1. 编写函数listToTree: int list -> tree，将一个表转换成一棵平衡树。

提示：可调用split函数，split函数定义如下：

如果L非空，则存在L1, x, L2，满足：

split L = (L1, x, L2) 且

L = L1 @ x :: L2 且

length(L1)和length(L2)差值小于1。

fun split [] = ([], [])

| split[x] = ([x],[])

| split(x::y::L) =

let val(A,B) = split L

in (x::A,y::B)

end;

datatype tree = None

|Node of tree\*int\*tree;

fun listToTree [] = None

|listToTree (x::L) = let

val index = List.length L div 2

val ltree = List.take(L,index)

val rtree = List.drop(L,index)

in

Node(listToTree(ltree),x,listToTree(rtree))

end;

val l1 = [1,2,3,4,5,6,7,8,9];

val l2 = [10,11,12];

val listtotree1 = listToTree(l1);

val listtotree2 = listToTree(l2);

1. 编写函数revT: tree -> tree，对树进行反转，使trav(revT t) = reverse(trav t)。（trav为树的中序遍历函数）。假设输入参数为一棵平衡二叉树，验证程序的正确性，并分析该函数的执行性能（work和span）。

fun revT(T) = case T of None => None

|Node(ltree,x,rtree) => Node(revT(rtree),x,revT(ltree));

val l1 = [1,2,3,4,5,6,7,8,9];

val test = revT(listToTree(l1));

work = O(2n) n是树的高度

span = O(n)

3. 编写函数binarySearch: tree \* int -> bool。当输出参数1为有序树时，如果树中包含值为参数2的节点，则返回true；否则返回false。要求：程序中请使用函数Int.compare（系统提供），不要使用<, =, >。

fun binarySearch (None, x) = false

| binarySearch(Node(L, v, R), x) = case Int.compare(x, v) of

GREATER => binarySearch(R, x)

|LESS => binarySearch(L, x)

|EQUAL => true

binarySearch(listToTree([1,2,3]),3);

val it = true : bool

binarySearch(listToTree([1,2,3]),4);

val it = false : bool

4.

datatype tree = None

|Node of tree\*int\*tree;

datatype order=LESS | EQUAL | GREATER;

fun compare(x:int, y:int):order=

if x<y then LESS else

if y<x then GREATER else EQUAL;

fun treeCompare(Node(L1,v1,R1),Node(L2,v2,R2)) = compare(v1,v2)

| treeCompare(Node(L,v,R),None) = LESS

| treeCompare(None,Node(L,v,R)) = GREATER

| treeCompare(None,None) = EQUAL;

fun listToTree [] = None

|listToTree (x::L) = let

val index = List.length L div 2

val ltree = List.take(L,index)

val rtree = List.drop(L,index)

in

Node(listToTree(ltree),x,listToTree(rtree))

end;

fun SwapDown None = None

| SwapDown(Node(L,v,R)) = case treeCompare(L,R) of

LESS => (case treeCompare(L,Node(L,v,R)) of

LESS => let val Node(l1,l,r1) = L

in Node(SwapDown(Node(l1,v,r1)),l,R)

end

|GREATER => Node(L,v,R)

|EQUAL => Node(L,v,R))

|EQUAL => (case treeCompare(R,Node(L,v,R)) of

GREATER=> Node(L,v,R)

|EQUAL => let val Node(l1,r,r1) = R

in Node(L,r,SwapDown(Node(l1,v,r1)))

end

|LESS => let val Node(l1,r,r1) = R

in Node(L,r,SwapDown(Node(l1,v,r1)))

end)

|GREATER => (case treeCompare(R,Node(L,v,R)) of

GREATER=> Node(L,v,R)

|EQUAL => let val Node(l1,r,r1) = R

in Node(L,r,SwapDown(Node(l1,v,r1)))

end

|LESS => let val Node(l1,r,r1) = R

in Node(L,r,SwapDown(Node(l1,v,r1)))

end);

fun heapify None = None

| heapify(Node(L,v,R)) = SwapDown(Node(heapify(L),v,heapify(R)))

val t1 = listToTree([2,1,3])

val t2 = heapify(t1);