotateLeft(ancestor);
            adjustRightLeft(ancestor, inserted);
    }//Case 6:insert at the left child tree of the right child tree of the ancestor
}//     fixAfterInsertion

/* the iterative delete function */
template<class T> void AVL<T>::delete(T item)
{
    if(item.nodePtr->parent->parent == item.nodePtr)
    //item is located at the root node
            deleteLink(itr.nodePtr->parent->parent);
    else if (item.nodePtr->parent->left == item.nodePtr)
    //item is located at left child
            deleteLink(itr.nodePtr->parent->left);
    else        //item is located at right child
            deleteLink(itr.nodePtr->parent->right);
}//     delete

template<class T> void AVL<T>:: deleteLink(Link& link)
{
    if(link->left == NULL || link->right== NULL)     //the linked node has at most one child
            prune(link);
    else
    {
```

```
                deleteSuccessor(link);
        }//     the linked node has two child
}//     deleteLink

template<class T> void AVL<T>:: deleteSuccessor(Link& link1)
{
        T successor;
        Link link = link1->right;
        if(link->left == NULL)
        {
                successor = link->item;
                link1->item = successor;
                prune(link);
        }//the left tree of link is empty
        else
        {
                Link temp = link;
                while(temp->left != NULL)
                        temp = temp->left;
                successor = temp->item;
                link1->item = successor;
                prune(temp->parent->left);
        } //the left tree of link is not empty, move downward to the most left side
            of link->right, assign the value to successor and delete it.
}//deleteSuccessor
```

3.1.2 another possible implementation (using stack to simplify)

For the AVL tree, the interactive implementation of insert and delete operation can be achieved using stack which greatly simplify the process and the sample code is shown as following:

```
template<class T> void AVL<T>:: Insert (AVLNode<Type>* &rt, const Type& x)//rt is
    the root node
{
        AVLNode<Type>* pr = NLL://father node
        AVLNode<Type>* t = rt;//child node
        stack<AVLNode<Type>*> st;
        while(t != NULL)//find suitable place to insert
    {
        if(x == t->data)
            return;
        pr=t;
        st. push(pr) ://record the path

        if(x < t->data)
            t =t->leftChild;
        else
            t = t->rightChild;
    }
    t = new AVLNode<Type>(x);
    assert(t != NULL);

    if(rt = NLL)//if root is empty, insert directly
    {
        rt=t;
        return;
```

```
    }
    if(x < pr->data)//if not root
        pr->leftChi1d = t:
    else
        pr->rightChild= t;

    while(!st.empty())
    {//trace back all the nodes in the stack, and judge whether they are balanced after the insersion

        pr = st.top();
        st.pop();

        if(pr->leftChild == t)
        //mark bf=right-left,thus insert at left, child tree becomes higher, bf decrease, vice versa.
            pr->bf--;
        else
            pr->bf++;

    //Will not influence the balance of the whole AVL Tree and bf
        if(pr->bf == 0)
            break;
        else if(pr->bf==1 ||pr->bf==-1)
        //this node is balanced cannot guarantee the balance of other nodes in the stack, thus track back the previous one

            t = pr;

        else//not balance
    {
        if(r->bf < 0)
        {
            if(t->bf < 0)
            {
                RotateR(pr);
            }
            else
            {
                RotateLR(pr);
            }
        }
        else
        {
            if(t->bf < 0)
            {
                RotateRL(pr);
            }
            else
            {
                RotateL(pr);
            }
        }
        break;
        }
    }//After rotation, need to set the child tree to the father node of the
```

```
        previous child tree
    if(st. empty0)
    //stack is empty means that trace to the root node
        rt= pr;
    else
    {
        AVLNode<Type> *s = st. top();
    //reconnect, the element on the top on the stack is the father node of this
        tree, find the corresponding place and connect
        if(pr->data < s->data)
            s->leftChild = pr;
        else
            s->rightChild = pr;
    }
}
```

3.2 the provided recursive implementation

For the AVL tree, the provided recursive implementation of insert and delete operation achieve the purpose shown as the following code:

```
/* the recursive insertion function */
template<class T> void AVL<T>::insert(T item)
{
    root = _insert(root, item);
    return;
}
template<class T> avlnode<T> *AVL<T>::_insert(avlnode<T> *pt, T val)
{
    if (pt == 0)  // if the tree is empty, we have to create a root node
    {
        avlnode<T> *newnode = new avlnode<T>;
        (*newnode).setdata(val);  // the stored value is the one given as argument
        (*newnode).setbalance(0);  // the balance must be 0
        // note that left and right pointer are 0 by default
        /* for the upward propagation of balance changes (and rotations, if
           necessary) we initialise the bad child and bad grandchild */
        bchild = newnode;
        bgrandchild = 0;
        /* mode indicates, if balances need to be adjusted; a value false means
           that we are done */
        mode = true;
        return newnode;
    }
    if (val == (*pt).getdata())
    {
        /* the first case is the do-nothing case, when the given value already
           occurs in the AVL tree */
        mode = false;
        return pt;
    }
    if (val < (*pt).getdata()) // the case for insertion into the left successor
        tree
    {
        avlnode<T> *pt_new;
```

```
        /* the recursive call returns a pointer to an updated left successor tree; it remains to adjust the balance */
        pt_new = _insert((*pt).getleft(), val);
        (*pt).setleft(pt_new);
        /* the first case is the do-nothing case; the insertiion into the left successor did not alter the height */
        if (mode == false)
            return pt;
        else
        {
            /* first compute the new balance, i.e. decrement it, as the insertion was done in the left successor tree */
            int newbal = (*pt).getbalance() - 1;
            /* if the new balance is -1, 0 or 1, only the balance needs to be updated */
            if (newbal <= 1 && newbal >= -1)
            {
                (*pt).setbalance(newbal);
                /* if the new balance is 0, no more changes further up the tree are necessary */
                if (newbal == 0)
                    mode = false;
                else
                {
                    /* otherwise, the bad child and bad grandchild need to be moved one step up the tree */
                    bgrandchild = bchild;
                    bchild = pt;
                }
                return pt;
            }
            /* this leaves the case, when the old balance was already -1;
             the first case covers the bad grandchild being the left successor of the bad child; i.e., a single right rotation is required */
            if ((*bchild).getleft() == bgrandchild)
            {
                avlnode<T> *newnode;
                /* the new balance values 0, 0 are determined by the previous analysis; a single right rotation produces these values */
                newnode = rotateright(pt, bchild, 0, 0);
                /* as the root of the new subtree has balance 0, no more balance changes are needed */
                mode = false;
                return newnode;
            }
            /* the second case covers the bad grandchild being the right successor of the bad child; i.e., a left rotation with child and grandchild followed by a right rotation with parent and the former grandchild are required */
            avlnode<T> *newnode1;
            avlnode<T> *newnode2;
            /* again, new balance values are determined by the previous analysis; a left-right rotation produces these values */
            int c = 0, n = 0;
            if (val < (*bgrandchild).getdata())
                n = 1;
```

```
            else if (val > (*bgrandchild).getdata())
                c = -1;
            newnode1 = rotateleft(bchild, bgrandchild, c, 0);
            newnode2 = rotateright(pt, newnode1, n, 0);
            /* again, as the root of the new subtree has balance 0, no more balance
                changes are needed */
            mode = false;
            return newnode2;
        }
    }
    else // if (val > (*pt).getdata())
        // the dual case for insertion into the right successor tree
    {
        avlnode<T> *pt_new;
        /* the recursive call returns a pointer to an updated right successor tree;
            it remains to adjust the balance */
        pt_new = _insert((*pt).getright(), val);
        (*pt).setright(pt_new);
        /* the first case is the do-nothing case; the insertiion into the right
            successor did not alter the height */
        if (mode == false)
            return pt;
        else
        {
            /* first compute the new balance, i.e. increment it, as the insertion
                was done in the right successor tree */
            int newbal = (*pt).getbalance() + 1;
            if (newbal <= 1 && newbal >= -1)
            /* if the new balance is -1, 0 or 1, only the balance needs to be
                updated */
            {
                (*pt).setbalance(newbal);
                /* if the new balance is 0, no more changes further up the tree are
                    necessary */
                if (newbal == 0)
                    mode = false;
                else
                {
                    /* otherwise, the bad child and bad grandchild need to be moved
                        one step up the tree */
                    bgrandchild = bchild;
                    bchild = pt;
                }
                return pt;
            }
            /* this leaves the case, when the old balance was already 1;
             the first case covers the bad grandchild being the right successor of
                the bad child; i.e., a single left rotation is required */
            if ((*bchild).getright() == bgrandchild)
            {
                avlnode<T> *newnode;
                /* the new balance values 0, 0 are determined by the previous
                    analysis; a single left rotation produces these values */
                newnode = rotateleft(pt, bchild, 0, 0);
                /* as the root of the new subtree has balance 0, no more balance
                    changes are needed */
```

```
                mode = false;
                return newnode;
            }
            /* the second case covers the bad grandchild being the left successor
               of the bad child; i.e., a right rotation with child and grandchild
               followed by a left rotation with parent and the former grandchild
               are required */
            avlnode<T> *newnode1;
            avlnode<T> *newnode2;
            /* again, new balance values are determined by the previous analysis; a
                right-left rotation produces these values */
            int c = 0, n = 0;
            if (val < (*bgrandchild).getdata())
                c = 1;
            else if (val > (*bgrandchild).getdata())
                n = -1;
            newnode1 = rotateright(bchild, bgrandchild, c, 0);
            newnode2 = rotateleft(pt, newnode1, n, 0);
            /* again, as the root of the new subtree has balance 0, no more balance
                changes are needed */
            mode = false;
            return newnode2;
        }
    }
}

/* the recursive delete function */
template<class T> void AVL<T>::remove(T item)
{
    root = _delete(root, item);
    return;
}

template<class T> avlnode<T> *AVL<T>::_delete(avlnode<T> *pt, T val)
{
    // nothing needs to be done for an empty tree
    if (pt == 0)
        return pt;
    /* the first case occurs, when the sought value has been found */
    if (val == (*pt).getdata())
    {
        /* in case there is no left successor tree, the result of the delete is the
            right successor tree */
        if ((*pt).getleft() == 0)
        {
            mode = true;
            return (*pt).getright();
        }
        /* in case there is no right successor tree, the result of the delete is
           the left successor tree */
        if ((*pt).getright() == 0)
        {
            mode = true;
            return (*pt).getleft();
        }
        /* if both left and right successor trees are not empty, we use the
```

```
            auxiliary function findswapleft to swap the value to be deleted with the
             maximum in the left successor tree; in addition, the node containing
            this maximum is deleted, and the whole left successor tree is
            reorganised to become again an AVL tree */
        avlnode<T> *newnode;
        newnode = findswapleft(pt, (*pt).getleft());
        (*pt).setleft(newnode);
        /* still the balance needs to be adjusted; this is done in the same as
            below in the recursive case covering a deletion in the left successor
            tree */
        if (mode == false)  // no change of height, no action required
            return pt;
        else
        {
            // compute the new balance (increment)
            int newbal = (*pt).getbalance() + 1;
            /* if the new balance is -1, 0 or 1 only the new balance needs to be
                stored */
            if (newbal <= 1 && newbal >= -1)
            {
                (*pt).setbalance(newbal);
                /* in addition, a new balance != 0 indicates that the height has
                    not been altered, so no more changes are needed further up the
                    tree */
                if (newbal != 0)
                    mode = false;
                return pt;
            }
            /* if the old balance was already 1, rotations become necessary; for
                these determine the bad child and bad grandchild in the right
                successor tree */
            bchild = (*pt).getright();
            /* if the bad child had balance 0, only a single left rotation is
                needed */
            if ((*bchild).getbalance() == 0)
            {
                avlnode<T> *newnode;
                /* the new balance values are those that must result from such a
                    single left rotation */
                newnode = rotateleft(pt, bchild, 1, -1);
                /* as the root of the new subtree has balance !=0, the height has
                    not been altered, so no changes are need further up the tree
                    indicated by mode */
                mode = false;
                return newnode;
            }
            /* also, if the bad child had balance 1, only a single left rotation is
                 needed */
            if ((*bchild).getbalance() == 1)
            {
                avlnode<T> *newnode;
                /* the new balance values are those that must result from such a
                    single left rotation */
                newnode = rotateleft(pt, bchild, 0, 0);
                /* in this case the new subtree has balance 0, so the height has
                    been altered; further changes are needed further up the tree */
```

```
                return newnode;
            }
            /* the remaining case concerns a bad child with balance -1; in this case a right rotation followed by a left rotation are needed */
            bgrandchild = (*bchild).getleft();
            avlnode<T> *newnode1;
            avlnode<T> *newnode2;
            /* the new balance values are those that must result from such a left-right rotation */
            int c = 0, n = 0;
            if ((*bgrandchild).getbalance() == 1)
                n = -1;
            if ((*bgrandchild).getbalance() == -1)
                c = 1;
            newnode1 = rotateright(bchild, bgrandchild, c, 0);
            newnode2 = rotateleft(pt, newnode1, n, 0);
            /* in this case also the new subtree has balance 0, so the height has been altered; further changes are needed further up the tree */
            return newnode2;
        }
    }
    /* as long as the sought value has not been found a recursive descent is required; the first case continues the search in the left successor tree */
    if (val < (*pt).getdata())
    {
        avlnode<T> *pt_new;
        /* the recursive call returns an updated left successor tree, in which the sought value (if in there) has been deleted */
        pt_new = _delete((*pt).getleft(), val);
        (*pt).setleft(pt_new);
        /* mode indicates, if balance adjustments are still needed; the first case is the no-action case */
        if (mode == false)
            return pt;
        else
        {
            /* otherwise, first compute the new balance (increment) */
            int newbal = (*pt).getbalance() + 1;
            /* if the new balance is -1, 0 or 1 only the new balance needs to be stored */
            if (newbal <= 1 && newbal >= -1)
            {
                (*pt).setbalance(newbal);
                /* in addition, a new balance != 0 indicates that the height has not been altered, so no more changes are needed further up the tree */
                if (newbal != 0)
                    mode = false;
                return pt;
            }
            /* if the old balance was already 1, rotations become necessary; for these determine the bad child and bad grandchild in the right successor tree */
            bchild = (*pt).getright();
            /* if the bad child had balance 0, only a single left rotation is needed */
```

```
            if ((*bchild).getbalance() == 0)
            {
                avlnode<T> *newnode;
                /* the new balance values are those that must result from such a single left rotation */
                newnode = rotateleft(pt, bchild, 1, -1);
                /* as the root of the new subtree has balance !=0, the height has not been altered, so no changes are needed further up the tree indicated by mode */
                mode = false;
                return newnode;
            }
            /* also, if the bad child had balance 1, only a single left rotation is needed */
            if ((*bchild).getbalance() == 1)
            {
                avlnode<T> *newnode;
                /* the new balance values are those that must result from such a single left rotation */
                newnode = rotateleft(pt, bchild, 0, 0);
                /* in this case the new subtree has balance 0, so the height has been altered; further changes are needed further up the tree */
                return newnode;
            }
            /* the remaining case concerns a bad child with balance -1; in this case a right rotation followed by a left rotation are needed */
            bgrandchild = (*bchild).getleft();
            avlnode<T> *newnode1;
            avlnode<T> *newnode2;
            /* the new balance values are those that must result from such a left-right rotation */
            int c = 0, n = 0;
            if ((*bgrandchild).getbalance() == 1)
                n = -1;
            if ((*bgrandchild).getbalance() == -1)
                c = 1;
            newnode1 = rotateright(bchild, bgrandchild, c, 0);
            newnode2 = rotateleft(pt, newnode1, n, 0);
            /* in this case also the new subtree has balance 0, so the height has been altered; further changes are needed further up the tree */
            return newnode2;
        }
    }
    /* the second case continues the search in the right successor tree */
    else // if (val > (*pt).getdata())
    {
        avlnode<T> *pt_new;
        /* the recursive call returns an updated right successor tree, in which the sought value (if in there) has been deleted */
        pt_new = _delete((*pt).getright(), val);
        (*pt).setright(pt_new);
        /* mode indicates, if balance adjustments are still needed; the first case is the no-action case */
        if (mode == false)
            return pt;
        else
```

```
        /* otherwise, first compute the new balance (decrement) */
        {
            int newbal = (*pt).getbalance() - 1;
            /* if the new balance is -1, 0 or 1 only the new balance needs to be stored */
            if (newbal <= 1 && newbal >= -1)
            {
                (*pt).setbalance(newbal);
                /* in addition, a new balance != 0 indicates that the height has not been altered, so no more changes are needed further up the tree */
                if (newbal != 0)
                    mode = false;
                return pt;
            }
            /* if the old balance was already -1, rotations become necessary; for these determine the bad child and bad grandchild in the left successor tree */
            bchild = (*pt).getleft();
            /* if the bad child had balance 0, only a single right rotation is needed */
            if ((*bchild).getbalance() == 0)
            {
                avlnode<T> *newnode;
                /* the new balance values are those that must result from such a single right rotation */
                newnode = rotateright(pt, bchild, -1, 1);
                /* as the root of the new subtree has balance !=0, the height has not been altered, so no changes are needed further up the tree indicated by mode */
                mode = false;
                return newnode;
            }
            /* also, if the bad child had balance -1, only a single right rotation is needed */
            if ((*bchild).getbalance() == -1)
            {
                avlnode<T> *newnode;
                /* the new balance values are those that must result from such a single right rotation */
                newnode = rotateright(pt, bchild, 0, 0);
                /* in this case the new subtree has balance 0, so the height has been altered; further changes are needed further up the tree */
                return newnode;
            }
            /* the remaining case concerns a bad child with balance 1; in this case a left rotation followed by a right rotation are needed */
            bgrandchild = (*bchild).getright();
            avlnode<T> *newnode1;
            avlnode<T> *newnode2;
            /* the new balance values are those that must result from such a right-left rotation */
            int c = 0, n = 0;
            if ((*bgrandchild).getbalance() == 1)
                c = -1;
            if ((*bgrandchild).getbalance() == -1)
```

```
                    n = 1;
                newnode1 = rotateleft(bchild, bgrandchild, c, 0);
                newnode2 = rotateright(pt, newnode1, n, 0);
                /* in this case also the new subtree has balance 0, so the height has been altered; further changes are needed further up the tree */
                return newnode2;
            }
        }
}
```

3.3 difference analysis

For the iterative implementation of AVL tree insert and delete operation, we claim that there are mainly two ways to achieve the purpose. The first way is conventional but very complex. The way to do the insert and delete operation with AVL balanced tree nodes is to first use the method of BST(binary search tree) to insert and delete, and then maintain the inserted or deleted tree to meet the requirement of AVL balanced tree, that is the balance must be -1 or 0 or 1. If using the iterative idea to implement, there are a lot of things to consider while deleting, and the code thus is messy and disgusting. The second way is using stack to trace and record the path while traversing thus simplifying the process of adjusting balance. But the disadvantage of this is very clear, that is it will use much memory location to perform stack functionality.

For the recursive implementation of AVL tree insert and delete operation, it seems to be very easy to implement compared with the iterative implementation, and the code is clearer and readable. However, due to the nested calling while doing the recursion, it maybe time-consuming. And by the way, it also requires much memory location.

4 Assignment 6 Exercise 4

4.1

for an AVL tree satisfies the median property and it has odd number of nodes, then l(v) will be the median of the set. And the number of nodes in the left subtree must be equal to the right subtree. And we notice that the two subtrees rooted at v also satisfies the median property, so the symmetric structure is about the root v.Then the AVL property of root node v is satisfied. Induction shows that all AVL trees which satisfies the median property with odd number of nodes are perfectly balanced. For an AVL tree satisfies the median property and it has odd number of nodes, then the l(v) must be the high median of the set.

4.2

For insert(x) operation, we developed a recursive algorithm.

1) If the root of the AVL tree is NULL, we just create a new node which contains the value x and set the root pointer to this node.

2) If the root is not NULL, then we perform an AVL insert operation. A new node containing x is created and is set as child of that leaf node according to the comparison result between x and the value of that leaf node.

3) Then we need to adjust the tree to satisfy both the AVL property and the median property of the new tree.

– If the tree rooted at v has odd number of nodes before insertion:

a) If the value of the new node is greater than the root value, we need to find the smallest element that is greater than v.value. Take the node out of the tree and place it to the root v. Then perform insert(v.value) to the subtree rooted at root.left.

b) Else, nothing to do with the current tree rooted at v.

– If the tree rooted at v has even number of nodes before insertion:

a) If the value of the new node is greater than the root value, nothing to do with the current tree rooted at v.

b) Else, we need to find the largest element that is smaller than v.value. Take the node out of the tree and place it to the root v. Then perform insert(v.value) to the subtree rooted at root.right.

4.3

Perform the AVL remove operation, the rest operations are the same as the insert operation. And the entire implementation is shown in the code.

4.4

We consider the worst cases, every node requires $2log2$ height (height is the height of the subtree rooted on the node). Assume n is the 2 to the power of m, $h(2n) = h(n) + 2n$. so h(2 to the power of $m + 1$)=h(2 to the power of m)+ 2 to the power of $m + 1$. By using the characteristic polynomial equation we can get $h(n) \in O$ (n—n= 2 to the power of m). So $h(n) \in O(n)$.