* Recursive:
* If you want to use counter🡪put “+1” at each return function.
* AVL tree—rotation
* Insert O(log n)
* Search O(log n) worst case, because the tree is always gonna be balanced.
* Delete O(log n)
* AVL tree insert is slower than normal trees.
* 44
* 17 62
* 50 78
* 48 54 88
* rotate to:

62

44 78

17 50 88

48 54

good idea or bad idea?

Know big picture of AVL trees.

Tell whether the tree is balanced or not, after add an value.

Memorize all the rotation? –No.

--Red/black tress.

--splay tree

--(2,3)-tree, exact balance everywhere.

2.3 tree, for binary tree as well, the insert always happens at the leaf.

2.3 tree:

insert: O(log n)

delete: O(log n)

search: O(log n)

2.3 tree—add to an existing leaf.

50

70 90

30 39

100

80

60

40

10 20

* add rule:
* add to leaf, if it has three, move the middle one and break the leaf into two.
* <http://en.wikipedia.org/wiki/2–3_tree>
* array of n itns, int a[], int n, and int M; every A[i] belongs to [0,m]
* output list of what is duplicated. DO so in O(nt m)
* 4 4 4
* What if I have a program using binary search tree, saving the tree in the file.
* 7 l (leaf)
* 12 l
* 8 I (internal vertices)
* 17 l
* 23 l
* 20 i
* 15 i
* for each node:
* act on the left
* act on the right
* act on the node.
* Post-order traversal.
* Three orders of traversal:
* 1. operate on left
* 2. operate on node
* 3. operate on right
* Pre-order:15 8 12 20 17 21
* Relatively balanced: ht(left) and ht(right) after <=1
* Operation:
* **Add:**
* Expected time: O(log n)
* Worst case time: O(n) –like linkedlist
* **Contains:**
* Expected time: O(log n)
* Worst case time: O(n)
* **Delete**:
* Expected time: O(log n)
* Worst case time: O(n)
* If can always be a linkedlist sort of thing.
* **Removal**:
* If leaf parent’s pointer to it is null
* Change the pointer for its parent to its child.
* **The third type of traversal:**
* Operate on node:
* 1 2 3 4 5 6 7
* <http://www.cise.ufl.edu/~sahni/cop3530/slides/lec216.pdf>
* in-order: they are in order.
* post-order: great for evaluating the syntax tree

task:

create empty BST

add all of array value to it

print in-order traversal

time: O(n\*(log n))

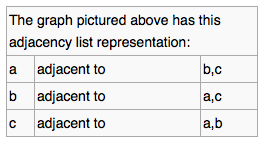
* worst case: O(n^2)
* the next element, if greater than what I was looking for, go next.
* How to add a value to this: add 8,
* How many levels O(log n)
* P[3 logn levels]<=n/2^(3\*logn) =1/n^2
* SKIPLIST
* Time:
* “with high probability”
* find: O(log n)
* add: O(log n)
* remove: O(log n)
* How remove work: find the value, delete it, connect the pointer and find its up pointer delete all of them.
* <http://en.wikipedia.org/wiki/Skip_list>
* performance:
* increase levels-increase memory usage.
* Binary search tree, performs, in expectation.
* The skiplist performs with high probability.
* Space usage: O(n) for BST, skiplist O(n) 2n nodes. +2 logn (negative and positive infinity).
* For skiplist: worst bounded case for space usage: O(n\*log n)
* <http://en.wikipedia.org/wiki/Red–black_tree>
* map data structure:
* map<string,int> m;
* m[“look”]=4;
* string, in this case, is a key.
* UCLA ID:
* A billion elements. Wasted space.
* h(student)=’’ID#” 10000
* //first 5 digits
* key: what indentifies each value information associated.
* Small L is good for this
* Want the table to be sparse.
* The disadvantage of sparse table:a lot of missed memory.
* What if I want to start my table small.
* Hash table:
* Yes o(1) lookup
* O(1) add
* O(1) delete
* Hash table.
* Priority queue
* Priority queue: as a queue
* Add: O(1)
* Find: O(n)
* Delete: O(1)-add t1
* O(n)
* Priority queue as balanaced binary tree:
* Add: O(log n)
* Find min: O(log n)
* Double-min: O(log n)
* Priority queue as a heap:
* Add: O(log n)
* Find-min O( log n)
* Delete min: O(log n)
* A heap is a binary tree. (not a binary search tree )
* Heap: every node is smaller than its parent node. (min heap)
* There’s also a max heap, where each element is largers than it’s parent.
* Notice there’s no order for the children.
* Complete, not necessarily full.
* Find min(look at the root).
* STLmap,STL set.
* The heap is one maximally efficient implementation of an [abstract data type](http://en.wikipedia.org/wiki/Abstract_data_type) called a [priority queue](http://en.wikipedia.org/wiki/Priority_queue), and in fact priority queues are often referred to as "heaps", regardless of how they may be implemented. Note that despite the similarity of the name "heap" to "[stack](http://en.wikipedia.org/wiki/Stack_(abstract_data_type))" and "[queue](http://en.wikipedia.org/wiki/Queue_(abstract_data_type))", the latter two are abstract data types, while a heap is a specific data structure, and "priority queue" is the proper term for the abstract data type.
* No heap node.
* Transform into an array:
* 0 1 2 3 4 5 6 7 8 9
* E H Q M I S T R Y Z
* From top to bottom, left to right.
* O(n) time operation. Find the smallest.

* Chemist
* Approach 1:
* O(nlogn )
* H[0 0] is a heap
* How do I make H[0 1] a heap?
* Add H[i] to heap
* Approach 2:
* n/2 all leaf nodes are heap
* n/4 their parents one op
* n/8 their parents
* approach 2:
* n/2 all lef nodes are heap
* n/4 their parents one op
* n/4 their parents two ops.
* O(n)
* --a converging sum
* spreading out the cost.
* Given any array, you can heap it in O(n) times.
* Do k remove min operation, and restore the elements that been removed.
* Each remove min take log n time do k of them.
* If k is larger, then k(log n) could be larger than n.
* O(k logn +n )
* A heap is only an array.
* C H E M I S I
* E H S M I T C
* H I S M I E C
* Shell sort:
* Final:
* C++ syntax is not going to be emphasized
* In fact, if you asked to write a function, not even be required to write it in c++
* Open hash maps (how google works!)
* closed: chained
* We have an array, when put data in, base on the hash function, if occupied, we movie
* Open: a list of pointers.
* To seaerch out hash table
* we compute a target bucket number
* we then look in that bucker for out value
* if we find it, great!
* If we don’t find the value…
* If we want to store up to 1000 items in an open hash table and be able to find any items in rougly 1.25 serches
* 1.25=1+L/2
* total=total%hash\_table\_size;
* choosing the hash function:
* 1. the hash function must always gives us the same buck # for a given input value.
* Some hashtable:
* CFC32
* CRC64
* MD5
* SHA2
* The maxheap:
* Which is the following are valid maxheaps?
* 1. the value contained by a node is always >= the values of the node’s children (they can be the same)
* how to sort:
* remove the biggest value from the heap
* add to the array.
* 3
* For list a,
* .end() function can only be assigned to the iterator.
* int i[] = {50, 29, 100, 72};
* list<int> listy(i,i+2);
* list is 50,29
* if:
* int i[] = {50, 29, 100, 72};
* list<int> listy(i+1,i+2);
* list is 29.
* bool isNegative (int i) {
* return i > 0;
* }
* int main () {
* int i[] = {-10, 22, 50, 29, 100, 72, -5, 17};
* list<int> listy(i, i+8);
* list<int>::iterator b = listy.begin();
* list<int>::iterator e = listy.end();
* cout << \*find\_if(b, e, isNegative) << endl;
* }
* find the first element.
* Breath first search:
* ---shortest path
* ---bipartite testing
* depth first search:
* weakly connected—if I remove the arrows (turn a directed graph to an undirected graph, it would be a connected component)

strongly connected component: directed graph.

Degree:size of adjacency list.

In-degree:number of incoming edges.



any vertex with a in-degree of zero, is not part of any cycle.

While (V=/=0)

Remove all vertices

With in degree 0.

* For each e=(u,v)
* count[v]++; O(m)

For each V in G

If count=0, remove v O(n)

We represent the running time of a graph using two parameters:

N and M.

Create count[]

Queue<vertex> r;

add everything to it that has a in-degree of zero.

Whiel(r=/=empty)

U<- r.remove first

For each e=(u,v)

Counter[v]--; O(m)

If count[v]==0

Add v to r.

If the queue is empty, O(n+m)

Euler tour: O(m+n)