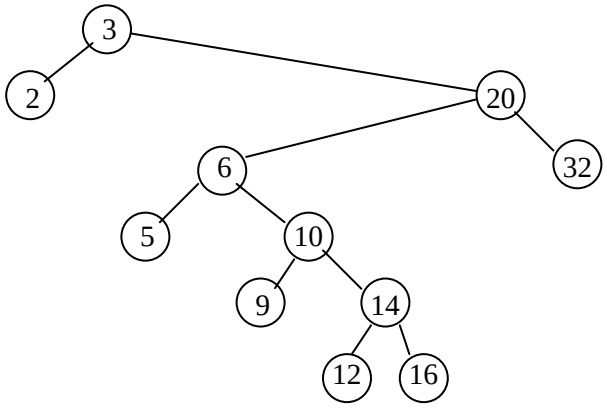
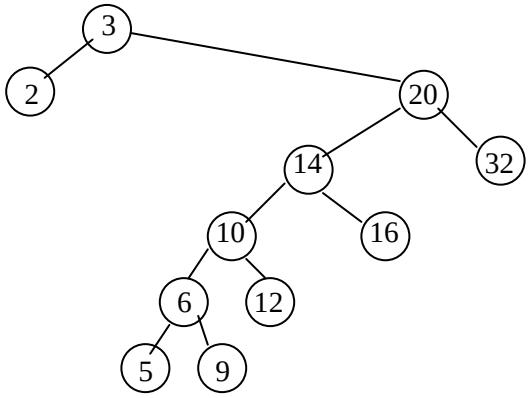
1. Class program (15 points)

This question will attempt to determine whether you wrote the code for p5. There is no need for you to see sample questions if you actually participated in writing the code.

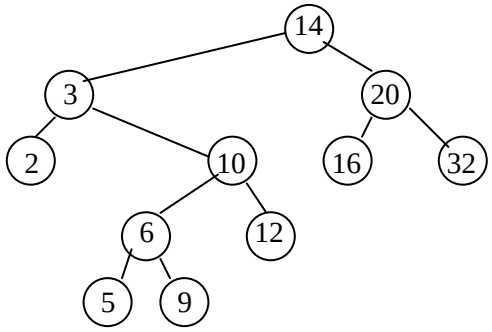
2. Trees (25 points)) Possible "trees" are AVL, splay, B-Trees, and skip lists. Show the result of accessing node 14 in the following splay tree. To receive partial credit you should show the intermediate tree after the zig-zig



After Zig-Zig



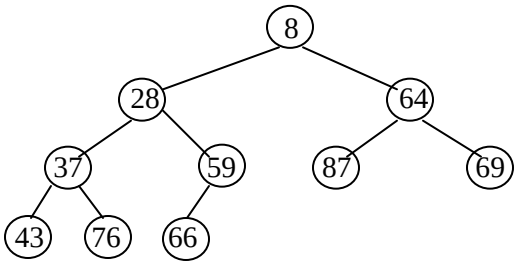
After Zig-Zag



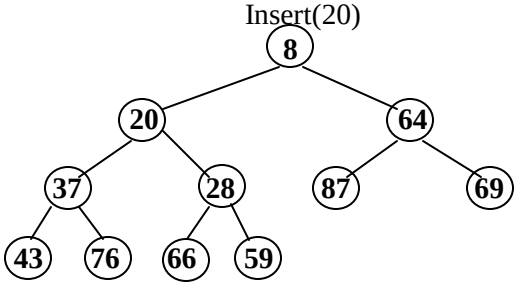
3. Hash (25 points) Possible areas explored are linear probing, quadratic probing, double hashing, and extendible hashes. Assuming linear probing with a hash function of key mod TableSize, an initial TableSize of 3, and rehashing when the load factor exceeds 0.5, show the hash table after each of the operations. Also fill-in the TableSize column.

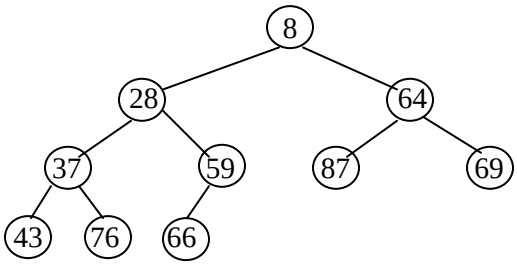
Operation	Table Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
insert 21	3	**21**																			
insert 22	**7**	**21**	**22**																		
insert 4	**7**	**21**	**22**			**4**															
insert 0	**17**	**0**				**21**	**22**	**4**													
insert 1	**17**	**0**	**1**			**21**	**22**	**4**													
delete 0	**17**		**1**			**21**	**22**	**4**													
insert 34	**17**	**34**	**1**			**21**	**22**	**4**													

4. Priority Queues (25 points) Possible priority queues are binary heap, and d-heaps. For each of the following binary heaps, show the state of heap after operation specified.

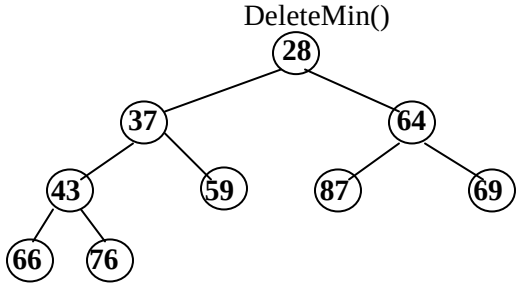


Insert(20)





DeleteMin()



5. Sorting I (25 points) Possible sorts are shellsort, heapsort, quicksort, and radix sort. For Hibbard's increments of 1,3,7, …, 2^k -1, show the state of the array during a Shellsort. Fill in the increment for each row. There may be more rows than needed.

Increment	9	18	4	83	6	59	12	22	3	35	67	16	7	44	1
7	**1**	**3**	**4**	**67**	**6**	**7**	**12**	**9**	**18**	**35**	**83**	**16**	**59**	**44**	**22**
3	**1**	**3**	**4**	**12**	**6**	**7**	**35**	**9**	**16**	**59**	**44**	**18**	**67**	**83**	**22**
1	**1**	**3**	**4**	**6**	**7**	**9**	**12**	**16**	**18**	**22**	**35**	**44**	**59**	**67**	**83**

6. Sorting II (25 points) QuickSort. Sort the following array using a Median of Three and a cutoff of three (series of three will just use insertion sort with no pivot). Place the pivot in the next to last position and the largest of the three in the to last position. Do not find the largest element and do not put it at then end. You may use rows for intermediate steps, but circle each pivot when it is first placed in its correct ("golden") position. Intermediate steps will not be graded. Only those lines with new circled pivots will be graded. (1 point for each correctly circled pivot and 1 point for each correct swap (rounded up))

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	**Pts**
5A	9	5B	4	11	8	12	7	10	13	6	1	5C	3	2	**0**
2	**5C**	**1**	**4**	**3**	**| 5A**	**12**	**11**	**10**	**13**	**6**	**5B**	**9**	**8**	**7**	**6**
1	**|2|**	**4**	**5C**	**3**	**5A**	**6**	**5B**	**| 7 |**	**13**	**8**	**11**	**9**	**10**	**12**	**8**
1	**2**	**3**	**4**	**5C**	**5A**	**5B**	**6**	**7**	**11**	**8**	**10**	**9**	**|12|**	**13**	**6**
1	**2**	**3**	**4**	**5C**	**5A**	**5B**	**6**	**7**	**8**	**| 9 |**	**10**	**11**	**12**	**13**	**4**
1	**2**	**3**	**4**	**5C**	**5A**	**5B**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**1**

7. Graph I(25 points) There will be three graph problems. See the practice midterm for examples of all the possible formats.
8. Graph II (25 points)
9. Graph III (25 points)

10. ADT design (75 points) Please note that this problem does not reflect reality. It is just an ECS 110 problem placed within a familiar context.

Pacific Bell has a huge network of wires with which to connect each phone call within the 530 area code. We are only concerned with calls between different cities within the 530 area code. For each call, the routing program has three tasks: 1) it must find an available route from the source city to the destination city; 2) it must update its data structures so that they accurately reflect assigned paths; and 3) when the call is done, the program should update its data structures so that route is now freed for other calls. Describe and justify your choices of data structure(s) and routine(s) that you would use to implement the routing program. Be sure to describe your three operations, including their time complexity. You may assume the following:

- There are 300 cities in the 530 area code. Each city has one central location for all its routing switches. There are never more than 1,000,000 intercity calls at one time.
- The first three digits of a phone number are called a prefix. Phones with same prefixes are all in the same city.
- A "trunk line" connects one city to exactly one other city. Each trunk line can handle 100,000 calls at a time. The trunk lines rarely reach capacity. Each city is directly connected through trunk lines to five other cities.
- There is one computer that determines the route for all calls. The computer has 256M of RAM. It need not determine the specific wire to use for a connection; it need only determine the route.

First, each city will be assigned a number 0 to 300. The key to this problem is that the trunks rarely reach capacity. This means that a predetermined path from a source to a destination will work the vast majority of the time. Before the program runs, it will determine the unweighted shortest path between every city. To this it will use Djikstra's algorithm. Since this is a sparse graph we can use a heap for O(|E| log|E|) for each city, and thus O(|E| |V| log |E|) for the whole routine, where |V| is 300 and |E| is 1500. These paths would be stored in a 300 x 300 matrix of dynamically allocated vectors. The last city in the path would always be the destination city, as a sentinel

The program would also have a 300 x 300 Capacity matrix of integers that would keep track of the capacity of each trunk. Each position that has an associated trunk would be initialized to 100,000. Since this will be done only once, the $O(300^2)$ would irrelevant.

The program would also have a separate chaining hash table of size 1,000,000, CurrentPaths Each call would be number consecutively (which will restart when MAX_INT is reached). The hash table would store an object that contains the number of the call and the path of the call.

When a call is placed the program would look up the vector in the matrix associated with the source city and destination city. We actually will only need one half of the matrix--we would always have the smaller index in the first position of the indexing. In any case this would take O(1) time. The program would then check to see if the capacities in the Capacity are all greater than zero. If they then it would update the Capacity and Current Path data structures as described below. Otherwise it would do a Dijkstra's shortest path for the source and destination for O(|E| log |V|), and then update Capacity and CurrentPath data structures.

When a path has been found, it would be stored in the CurrentPath in O(1) time. The Capacity matrix edges involved would be decremented in O(|E|) time.

When the call terminated we could retrieve its path from the Current Path hash table in O(1) time. We would also delete its entry from Current Path in O(1) time. The Capacity matrix edges involved would be decremented in O(1) time.