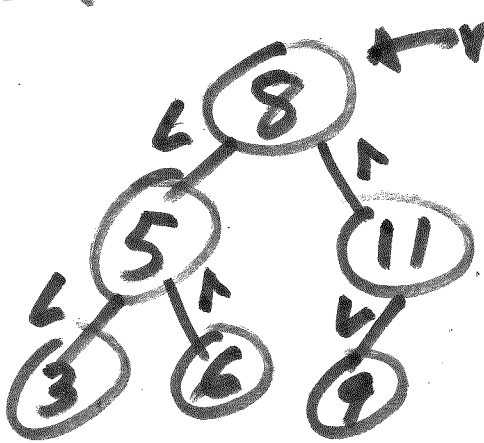
Binary Search Tree

- Each internal node stores an item (key, data)
- All the keys stored in the left subtree are less than "key"
- All the keys stored in th right subtree are greater than key



- Searching for a key will take time proportional to the distance from the root to the key.
- If we have a tree where height $\leq O(\log n)$ then we say that tree is balanced

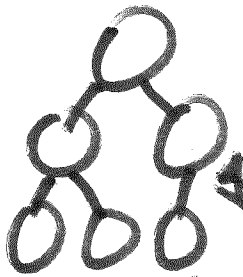
and find(key) takes O(log n).

treeSearch(6, root)

+ Operations

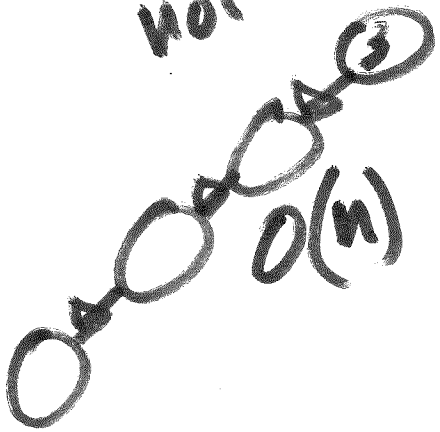
~~Insert~~

```
val = treeSearch(key, node)
    if node is null
        return null // key not found
    else if (key == node.key)
        // key is in current node return data
        return node->data
    else if (key < node->key) {
        return treeSearch(key, node->left)
    else // key is in right subtree
        return treeSearch(key, node->right)
```



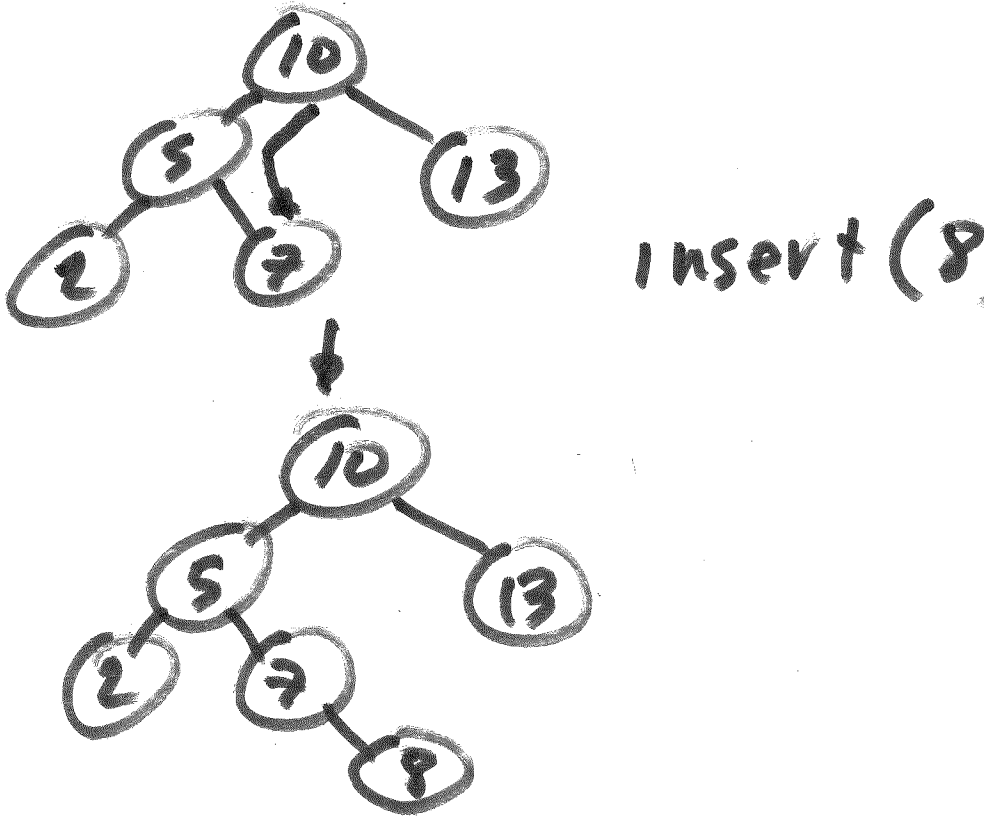
O(log n) if tree is balanced

If tree is not balanced



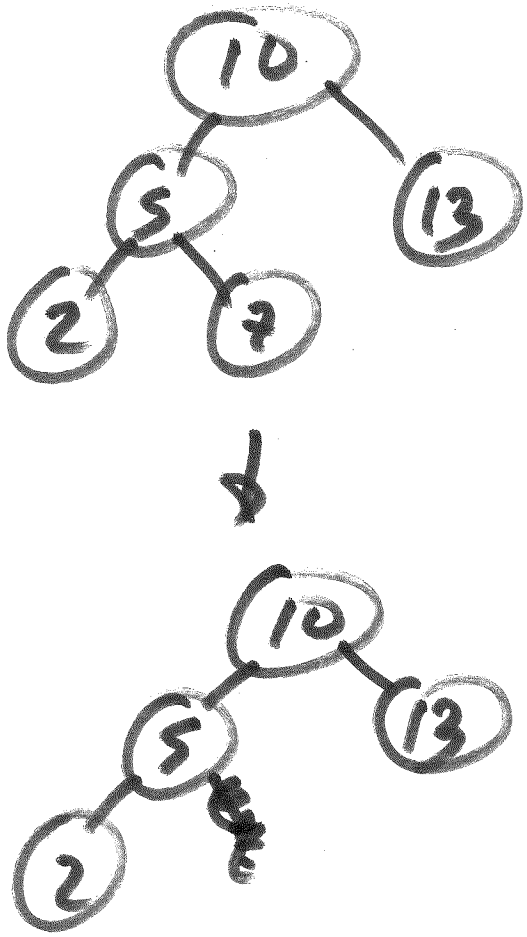
Insert (key, data)

- Search key in tree.
- If found then replace data
- If not found then add a new node at the bottom where the search ended



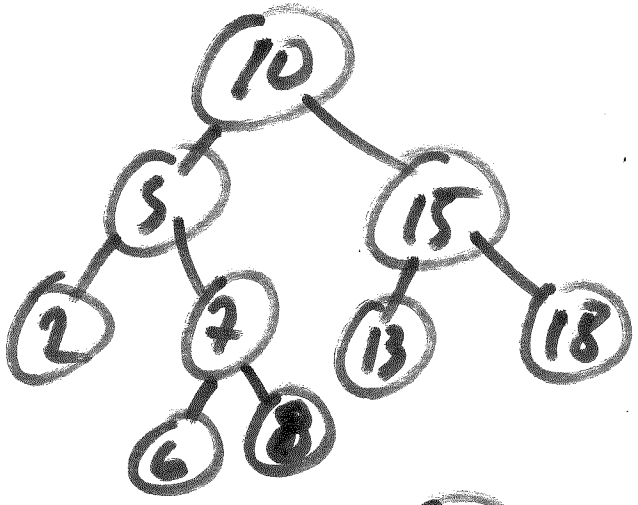
remove(key)

- Search key in tree
- If node is at the bottom just remove it.



Quiz 12

Given the tree



If we remove (5) what node will take its place

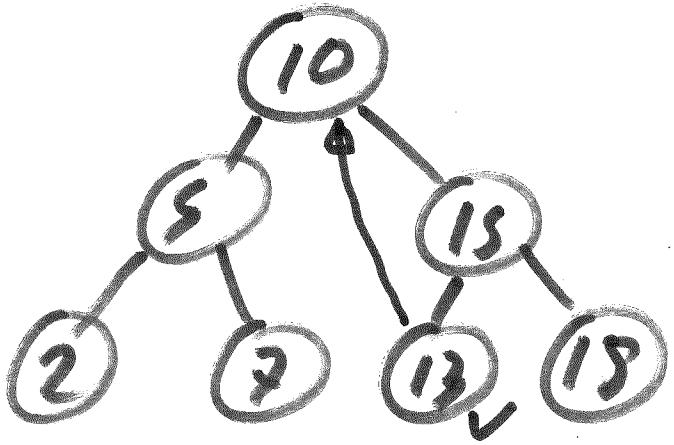
a) (2)

b) (7)

c) (6)

d) (8)

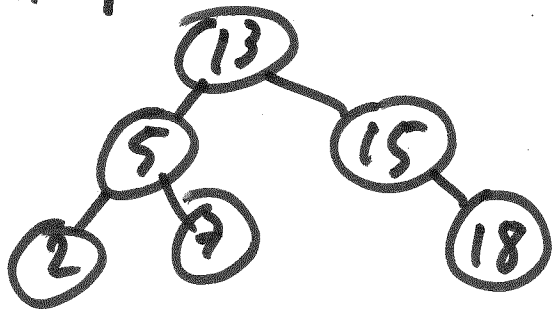
→ If node to remove is internal then



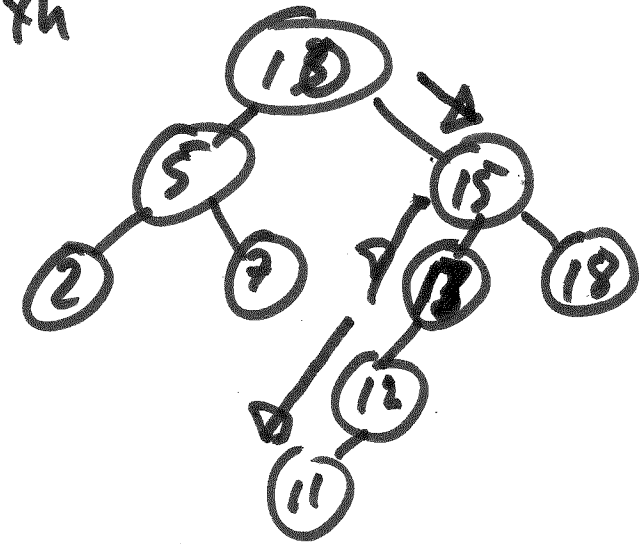
remove(10)

2, 5, 7, 10, 13, 15, 18
↳ successor

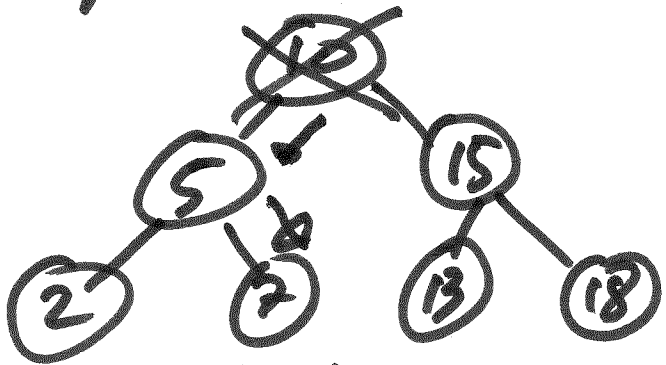
- Find the successor, that is the node that immediately follows after the node removed.
The successor of (10) in this tree is (10).
We move (13) in place of (10)



To obtain successor go to the right child of the node to remove and follow the left path

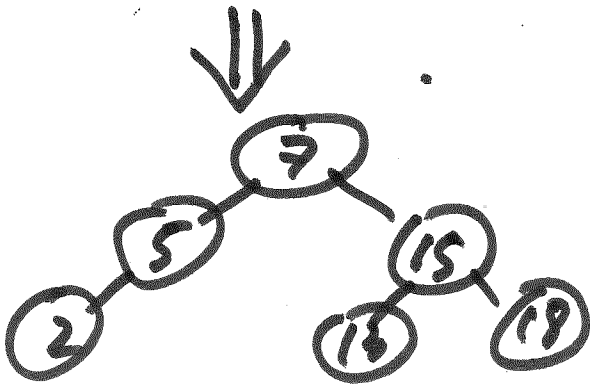


Also the predecesor could work

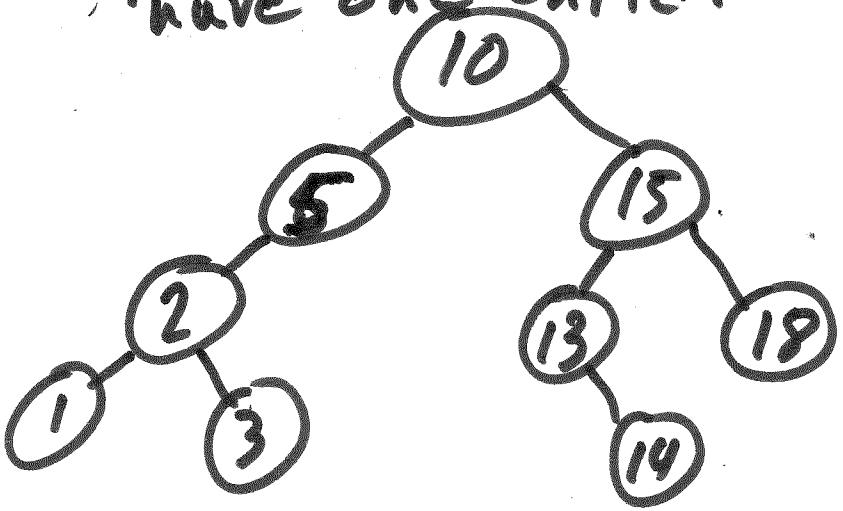


2, 5, 7, 10, 13, 15, 18

↑ predecessor



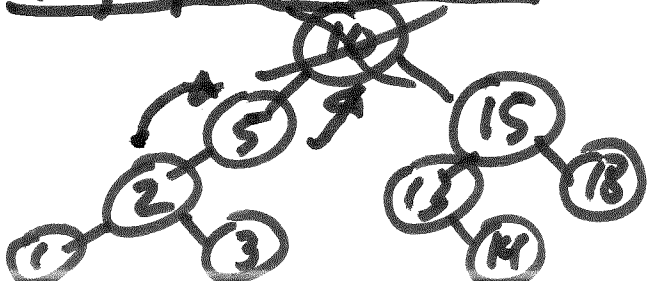
Irregular cases
predecesor or successor have one child:



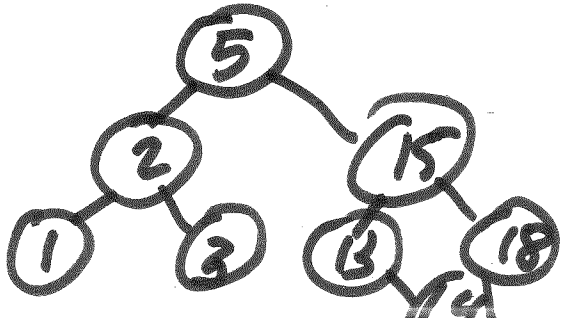
remove (10)

We still remove (10) and put the predecesor (5) or the successor (13) instead of (10) but ~~or they~~ we promote the child (should be one) instead of the predecesor or succesor to take place.

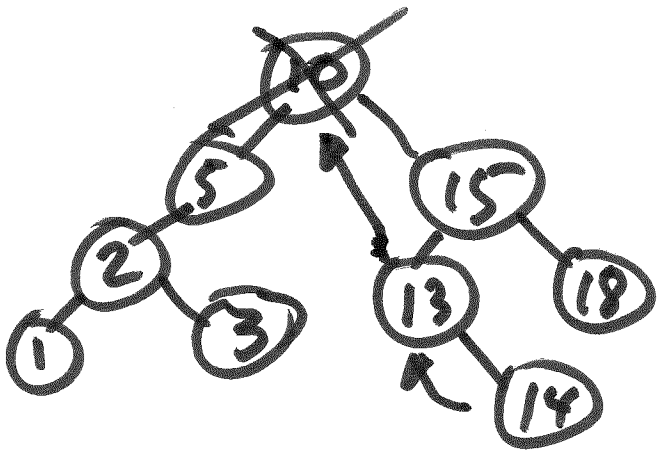
using predecesor

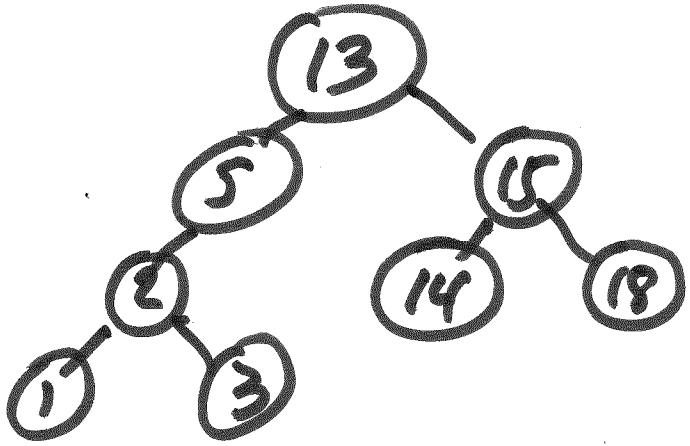


remove(10)



Using successor





The cost of removal is $O(h)$ where h is the maximum height of the tree.

If the tree is balanced then h ~~is~~ $O(\log n)$ and therefore removal is $O(\log n)$.

There is no guarantee that after a removal or insert the tree will be balanced.

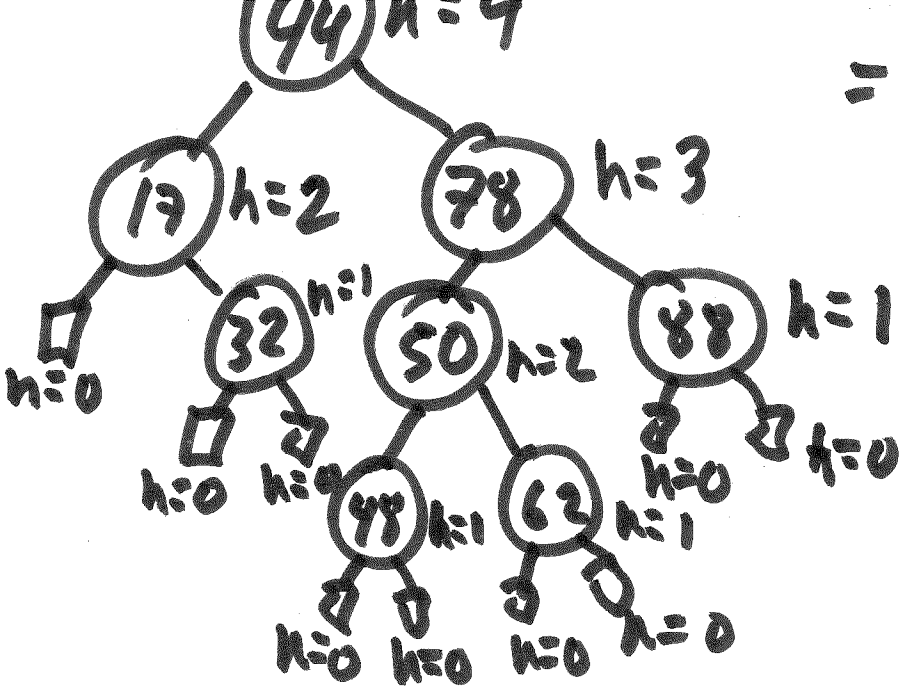
AVL trees

- They are binary trees that are balanced. That is, the height of the tree is always $O(\log n)$.
- To keep the tree balance, we rebalance (readjust) the tree after every insert or remove operation.

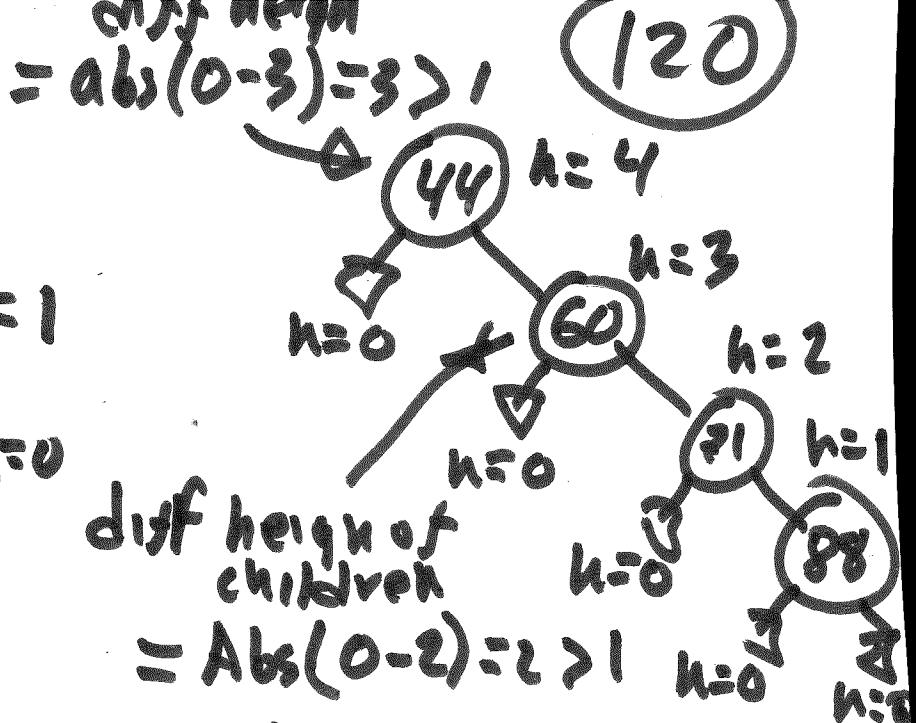
AVL Tree Property

For every internal node v of T, the heights of the children can differ at most by 1.

$$height(v) = \begin{cases} 0 & v \text{ is a leaf} \\ 1 + \max(height(left), height(right)) \end{cases}$$



AVL tree



Not an AVL tree

Claim

If we follow the AVL property then the tree's max height will be $O(\log n)$. i.e. the AVL Tree will be balanced

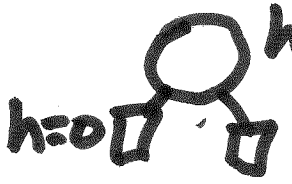
n = # of nodes.

Proof

Find the minimum number of nodes of an AVL tree of height H. (H = h)

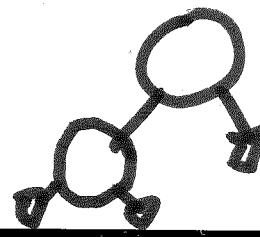
$n(H)$ = minimum number of internal nodes of an AVL tree of height H.

$n(1) = 1$

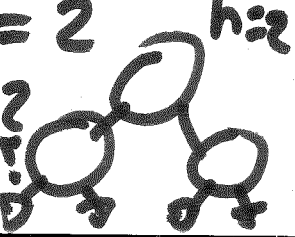


H=1
n=1

$n(2) = 2$



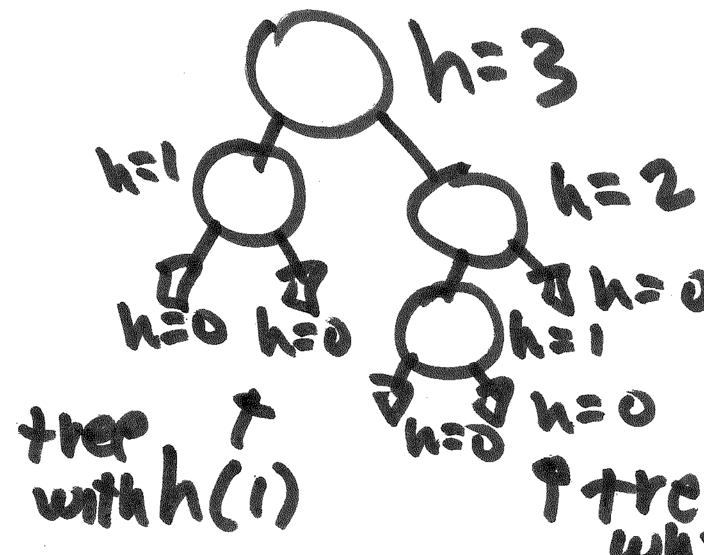
H=2



why not? Not min!

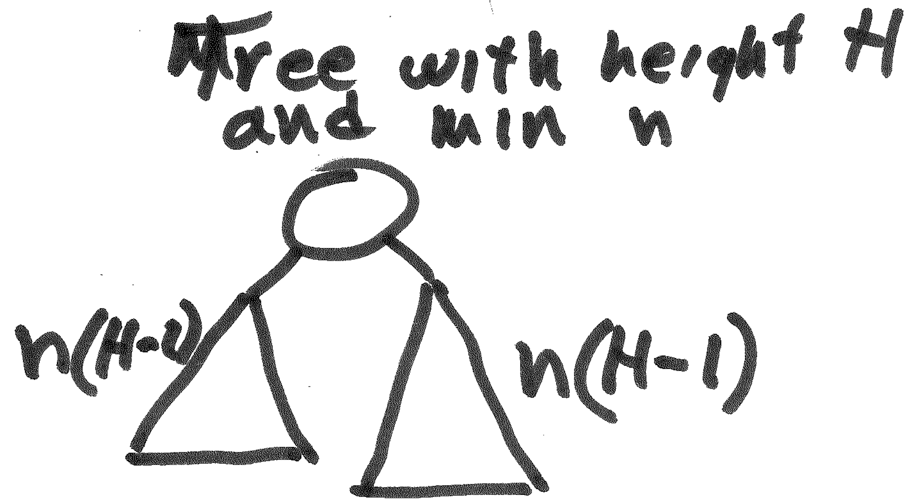
For $H \geq 3$ the AVL tree with height H and minimum number of nodes n contains a subtree of height $H-1$ and a subtree of height $H-2$

$n(3)$



In general

$n(H)$



$n(H) = 1 + n(H-2) + n(H-1)$ (123)

$n(H) > n(H-2) + n(H-1)$

since $n(H-1) > n(H-2)$

$n(H) > 2\, n(H-2)$

$n(H) > 4\, n(H-4)$

$n(H) > 8\, n(H-6)$

$\vdots \quad \vdots \quad 2i$

$n(H) > 2^i\, n(1)$

$H - 2i = 1$

$H = 1 + 2i$

$i = \frac{H-1}{2}$

$n > 2^{\frac{H-1}{2}}(1)$

$\log_2 n > \log_2 2^{\frac{H-1}{2}}$

$\log_2 n > \frac{H-1}{2} \longrightarrow 2\log_2 n + 1 > H$

$H = O(\log n)$

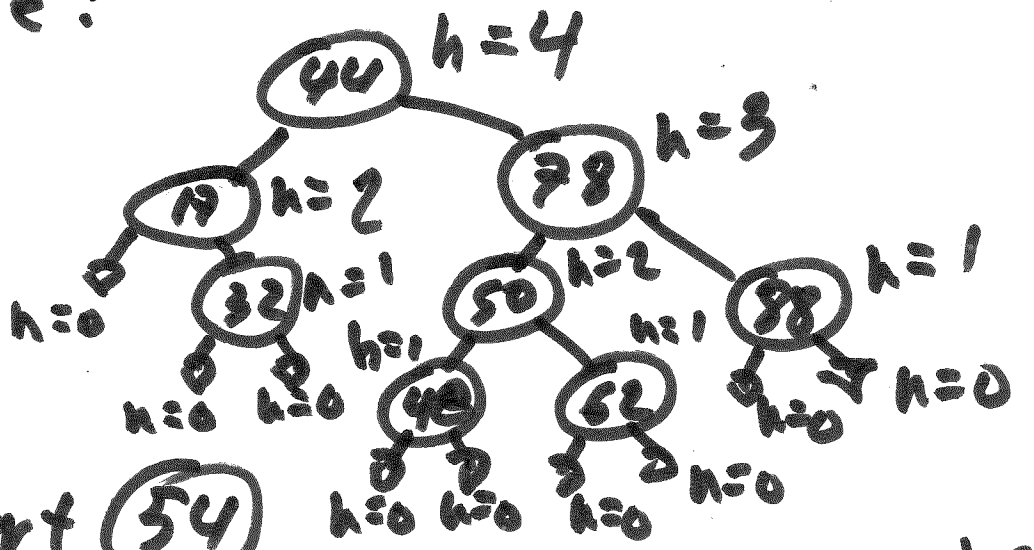
If we enforce the property of the AVL tree (difference in height of the children of a node ≤ 1) then the height is $O(\log n)$.

and therefore insert, remove, find are also $O(\log n)$.

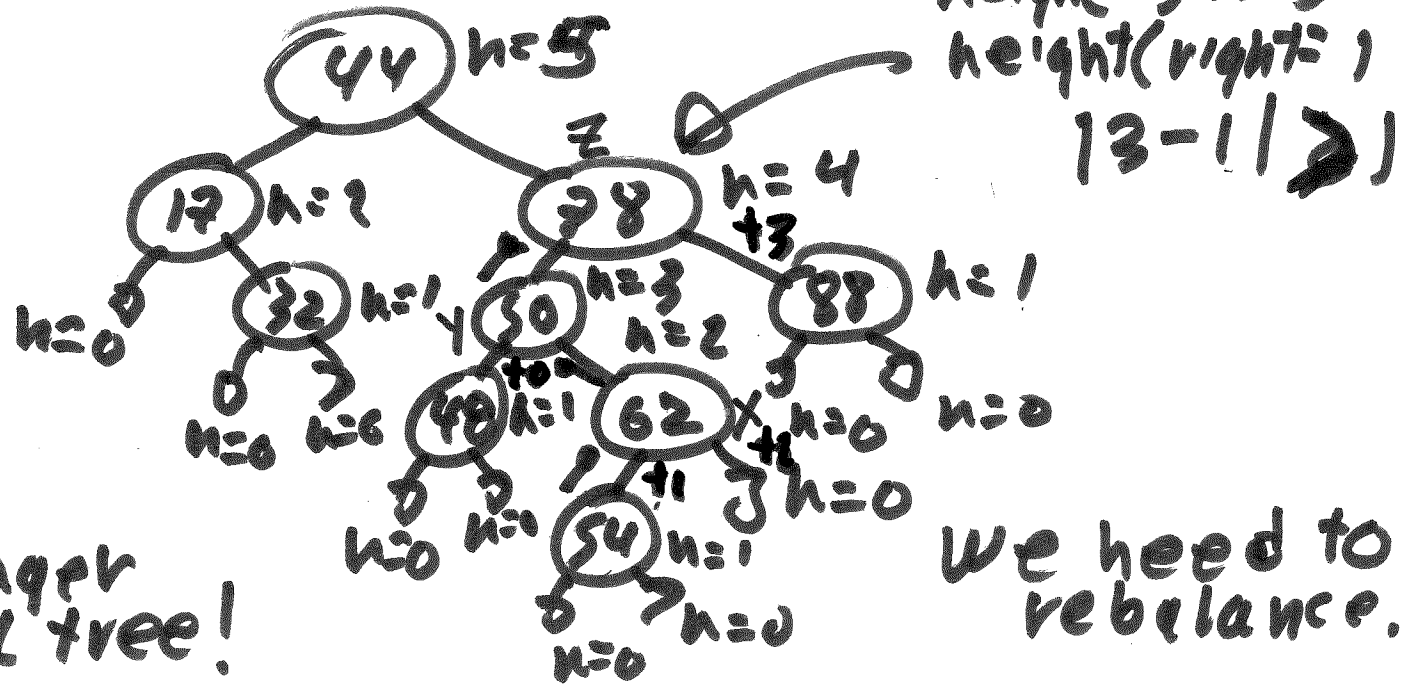
→ At every insert, remove we rebalance tree to make sure property is followed. rebalancing also has to be at most $O(\log n)$ to keep operations $O(\log n)$.

Insertion in an AVL tree.

Insert item like we usually insert in a binary tree:



Insert (54)

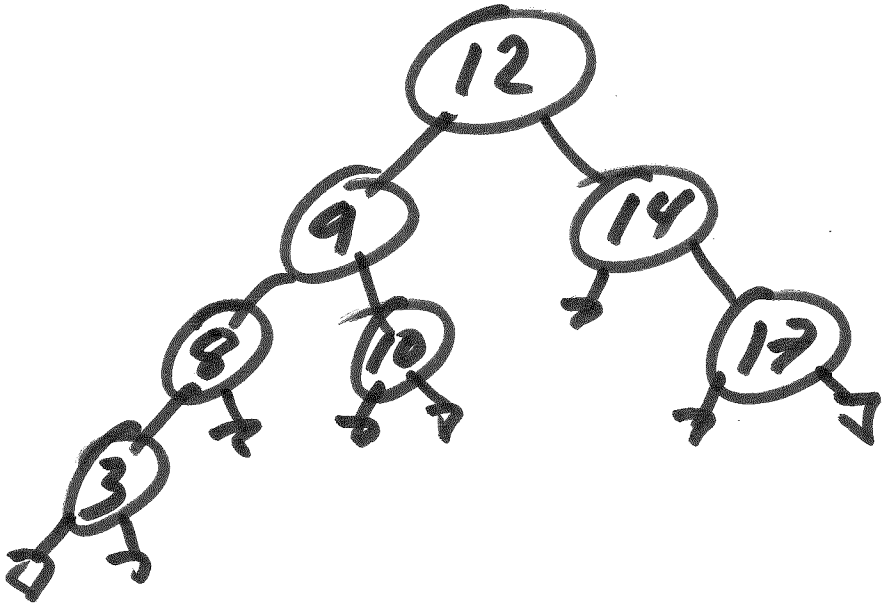


→ No longer an AVL tree!

We need to rebalance.

Quiz 13 AVL

What is the height of the root node in this tree (



a) 5

b) 4

c) 3

d) 2

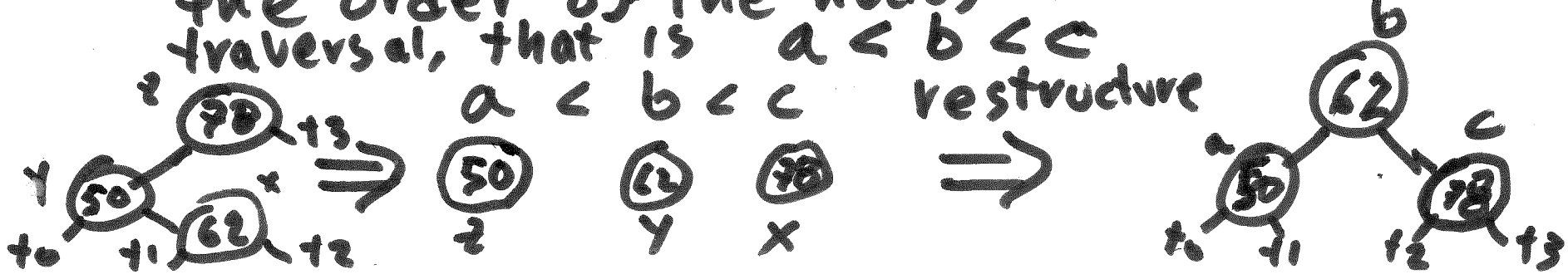
→ To rebalance tree, from inserted node go up until we find an ancestor that is root of the unbalanced tree, then we do the following labeling

z → Node that is the root of an unbalanced subtree. i.e. the difference in the height of the left subtree and right subtree is larger than 1

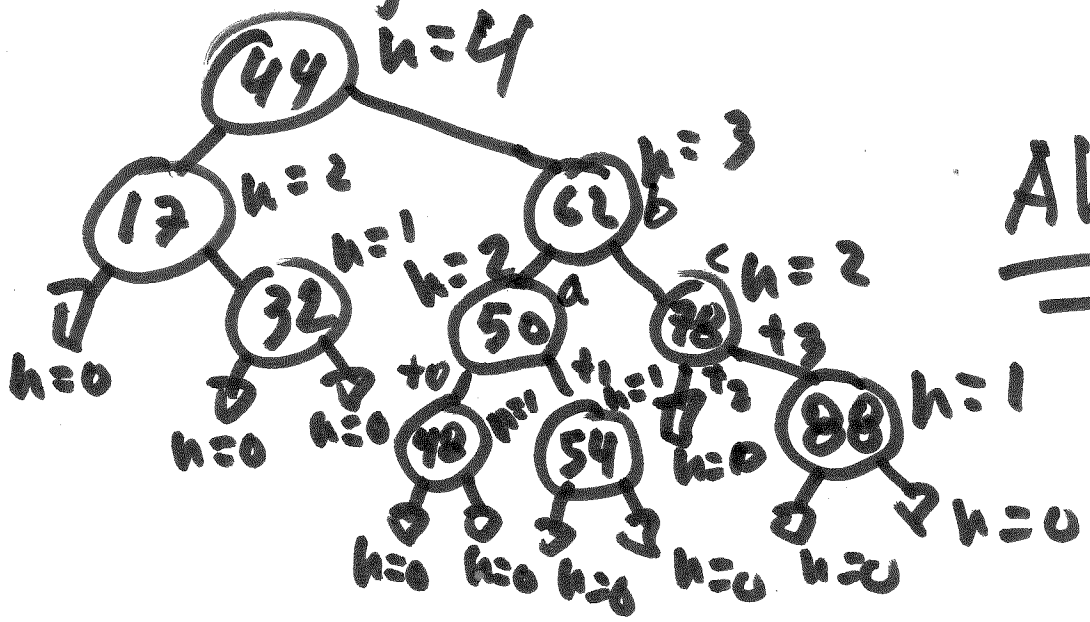
y → the child of z with largest height.

x → the child of y with largest height

→ Rename z, y, x as a, b, c based on the order of the nodes in an in-order traversal, that is a < b < c



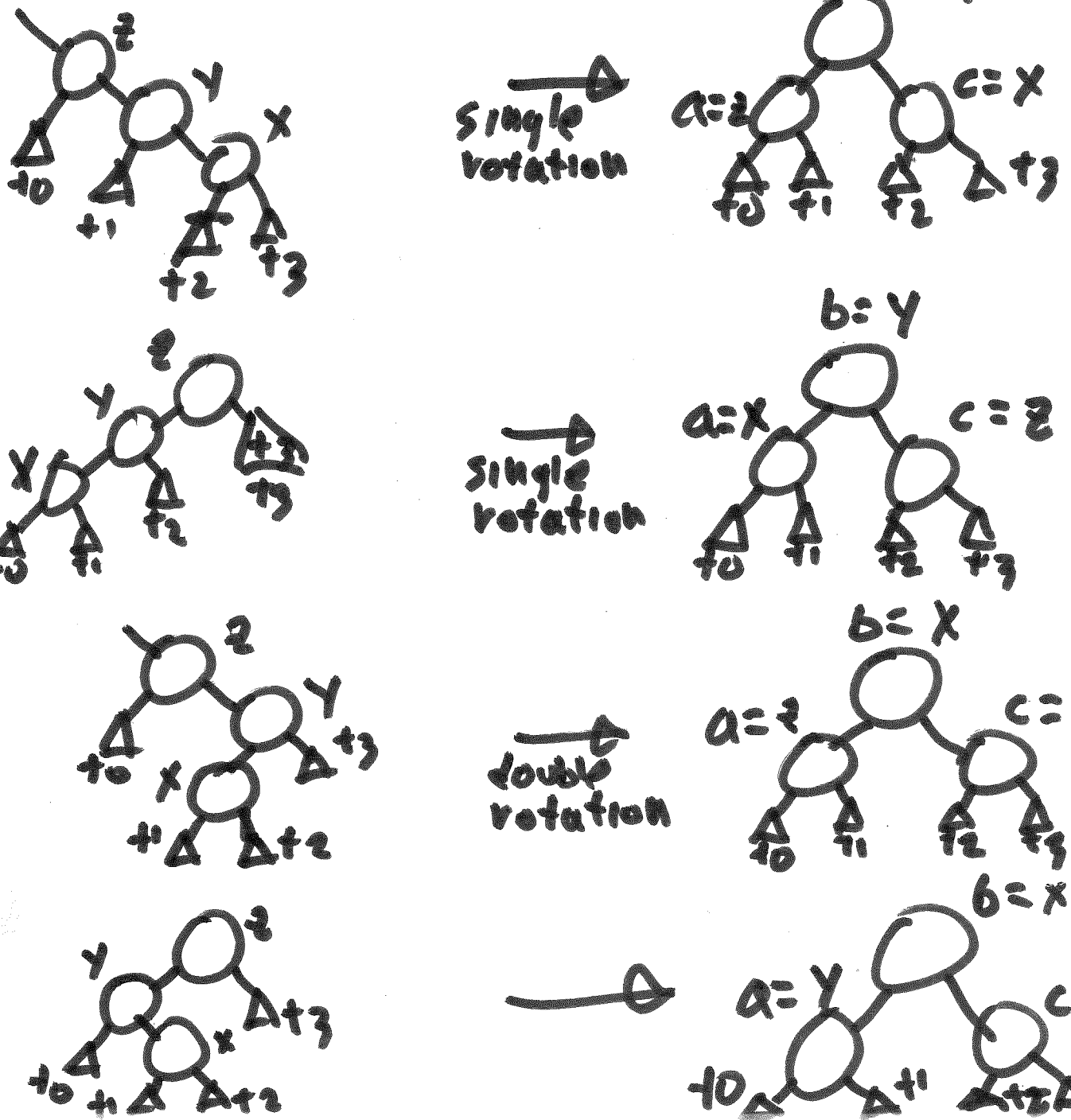
Rearranging a, b, c in the tree we get:



AVL tree!

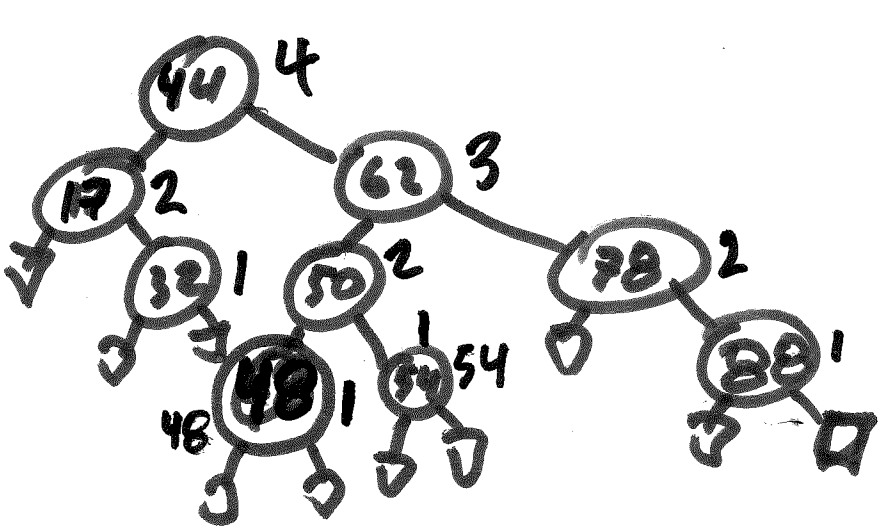
Restructuring has 4 cases

acbcc

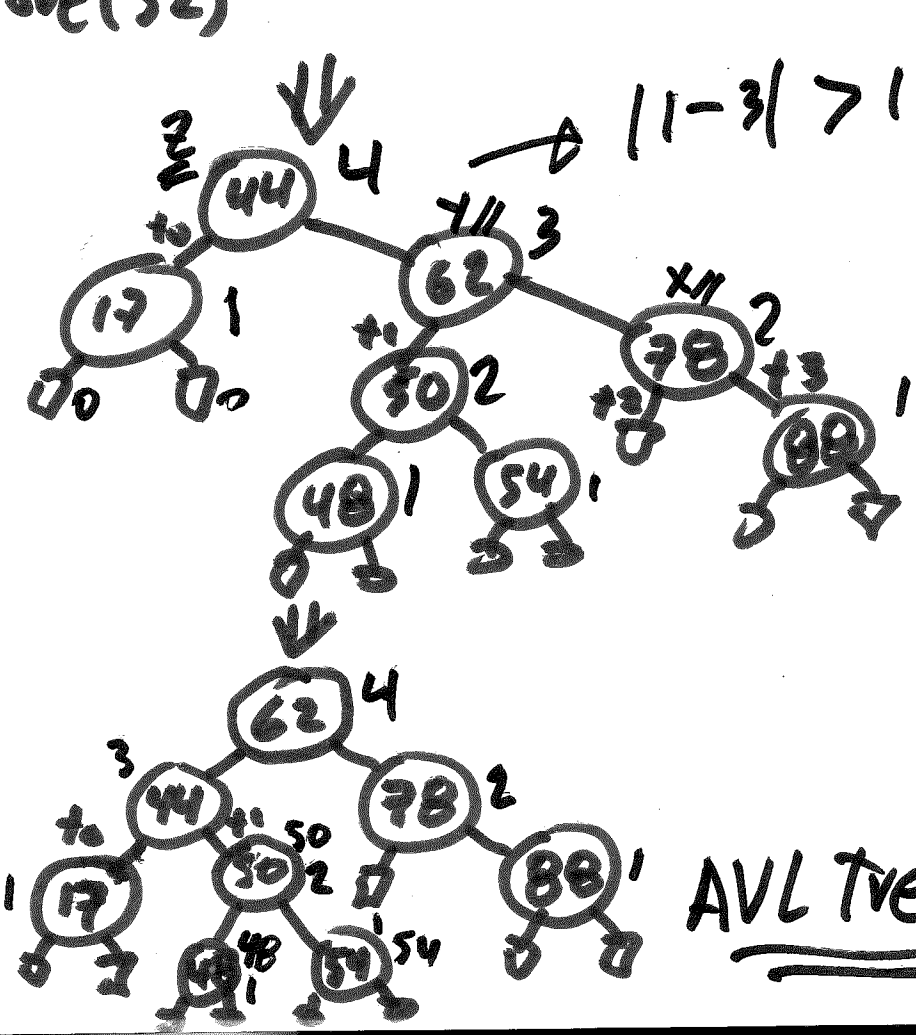


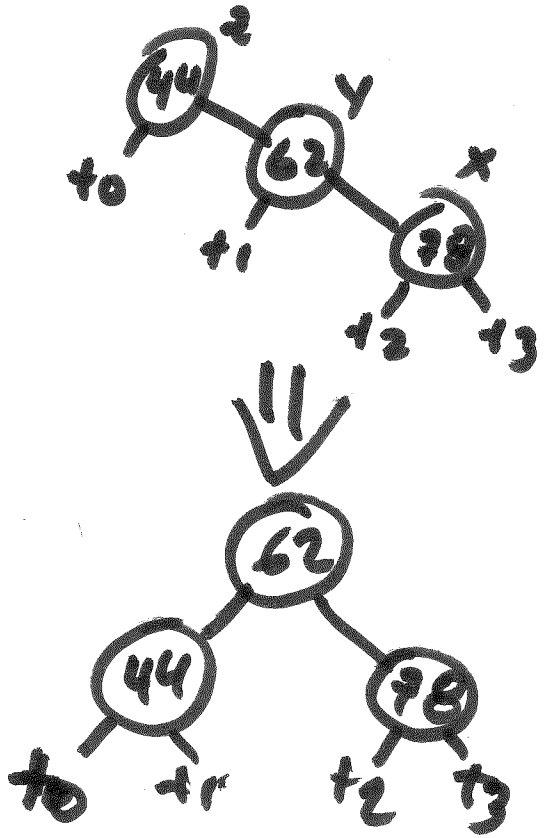
Removal

- Removing a node can cause a tree to become unbalanced
- Assuming that the node removed is w, we find the first unbalanced subtree by going up in the tree.
- Similar to insertion, we find z, y, x
  - z - Parent of unbalanced subtree
  - y - child of z with largest height
  - x - child of y with largest height. (if both children have same height, then use either one as x).
- Apply same steps of restructuring used in insertion.
- As this restructuring may upset the balance of another node higher in the tree, we continue trying to restructure until it is not needed or root is reached.



remove(32)

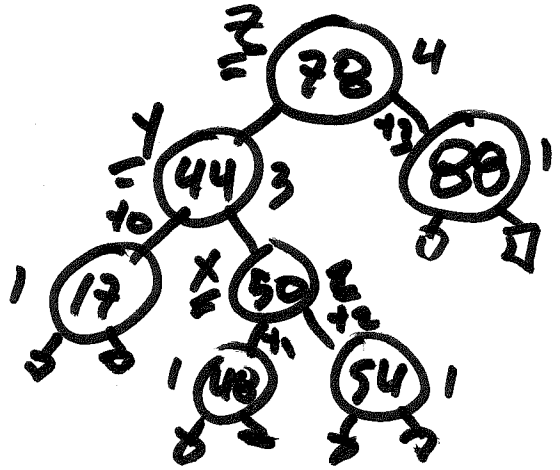




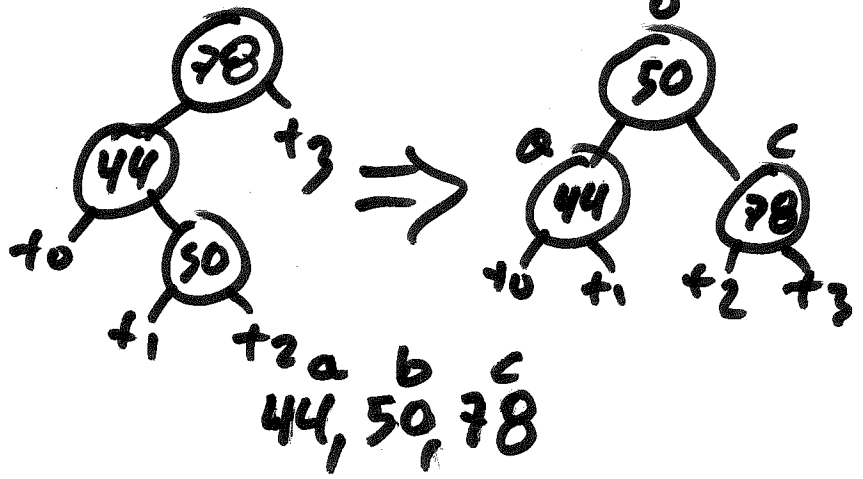
AVL Tree!

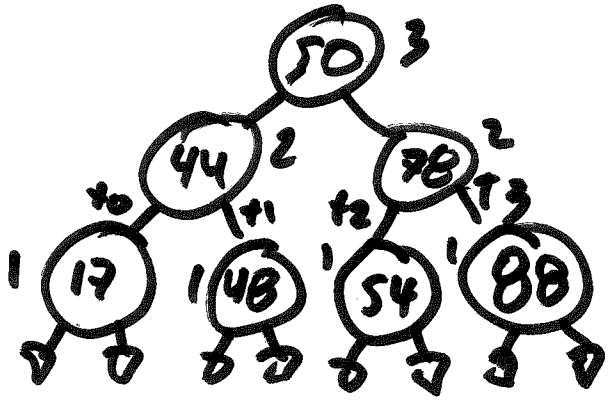
remove(62)

Remove (62) and it leaves a hole.
We put in its place either the predecessor (54)
or the successor (78). We will choose (78)



Not balanced

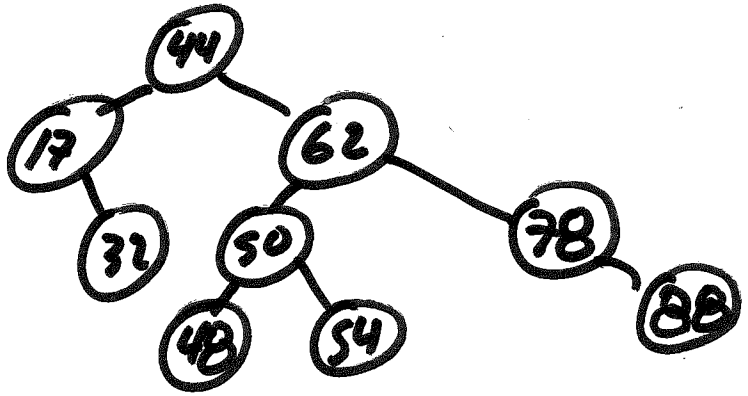




balanced!

Quiz 14

Starting at tree



If we remove (17) what are the values of z, y, x

a) z=62, y=50, x=54

b) z=62, ~~x=78~~, x=88

c) z=44, y=62, x=78

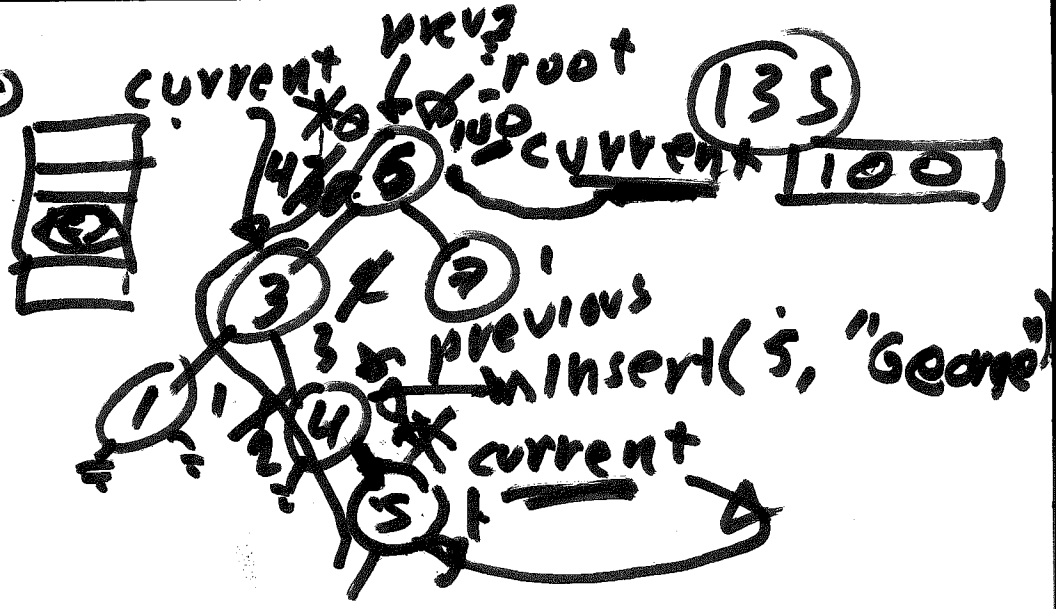
d) z=44, y=62, x=54

AVL Tree Implementation

AVL[redacted].h

```
struct AVLNode {
    int _height;
KeyType [redacted] _key;   // Type of key is int.
                           // The Keys can be other types like
DataType [redacted] _data; // const char * etc.
                           // Also data can be of other type
    AVLNode *_left;  //different than int.
    AVLNode *_right;
    AVLNode *_parent;
};

class AVLTree {
    AVLNode *_root;
    void restructure(AVLNode *n);
public:
    AVLTree();
    bool insert(KeyType key, DataType data);
    bool remove(KeyType key);
    bool find(KeyType key, DataType &data);
};
```

AVL.cpp




```
#include "AVL.h"

AVLTree::AVLTree() {
    _root = NULL;
}

bool
AVLTree::insert(KeyType key, DataType data) {
    // Find node to insert into
    AVLNode * current = _root;
    AVLNode * previous = NULL;
    while (current != NULL) { // Iterate until we
                              // get to the bottom
                              // or find key.
        previous = current;
        if (key < current->key) {
            current = current->left;
        } else if (key > current->key) {
            current = current->right;
        } else { // key found!
```

```
            // change data
            current->data = data;
            return true; // true if key found.
        } //else key found
    } //while
    //key was not found. Create new node.
    ALNode *n = new AVLNode();
    n->key = key;
    n->data = data;
    n->left = NULL;
    n->right = NULL;
    n->parent = NULL;
    n->height = 1;
    //Test if tree is empty
    if (previous == NULL) { // _root is NULL
        //insert n as root node
        _root = n;
        return false
    }
```

```
// insert at the left or right of previous.
if(key <= previous->key) {
    //insert at the left
    previous->left=n;
}
else {
    //insert at the right
    previous->right=n;
}
n->parent= previous;
//Adjust height of nodes in path
//from inserted node to root.
AVLNode *m = n->parent;
while(m != NULL) { //iterate until we
                   //reach root.
    int maxheight = 0;
    if (m->left!=NULL) { //checks left child
      maxheight = m->left->height;
    } if (m->right!=NULL && m->right->height > maxheight)
      maxheight = m->right->height;
}
```

O(log n)

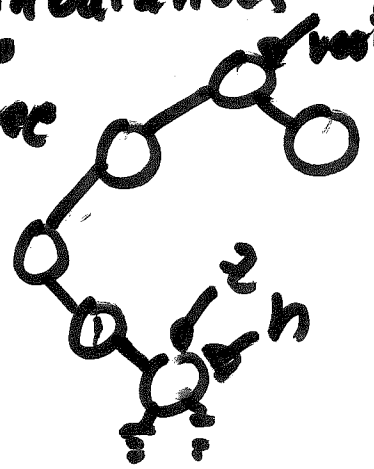
```
    // compute height of m
    m->height = maxheight + 1;
    m = m->parent; // go to parent
} // while
restructure(n); // restructure tree starting
                // at inserted node n.
return false;
}
```

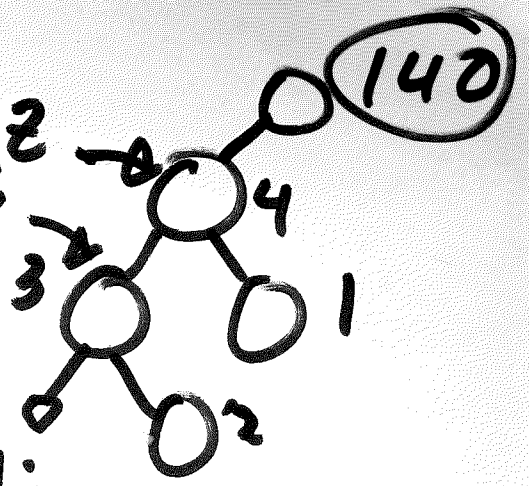
Quiz 15

In the following code that finds the node to insert

```
current=root; previous=NULL;
while (current!=NULL){
        if(key < current->key)
            current =current->left;
        }
        else if (key > current->key){
            current= current->right;
        }
        else{
            current->data=data;
        }
}
//what is the missing line?
a) current= previous;
b) current= previous->next;
c) previous = current;
d) previous=current->left;
```

```
void
AVLTree::restructure(AVLNode *n){
  //Rebalance AVLtree starting at node n
  AVLNode *z = n->parent;   // search node that is
  while (z != NULL){        // root of unbalanced
                            // subtree
    // Find the parent of unbalanced subtree
    int hleft = 0;
    if (z->left != NULL){
      hleft = z->_left->_height;
    }
    int hright = 0;
    if (z->right != NULL){
      hright = z->_right->_height;
    }
    int hdiff = hright - hleft;
    if (hdiff < 0) hdiff = -hdiff; // obtain abs
                                   // value
    if (hdiff <= 1){
      // node is balanced. Go up to find one
      z = z->parent;               // that is unbalanced.
      continue;
    }
//Recompute
//height of z
//in case height
//of children
//was modified
if(hleft > hright)
  z->height=
    hleft + 1;
}
else {
  z->height=
    hright + 1;
}
```

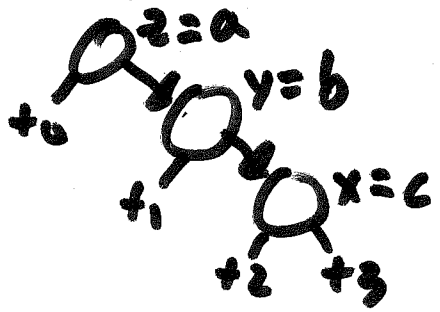




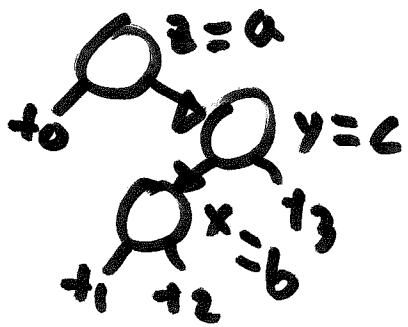

```
// z is unbalanced
// Find y. Child of z with max height
AVLNode *y = NULL;
int maxh = 0;
if (z->left != NULL) {
    y = z->left;
    maxh = z->left->height;
}
if (z->right != NULL && maxh < z->right->height)
{
    y = z->right;
}
assert(y != NULL); // should never happen
                   // but just in case
                   // we have a bug.

//Find x
AVLNode *x = NULL;
maxh = 0;
if (y->left != NULL) {
    x = y->left;
    maxh = x->height;
}
if (y->right != NULL && y->right->height > maxh){
    x = y->right;
}assert(x != NULL);
```

4 cases:

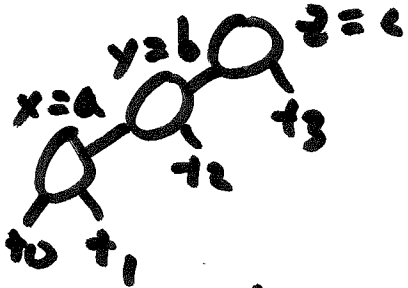
Case 1:



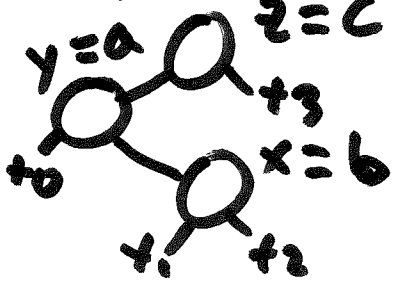
Case 2

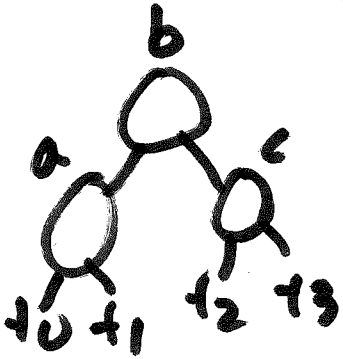


Case 3

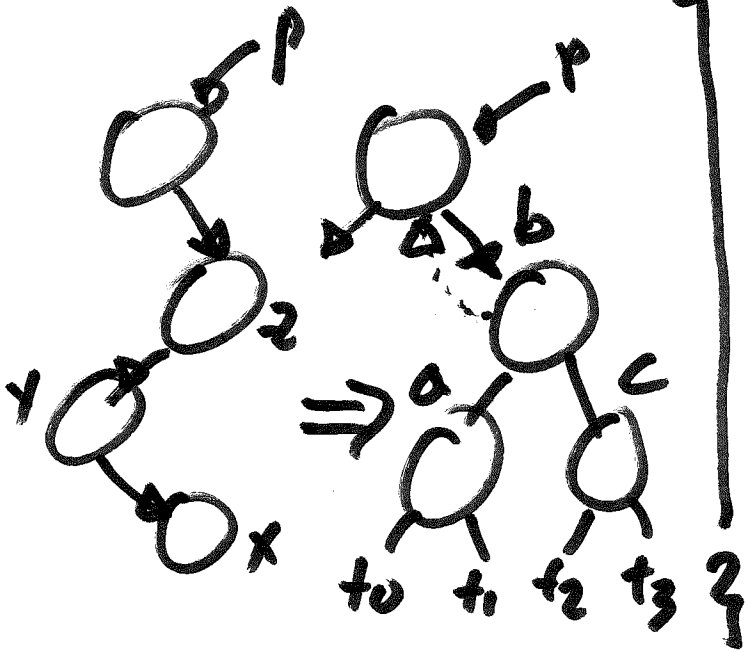


Case 4

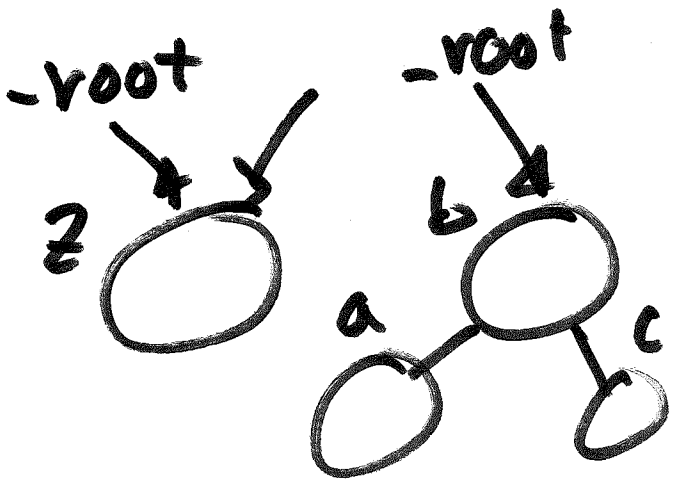





```
//Identify a, b, c
AVLNode *a, *b, *c, *t0, *t1, *t2, *t3;
if (z->right == y){
    //case 1 or 2
    if(y->right == x){
        //case 1
        a = z; b = y; c = x;
        t0 = z->left; t1 = y->left;
        t2 = x->left; t3 = x->right;
    }
    else {
        //case 2
        a = z; c = y; b = x;
        t0 = z->left; t1 = x->left;
        t2 = x->right; t3 = y->right;
    }
}
else {
    // case 3 or 4
    if (y->left == x) {
        //case 3
        a = x; b = y; c = z;
        t0 = x->left; t1 = x->right;
```



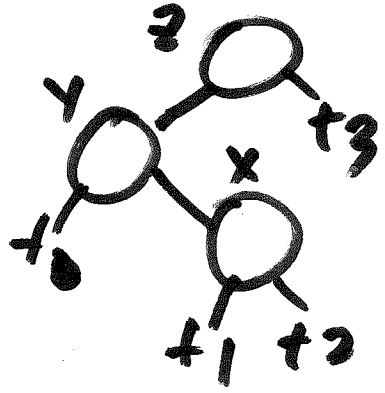

```
| t2=y->right; t3=z->right;
} // case 3
else {
  | // Case 4
    a=y; b=x; c=z;
    t0=y->left; t1=x->left;
    t2=x->right; t3=z->right;
}
```




```
// Do rotation
AVLNode *p=z->parent;
if(p!=NULL){
  | if(p->left==z){
    | // attach b to left of p
      p->left=b;
  | }
  | else{
    | // attach b to right of p
      p->right=b;
  | }
}else{ // z is root
    _root=b;
}
```

Quiz 16

Assuming Case 4 in restructure



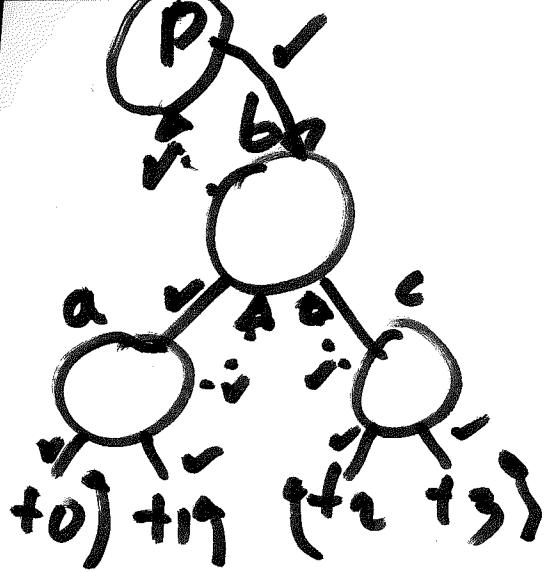
the code to identify the nodes ~~which are~~ ~~identify~~ t2 and t3 is which one

a) t2 = x -> left; t3 = x -> right

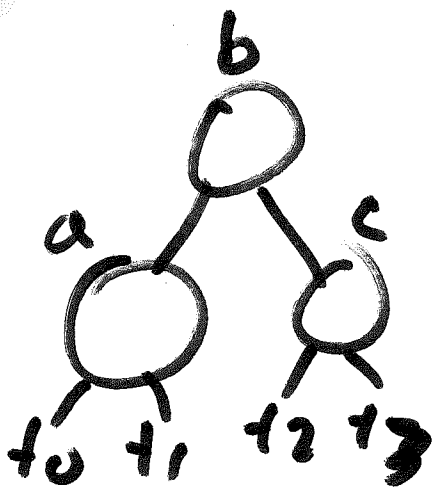
b) t2 = z -> right; t3 = y -> right

c) t2 = x -> right; t3 = z -> right

d) t2 = y -> right; t3 = z -> right




```
b->_parent = p;
b->_left = a;
b->_right = c;
a->_parent = b;
a->_left = t0;
a->_right = t1;
c->_parent = b
c->_left = t2
c->_right = t3
//Connect parent of t0, t1, t2, t3
if (t0 != NULL) {
    t0->_parent = a;
}
if (t1 != NULL) {
    t1->parent = a;
}
if (t2 != NULL) {
    t2->_parent = c;
}
if (t3 != NULL) {
    t3->_parent = c;
}
```



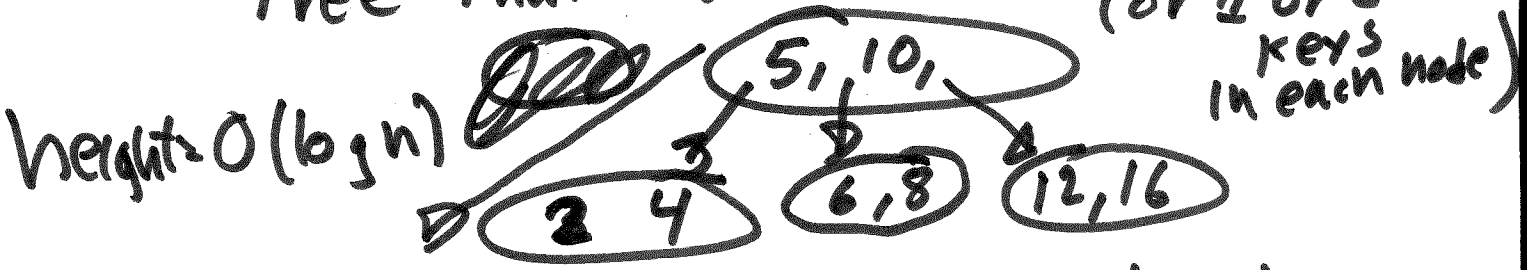

```
//Fix height of a
int maxheight = 0
if (a->left != Null){
   maxheight = a->left->_height;
}
if (a->_right && a->_right->height > maxheight){
   maxheight = a->right->height;
}
a->height = maxheight + 1;

//Fix height of c
maxheight = 0;
if (c->left != Null){
   maxheight = c->left->height;
}
if (c->right != Null && c->right->height > maxheight){
   maxheight = right->height;
}
c->height = maxheight + 1;
```

```
//Fix height of b
maxheight=0
if (b->left!=NULL){
   maxheight=b->left->height;
}
if (b->right!=NULL && b->right->height>maxheight)
{ maxheight=b->right->height
 }
 b->height= maxheight+1;
 z=p; //Go up.
}//while z!=NULL.
}//restructure.
```

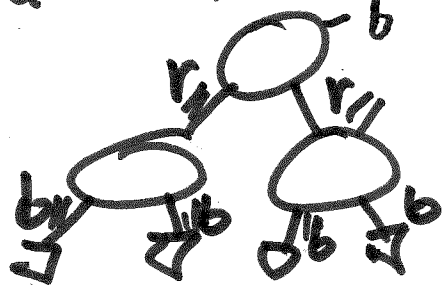
The AVL remove is similar to insert
- find node.
- remove node
- restructure

Other types of balanced trees

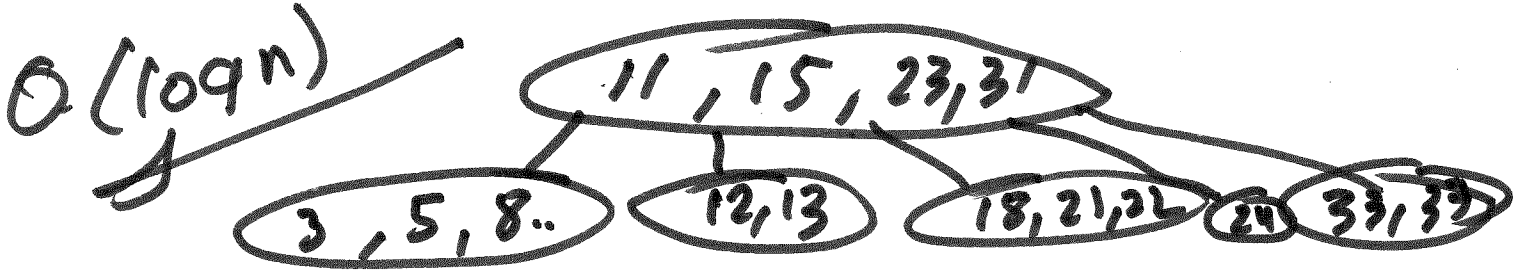
- AVL Trees
- 2-3 tree
  Tree that has 2 or 3 children (or 1 or 2 keys in each node)



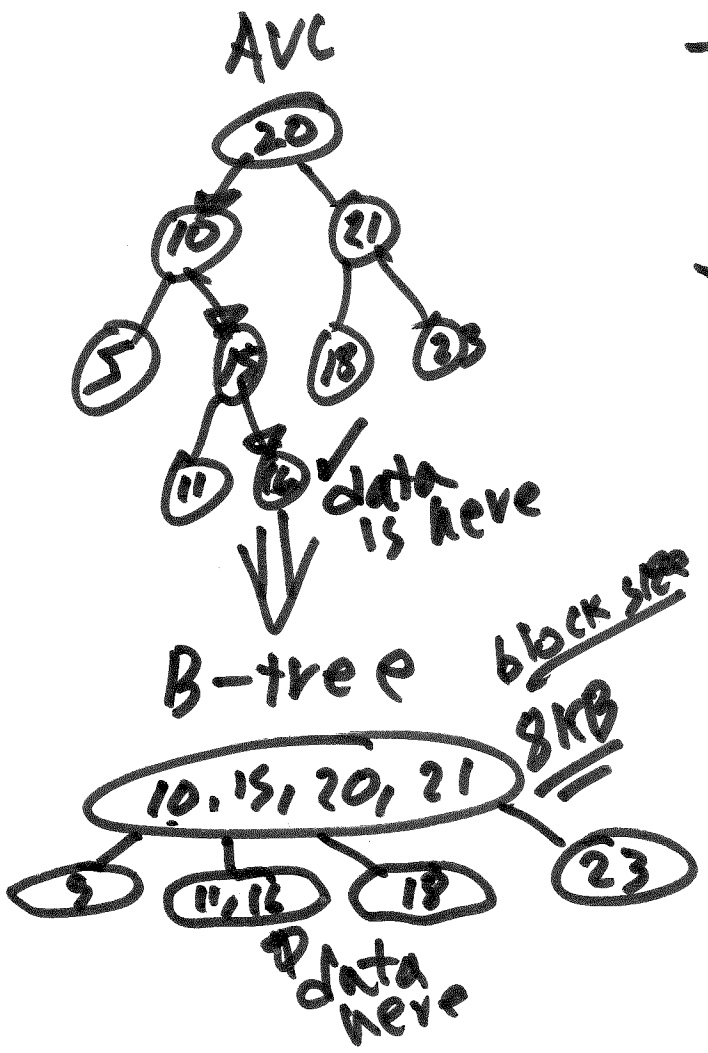
- Red-black tree. Each edge has a color red or black



- b-trees
  - They may have many keys in each node



- The more keys are in each node, the height will be smaller.
- B-trees are important in databases.
- Database are stored in disks.
- In a disk the read and write operations are in entire blocks, where each block is 1KB or 4KB, 8KB 16KB.
  - Even if you only want to read 1 byte, a whole block (E.g 8KB) has to be read.
- A database access will take as many seconds as the number of block reads in disk.



- In AVL tree shown, access to bottom of the tree will take 4 disk accesses.
- In B-tree shown, access to the bottom will take only 2 disk accesses
  - Assume each node is in a different block.
  - The maximum number of keys in a AVL node will be limited to the number of keys that you can fit in a block
  - B-trees are used to store indexes in databases.
  - B-trees minimize the number of disk accesses

MergeSort

- It is a sorting mechanism that uses divide an conquer

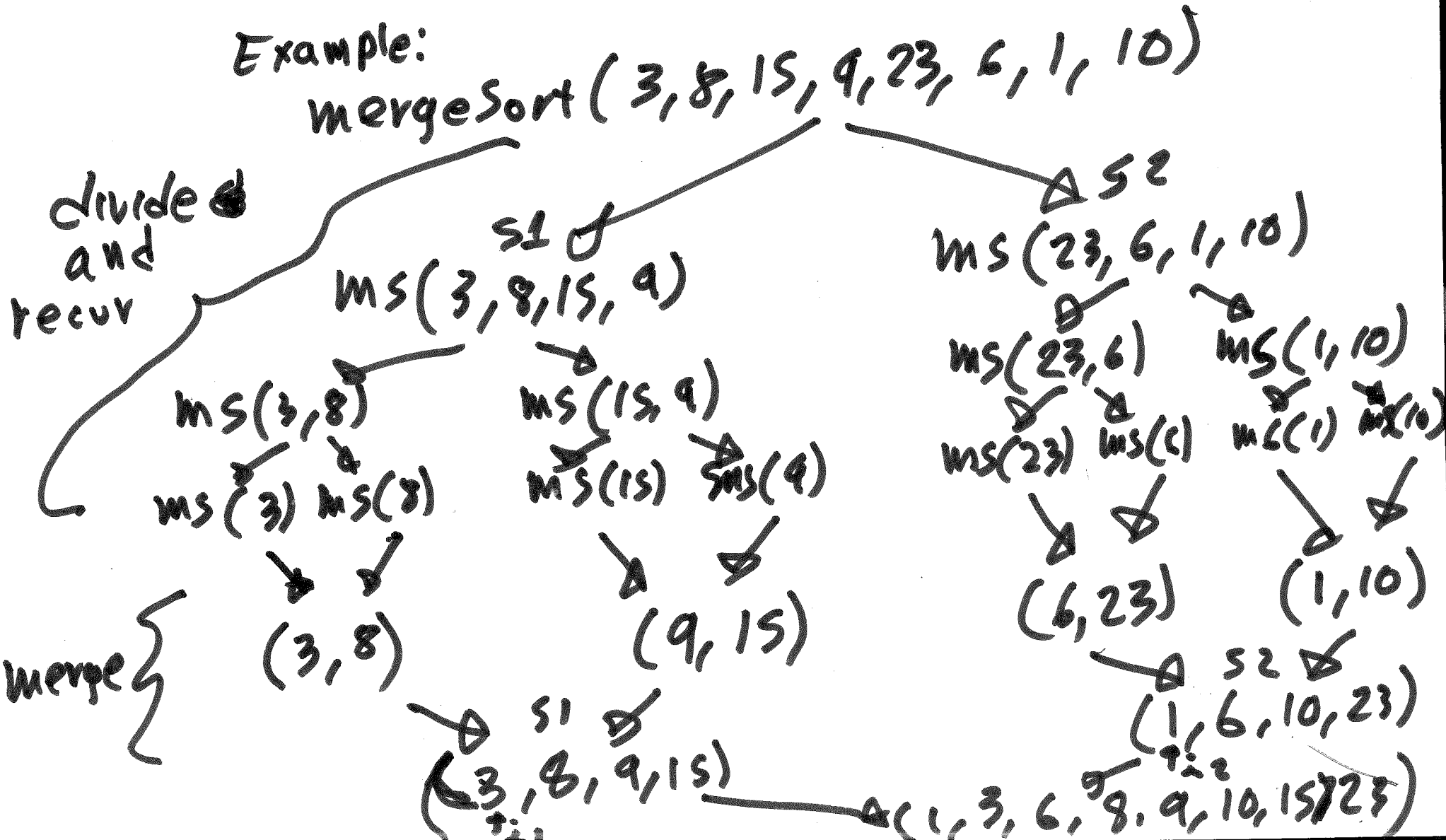
Divide and conquer

* three steps
  - Divide - divide data into two or more disjoint sets
  - Recur - solve the problem in the smaller subsets recursively.
  - Conquer - Take solutions of subproblems and merge them to create a solution of the original problem.

Merge sort uses <u>divide and conquer</u>

Algorithm mergesort(s)
Input: array s
Output: sorted array s

Example:



Quiz 17

What is wrong in the following code that recomputes the height of a node n

```
n->height= 1+max(n->left->height,
                 n->right->height);
```

a) The code is correct

b) It should be just max(n->left->height, n->right->height) no "+1" needed.

c) It has to check if n is root

d) left or right may be null.

Time complexity of Mergesort.

We use "amortized time", that is we look at the work done for one element, and then we multiply the work by n.

- Every element (e.g. 23) is copied log(n) + log(n) (23 is copied 3+3 for n=8).
- You can see this from the drawing above since it looks like a tree and an inverted tree. The height of the tree is log(n).
- Work for one element is O(logn)

  Work for n elements is nO(logn)

  so the worst case takes O(nlogn).

Implementation

```
void mergeSort(int *s, int n){
  // sorts array s of type int
  // with n elements
  if (n <= 1) {
      return; // End condition
  }
  // Divide
  // create s1 and s2
  int n1 = n/2;
  int n2 = n - n1;
  int *s1 = new int[n1];
  int *s2 = new int[n2];
  // Put elements into s1, s2
  int i;
  for(i = 0; i < n1; i++){
    s1[i] = s[i];
  }
```

```
for (i=0; i<n2; i++){
    s2[i] = s[i+n1];
}

//call mergesort recursively
mergesort(s1, n1);
mergesort(s2, n2);
// s1 and s2 are sorted
//merge them back into s
int i1=0; //index for s1
int i2=0; // index for s2
i=0;      // index for s

while (i1<n1 && i2<n2) {
    if(s1[i1] < s2[i2]) {
        s[i] = s1[i1]; // i1 has s1 is smallest.
        i1++;
    }else{
        s[i] = s2[i2]; i2++;
    }
```

s1 (3, 8, 9, 15) — $n_1 = 4$

s2 (1, 6, 10, 23) — $n_2 = 4$

s (1, 3, 6, 8, 9, 10, 15,)

```
    i++;
  } //while

  //Copy remaining elements
  //Copy remaining elemts in S1, if any
  for(; i1<n1; i1++, i++){
     S[i] = S1[i1];
  }
  //Copy remaining elements in S2, if any
  for(; i2<n2; i2++, i++){
     S[i] = S2[i2];
  }
 //S is sorted!!
  // Delete temporal arrays S1, S2
  delete [] S1;
  delete [] S2;
} // End of mergeSort.
```

Notes on Mergesort

- MergeSort is not an "in-place" algorithm, that is we cannot use the same array S to do the sorting. We need to allocate extra memory.
- However, we could reduce the number of times memory is allocated with "new" using a scratch array that is allocated at the beginning and deleted at the end.

Mergesort takes $O(n \log n)$ worst case

QuickSort

- Popular sorting algorithm that also uses divide-and-conquer.

Algorithm quicksort(S)
input: unsorted array S
output: sorted array S

1. Divide
   - If S has only one element return
   - Otherwise choose an element x in S called the "pivot".
   - Divide S into three sets: L, E, G such that the elements in L are smaller than x, the elements in E are equal to x, and the elements in G are larger than x.

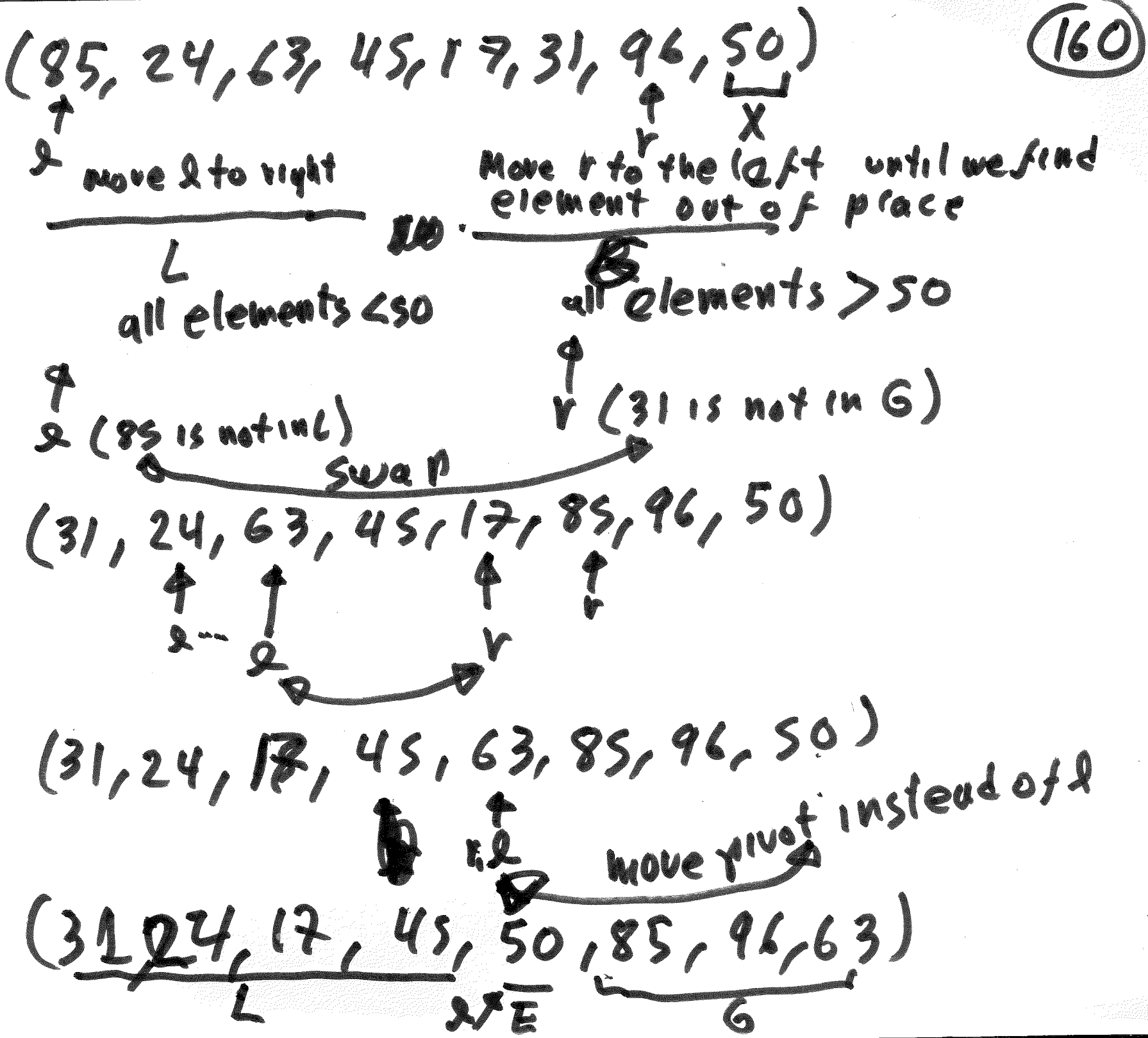
$\forall y \in L, y < x$
$\forall y \in E, y == x$
$\forall y \in G, y > x$

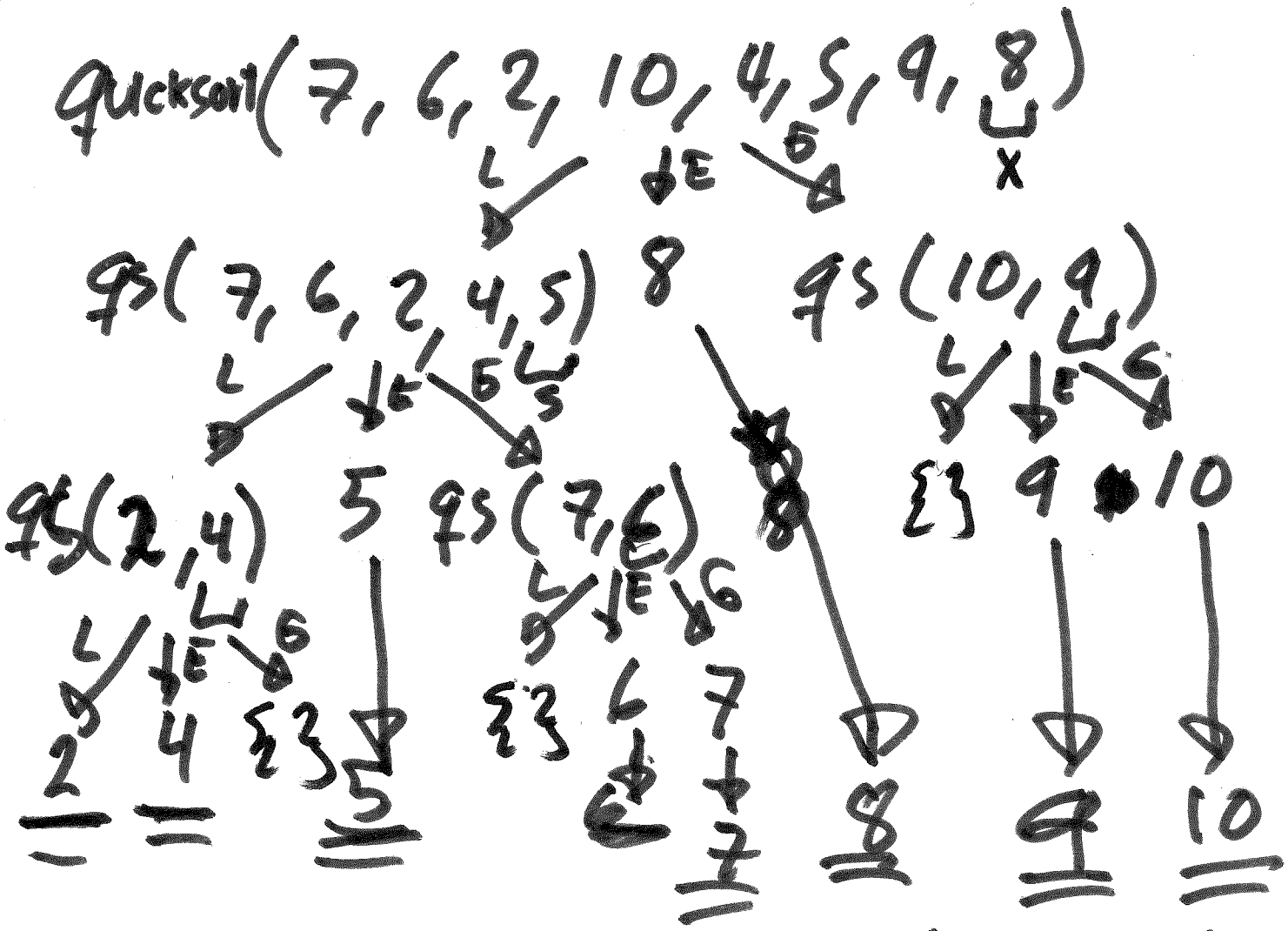
2. Recur
   quicksort(L), E, quicksort(G)

3. Conquer
   put elements in L, E, G back into S.

In-place quicksort

- We use the same array S for storage of L, E, G at each step.
- We use subranges to represent each set.
- The last element is used as pivot x.
- To merge L and G we will scan L left to right ~~and to~~ and G right to left to look for element not in place and swap them.
- This is how we create L, E, G in place.


(85, 24, 63, 45, 17, 31, 96, 50)
l
move l to right
r
Move r to the left until we find element out of place
X
L
all elements < 50
G
all elements > 50
l (85 is not in L)
r (31 is not in G)
swap
(31, 24, 63, 45, 17, 85, 96, 50)
l
r
(31, 24, 17, 45, 63, 85, 96, 50)
move pivot instead of l
(31 24 17, 45, 50, 85, 96, 63)
L
E
G



In average if we choose a random pivot, L and G will have more less the same size. That means that every time we split S into L, E, G we will have half the elements

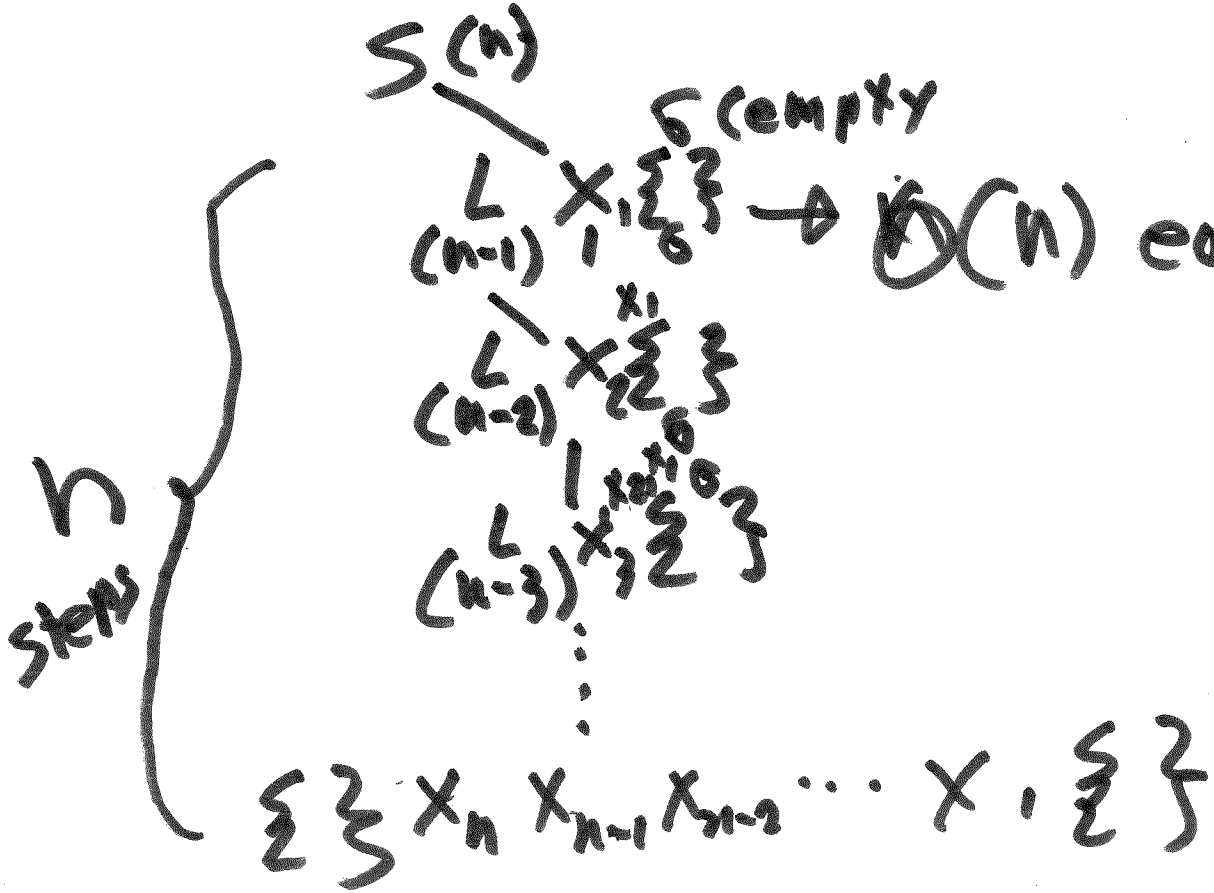
n → n/2 → n/4 → ... → n/n — work for 1 element

n/2 n/4 n/4 ... — work for 1 element in average is $O(\log n)$.

For n element in
average total work is

n O(log n). → average case O(n log n).

However in the worst case
if the pivot is either the largest or the
smallest then L or G will be empty.



n O(n)
steps form L, G
total worst case
$O(n^2)$.

In practice quicksort works well. The $O(n^2)$ time usually will not happen.

Quiz 18

What algorithm has a worst-case time of $O(n^2)$

a) heapsort

b) mergesort

c) quicksort

Implementation

In place quicksort. No need to allocate memory. All sorting is done in the array itself.

```
// Sort an array of int's using quicksort
void quicksort(int *s, int n){
    quicksortSubrange(s, 0, n-1);
}
//sorts a subrange of integers using
// quicksort
void quicksortSubrange(int* s, int left, int right)
{
    //base case. Return if s subrange has no
    element>
    if(left >= right){
        return;
    }
```

Improvement

```
int i = rand(left,right);
int tmp = s[right];
s[right] = s[i];
s[i] = tmp;
```

```
// Use the last element as pivot
int x = s[right];
// Divide s into L, E, G
int l = left; // left index
int r = right-1; // right index
while (l < r) { // loop until l and r cross
    // Move l to the right until
    // we find a number that is larger
    // than pivot x since it is not in L.
    while (l < r && s[l] < x) {
        l++;
    }
    // Move r to the left until
    // we find a number that is smaller
    // than pivot x since it is not in G
    while (l < r && s[r] >= x) {
        r--;
    }
}
```

```
    // If still l < r and l and r
    // are indexes of elements that
    // should not be in L and G then
    // swap them
    if (l < r) {
      //swap s[l] and s[r]
      int tmp = s[l];
      s[l] = s[r];
      s[r] = tmp;
    }
  } // while
//swap now pivot with l
int tmp = s[l];
s[l] = x;
s[right] = tmp;
```

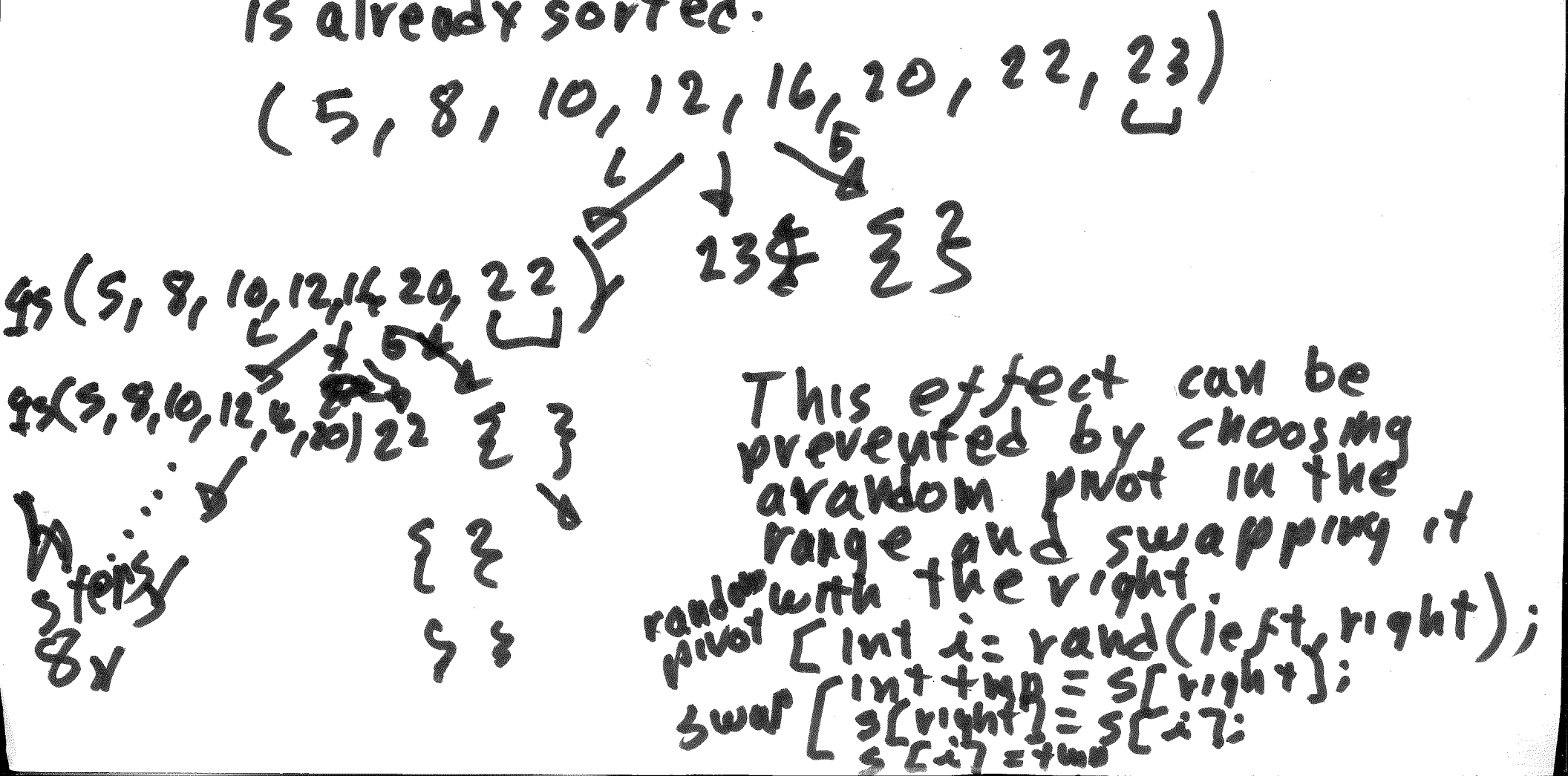
```
// L and G have been formed.
// Recur on L and G
quickSortSubrange(S, left, L-1);
quickSortSubrange(S, L+1, right);
} // End quicksortSubrange
```

In average, the pivot will be chosen in such a way that L and G will have approximately the same size. If this is the case and looking at the execution tree, each number will be copied at most log(n) times. Therefore the work done for n elements will be n * log(n) or O(n log(n))

average case

However, in the worst case if we choose the pivot x every time so L or G are empty then quicksort will degenerate and tak $O(n^2)$.

For example in the algorithm above we try to sort a sequence that is already sorted.



Quiz 19

If at every step of quicksort the pivot is chosen to be the minimum in the subrange, the algorithm will take

a) $O(n \log n)$

b) $O(n^2)$

c) $O(n^3)$

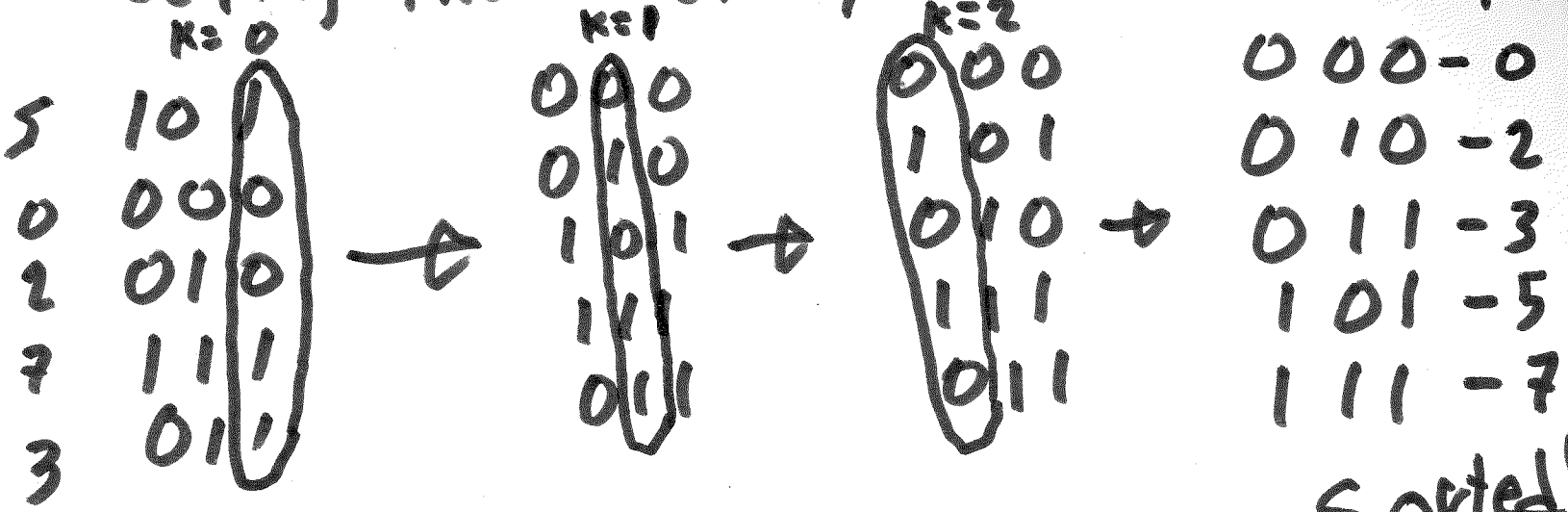
d) $O(1)$

Radix Sort

- Considers the structure of the keys.
- It assumes that the keys are represented in a M number system.
- Algorithm Radix Sort
  1. Assuming binary representation of the key, examine bits from right to left

5-	101
0-	000
2-	010
7-	111
3-	011

2. At every step k, sort the array by looking only at bit k and keeping the order of the keys unchanged.



Sorted!

Complexity:

b - # of bits used to represent a key

n - Number of entries sorted

Radix Sort takes = b steps

* O(n) → work at every step

= O(bn)

Implementation

```
// Sorts an array of unsigned ints
// using radix sort
void radixSort(unsigned *array, int n){
  //allocate scratch memory
  unsigned int *array2 = new int[n];
  assert(array2 != NULL);
  int nbits = sizeof(unsigned) * 8;
  //for all bit positions
  for(i=0; i<nbits; i++){
    int j=0; //index of array2
    //sort 0's first
    for(int k=0; k<n; k++){
      //check if column i (bit) of array[k] is 0
      if((array[k] & (1 << i))== 0)
                    mask with 1 at bit i
        //array[k] has 0 at bit i
        array2[j] = array[k];
        j++;
```

i: 001 ← 0, 010 ← 1, 100 ← 2

i ↓

a[k]=101

& 001 ← (1

001 ≠ 0

a[k] 010

& 001 == 0 ← skip

```
i
↓
0 1 1 0
2 0 1 0 0
---------
  0 1 0 0
```

```
for (int i=0; i<2; i++){
    // Sort 1's next
    // check if bit i of array[k] is 1
    for (int k=0; k<n; k++){
        if ((array[k] & (1<<i)) != 0 ){
            // array(k) has 1 at bit i
            array2[j] = array[k];
            j++;
        }
    }
    // Copy array2 to array
    memcpy (array, array2, n*sizeof(unsigned
} // for (i=0 ...
delete [] array2;
}
```

Bucket Sort

- It is a sorting algorithm useful to sort large sequence of numbers that have a small range of values. This means that the numbers will be repeated.

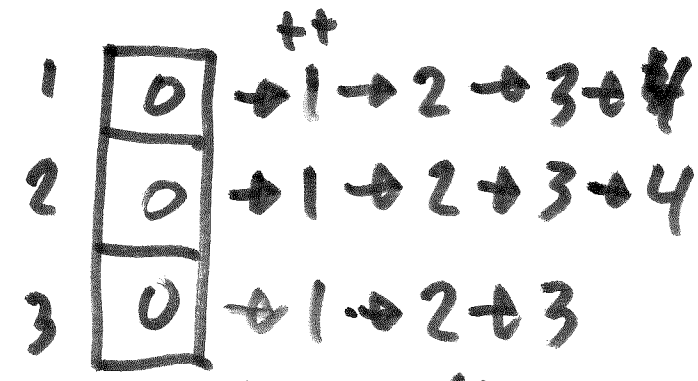
  Eg. n=1,000,000 numbers

  range of values is 1..10

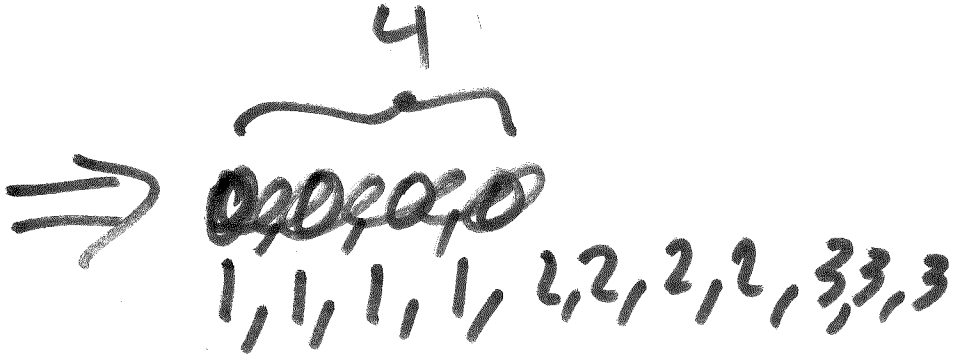
2, 1, 3, 1, 2, 2, 3, 1, 1, 3, 2

n=11

m= 1..3 = 3 (size of range)



array of counters (buckets)



Complexity of bucket sort

$O(n+m)$. But since $n >> m$ then

$\approx O(n)$

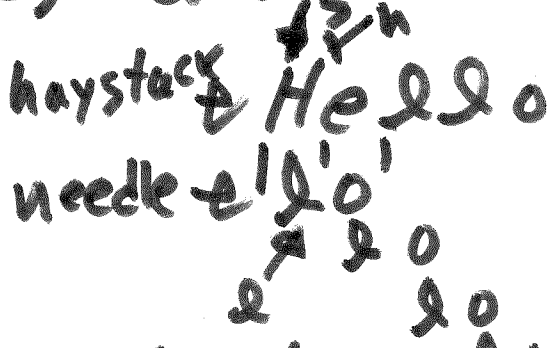
Excercise:
write ~~a~~ implementation for

```
bucketSort(int *a, int n){
    //implement it. You may comput
      min and max in
      O(n) time.
}
```

String matching

Find the position of a substring in a text.

Brute Force




```
char * strstr(char *haystack,
              char* needle){
  // Returns pointer to first occurrence
  // of "needle" in "haystack" or NULL if
  // it is not there.
  char *s=haystack;
  while(*s!='\0'){
    char *h=s;
    char *n=needle;
    while (*h!='\0' && *n==*h){ //compare
      h++; n++;                 // needle
    }                           // and s
    if (*n=='\0'){ // Found match.
      return s; // return position.
    }
    s++;
```

```
    }//while (*s != '\0')
    // Match not found
    return NULL;
}//End strstr
```

Complexity: N – Length of string (haystack)
M – Length of substring (needle)

In the worst case we do N steps ~~for~~ searching a substring that is M in length

$O(MN)$

Example of worst case.

haystack l l l l l l l l l l l l l l l
needle l l l l l h

Forces to do all iterations

Quiz 20

Can radix sort be used for strings.

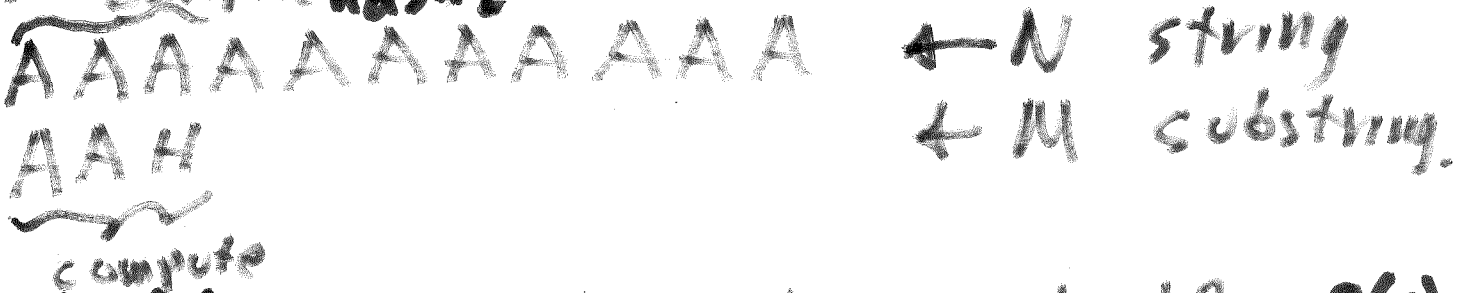
a) Yes

b) No

c) Yes. only if strings are made of '0' and '1' ascii characters.

Rabin-Karp String matching

It uses a hash function that is applied to the M characters of the substring and the string matched



← N string
← M substring.

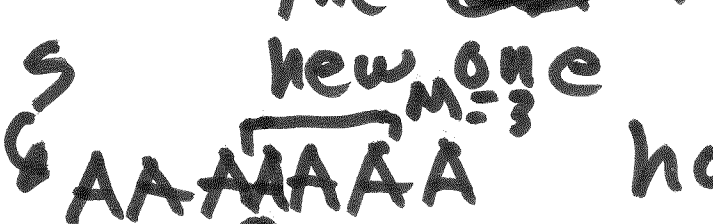
If hash1 != hash2 go to next char. Recompute hash2 in O(1)
If hash1 == hash2 then perform a brut force ~~brute~~ comparison

An example of hash function in a string is adding the ascii values

$$hash(s) = \sum_{i=0}^{n-1} s[i]$$

If we need to recompute hash 2 every single time, it will take O(NM), that is the same as before.

However, if we compute hash2 using the previous value, like subtracting the oldest character and adding the new one



$$hash(s,i,M) = \sum_{j=i}^{j=i+M-1} s[j] \rightarrow \text{takes } O(M)$$

s - string
i - start position
M - length of substring

"Hello world"
i M

this can be rewritten as

$$hash(s,i,m) = hash(s,i-1,M) - s[i-1] + s[i+M-1]$$

previous
remove oldest
add new.

takes O(1) assuming that we have computed the hash(s, i-1, m) already.

Rabin Karp algorithm (haystack, needle)

{

- Compute hash 1 from needle
- compute hash 2 from first M chars in haystack.

while we have not reach end (i=0..N-m)

if hash 1 == hash 2

- brute force match of needle and haystack at i.
- If they match return i

end

Recompute hash 2. Subtract oldest character in hash 2 and add new one

i++;

end

Return Null (No match found)

}

In the typical case (average case), hash1 and hash2 will be equal where only when the strings match. So in average: $O(N)$

(compare hash $O(1)$
recompute $O(1)$)
$* N$

Worst Case:
If we have a false match where the hash1 and hash2 are equal but the string and substring do not match at every position.

Worst Case: $O(MN)$

(compare hash $O(1)$
recompute hash $O(1)$
brut force math $O(M)$
$O(M$
$* N$

Improving hash function

We can improve the hash function by computing it using a polynomial like:

$S[i]$ → character at position i

$X(i)$ → hash value starting at position i

b - a constant

Eg. $b = 5$

$$X(i) = S[i]\,b^{M-1} + S[i+1]\,b^{M-2} \cdots + S[i+M-1]\,b^{0}$$

Note: If $b = 1$ then this hash function is the same as adding the ascii values as before.

Hello world

"hash" → hash = ascii('h')*5^4 + asii('e')*5^3 ...

$$X(i+1) = S[i+1]\,b^{M-1} + S[i+2]\,b^{M-2} \cdots + S[i+M]\,b^{0}$$

$$X(i+1) = X(i)\,b - S[i]\,b^{M} + S[i+M]\,b^{0}$$

$S[i+M-1]\,b$

Recomputing $X(i+1)$ from $X(i)$ takes $O(1)$.

$S[i]$ → remove oldest char

$S[i+M]$ → all new character

Implementation


```
char * rkstrstr(char * haystack,
                char * needle){
  // Rabin Karp matching algorithm.
  // We will use the simple hash function
  // where we add the ascii values of
  // the substring (b==1).
  int hash1=0;
  int hash2=0;
  int M= strlen(needle);
  int N= strlen(haystack);
  if(M>N){
    return NULL;
    // no match
  }
  char * hay= haystack;
  char * need= needle;
  while(*hay && *needle) { //Compute
    hash1= hash1 + *need; // hash1 and
    hash2= hash2 + *hay;  // hash2
    need++; hay++;
  }
```

char *hay-end = haystack + N - M + 1;
hay = haystack;
while (hay < hay-end) {
// compare hash and hash2
if (hash == hash2) {
// likely match. Test it.
char *p1 = hay, *p2 = pat;
while (*p1 == *p2 && *p2 != '\0') {
p1++;
p2++;
}
if (*p2 == '\0') {
return hay;
}
}
hash = update(hash, hay[0], hay[M], M);
hay++;
}
return NULL;

(184)

```
    // recompute hash2
    hash2 = hash2 -(*hay) + hay[m]
    or
                    old char
    hash2 = hash2 - hay[0] + hay[m];
    hay++; // go to next char
}//while
// No match
return NULL;
}
```

Data Compression

Fixed-length encoding

- Each character is represented with the same number of bits Example: ASCII. (8bits)

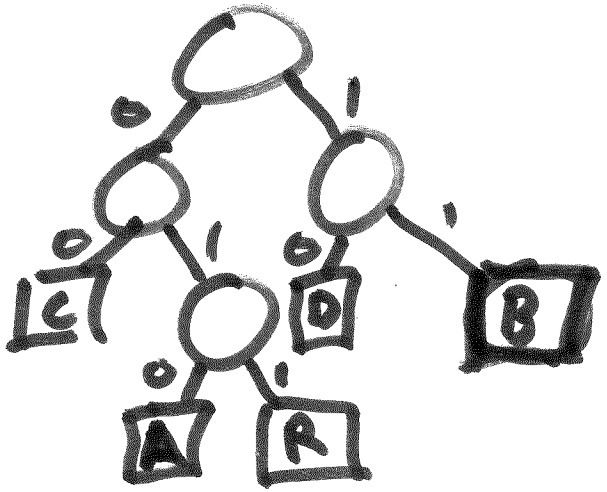
Variable length encoding

- Each character uses binary codes of different length.

Prefix rule

To prevent ambiguities in variable length encoding, each encoding needs to satisfy the prefix rule, that is, no code is a prefix of another code.

We use an encoding trie to define an encoding that satisfies the prefix rule:



A = 010
B = 11
C = 00
D = 10
R = 011

010 11 011 010 00 010 10 11 10 010

A B R A C A D A B R A

11 chars = 29 bits

Compare to the ascii representation

8 bits/char

11 chars x 8 bits = 88 bits

Can we do better than 29 bits?

- Yes. we can do better.
  A is used 5 times and has a representation of 010 (3 bits) and D is used 1 time and has a representation of 10 (2 bits).
- We could have a shorter representation of A and that will give us a smaller representation of the string.

Huffman Encoding Trie

- It builds an optimal encoding trie for a given string. Characters than happen more often than others get shorter representation.

- The algorithm builds a tree bottom-up using the characters and subtrees of smaller frequency first.

A B R A C A D A B R A

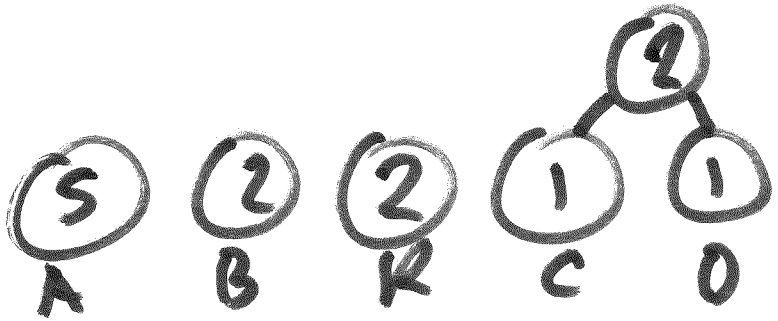
A	B	R	C	D	
5	2	2	1	1	← Frequency

O(n)

Step 1

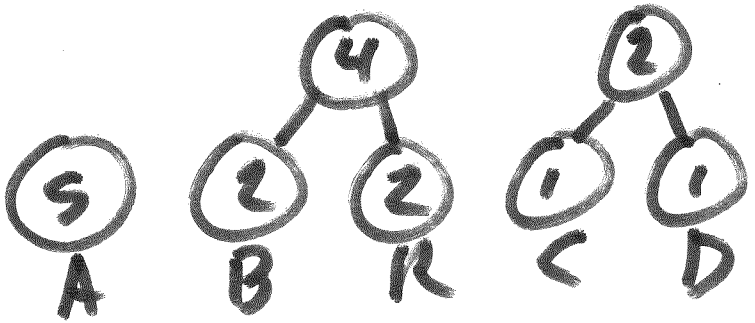
Step 2

Choose the 2 subtrees with minimum frequency.



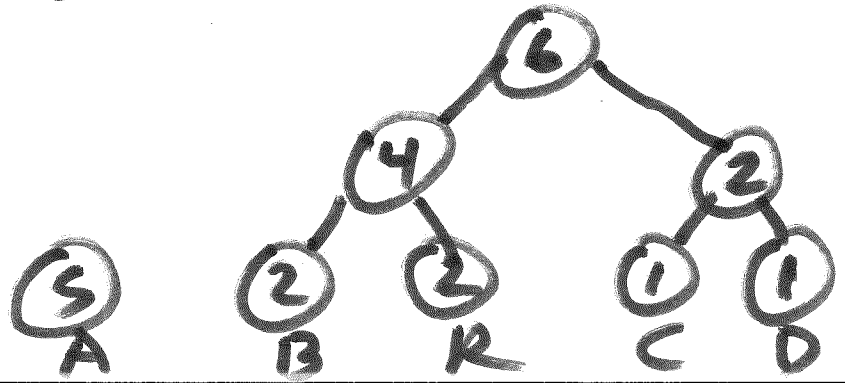
Step 3 -A

Again choose 2 subtrees with min frequency. If there are more than 2 min subtrees choose any.

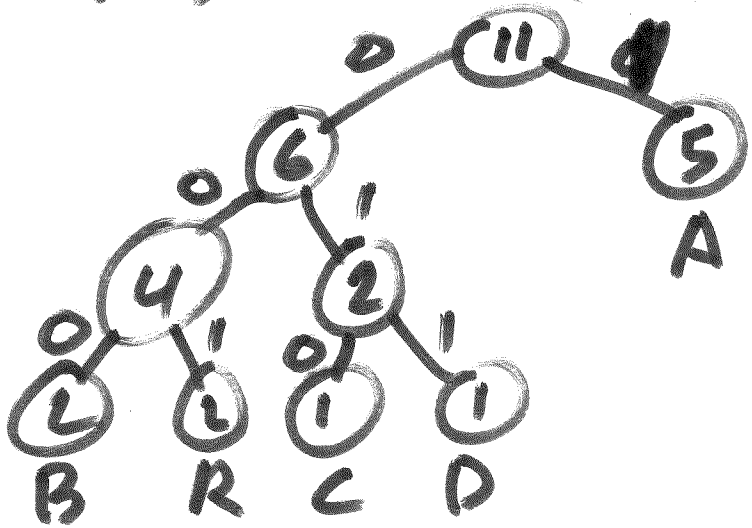


Step 4

Again choos 2 min subtrees



Step 5 choose min subtrees again



Optimal subtree

A - 1
B - 000
R - 001
C - 010
D - 011

ABRACADABRA

$\frac{1}{A}\ \frac{000}{B}\ \frac{001}{R}\ \frac{1}{A}\ \frac{010}{C}\ \frac{1}{A}\ \frac{011}{D}\ \frac{1}{A}\ \frac{000}{B}\ \frac{001}{R}\ \frac{1}{A}$

23 bits

Complexity of Huffman Encoding optimal!

O(n) - Build Frequency Histogram

O(KlogK) - Build tries assuming K characters.

- Huffman encoding will not do well if
  - We use all characters.
  - All characters appear in the string with the same frequency.
- However with normal text some characters appear more often than others so Huffman encoding will give a good compression.

Quiz 22

In what situation Huffman Encoding will not give a smaller representation:

a) All characters are used

b) All characters have the same number of occurrences.

c) a and b

d) a or b

Quiz 21

In Rabin-Karp string matching the worst case time is

a) $O(N)$

b) $O(M)$

c) $O(MN)$

d) $O(N^2)$