CSC 221 Final Exam Practice Problems Fall 2016

1. Consider the binary search tree in Figure 1. Draw a sequence of diagrams and describe how to delete **30**.

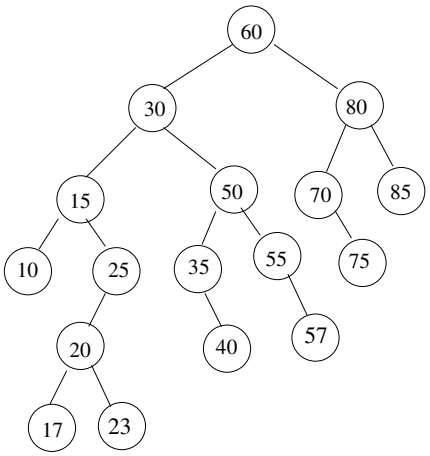
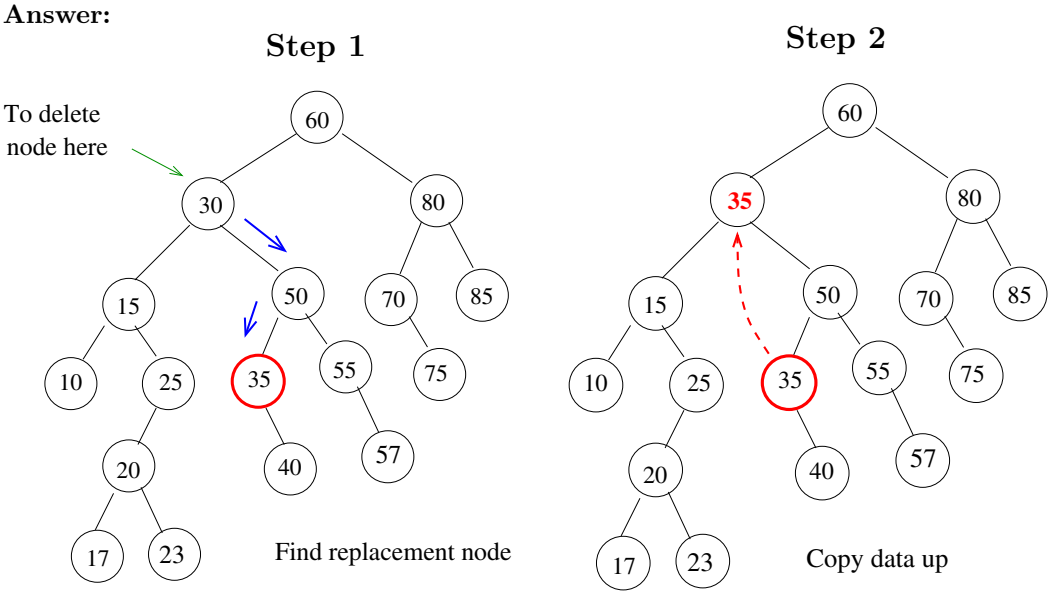
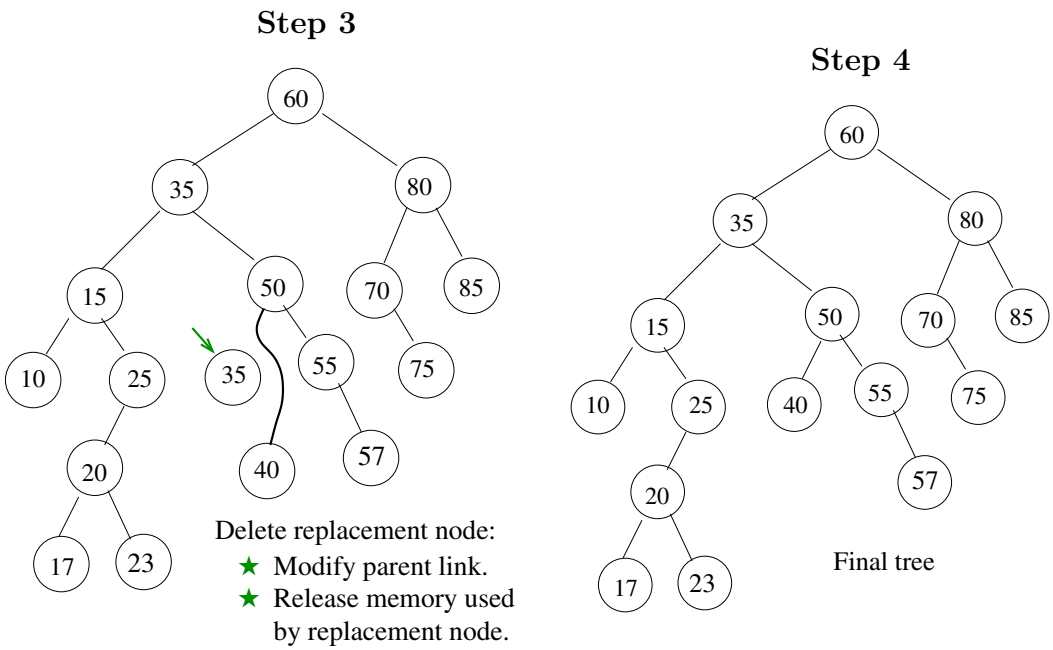


Figure 1: Example Binary Search Tree

Answer:





2. Consider the newly out of balance AVL tree shown in Figure 2. Draw a diagram showing how to re-balance the tree.

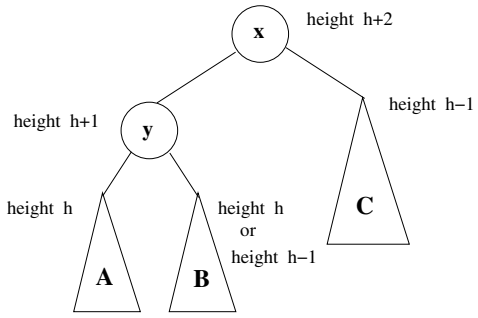
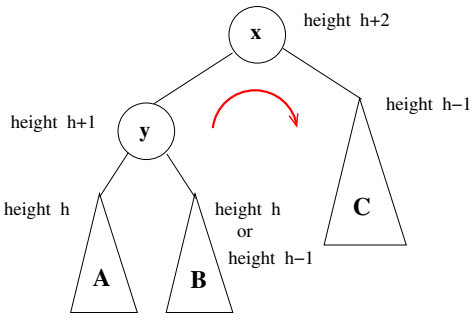
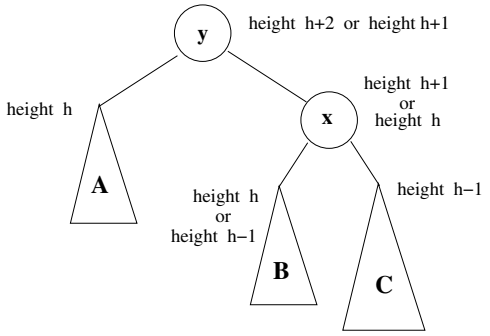


Figure 2: AVL Tree #1 needing rebalancing

Answer: We re-balance the tree by a single clockwise rotation about the unbalanced node **x**.



After the rotation, balance is restored. Also, notice that the rotation operation preserves the binary search property.



3. Consider the newly out of balance AVL tree shown in Figure 3. Draw a sequence of diagrams showing how to re-balance the tree.

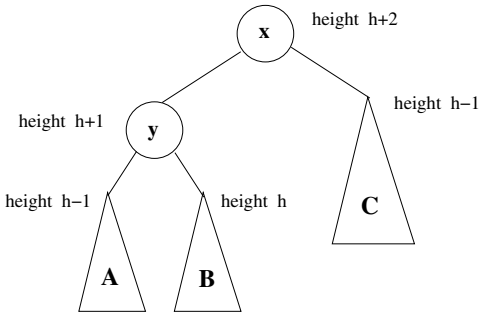
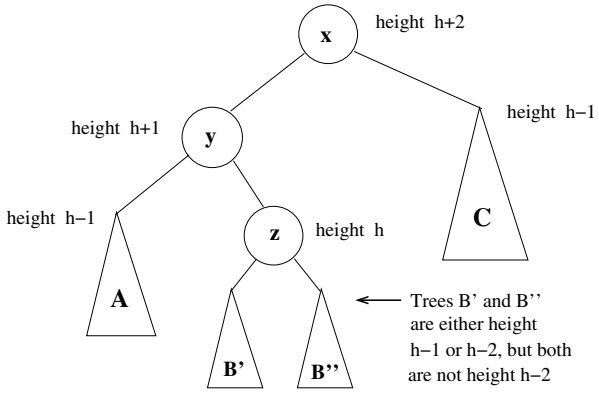
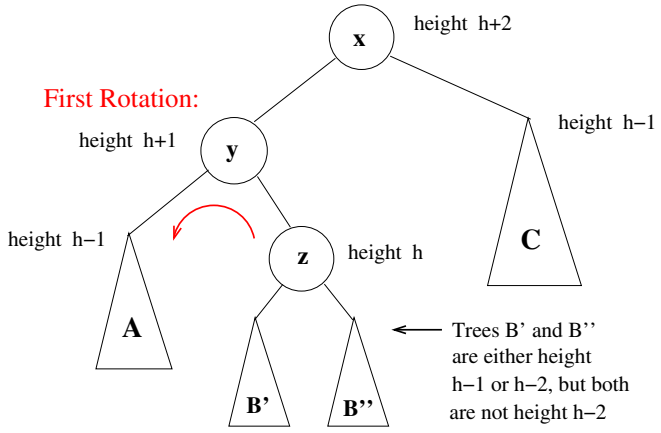


Figure 3: AVL Tree #2 needing rebalancing

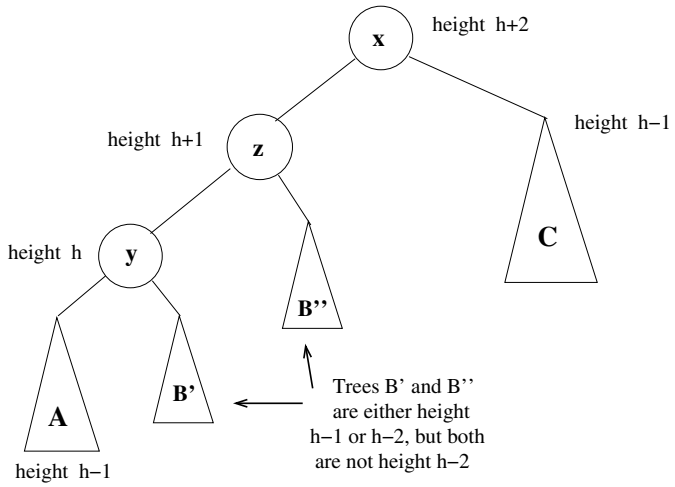
Answer: In this case, we need to use a **double rotation**. First, we expand the sub-tree **B**.



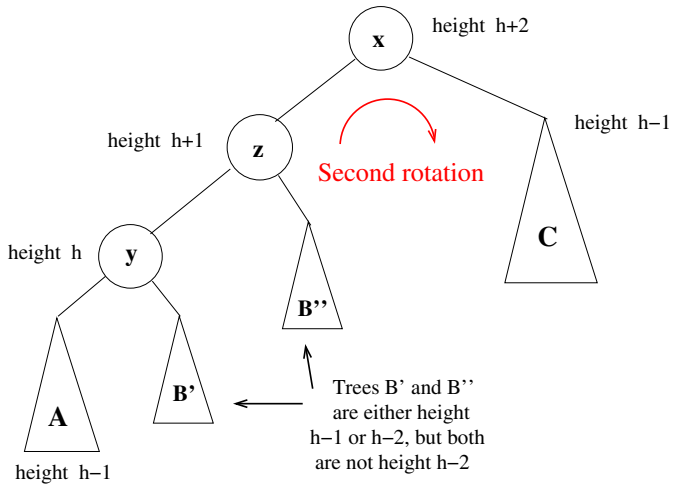
The double rotation begins with a counter-clockwise rotation about node **y**. This operation makes the imbalance worse, but it sets up a configuration in which a clockwise rotation about the root will restore balance. We illustrate it in the following diagrams.



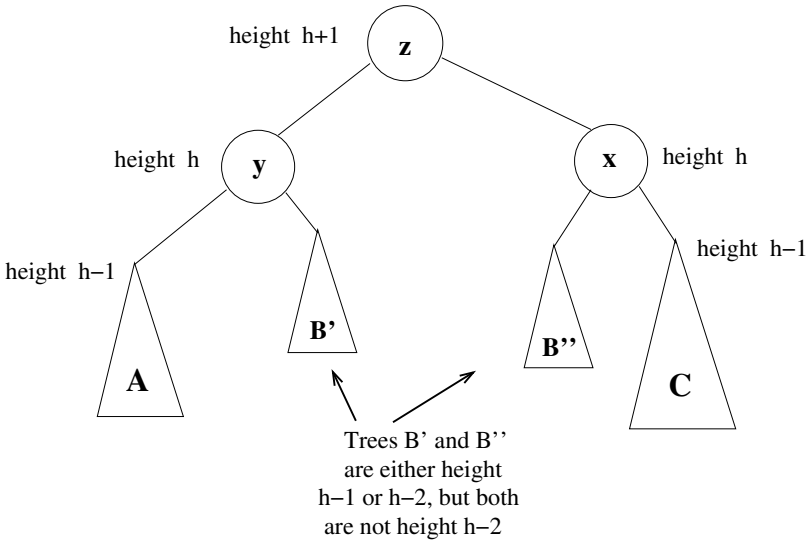
After the counter-clockwise rotation about **y**, we have:



The next rotation is clockwise about **x**:



After the second rotation, balance is restored.



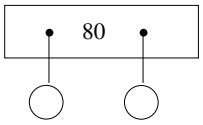
4. Draw a sequence of diagrams showing a sequence of insertions into a 2-3 tree, given the following keys:

```
80 30 17 42 19 61 23 44 31 11 73 55
```

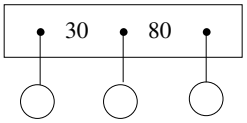
Answer: Start with an empty tree and insert...

Step 0

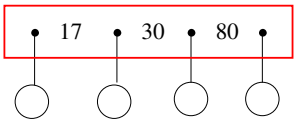
Step 1



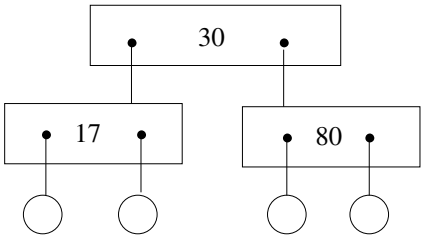
Step 2



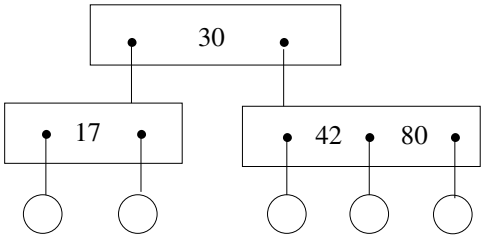
Step 2.5



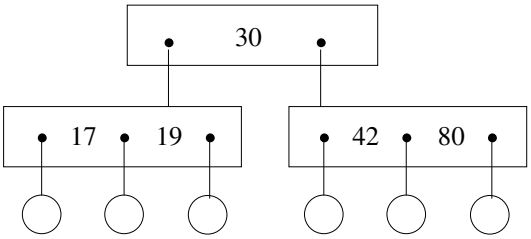
Step 3



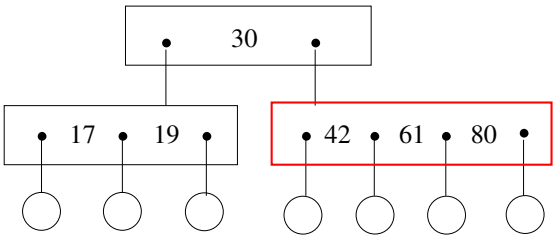
Step 4



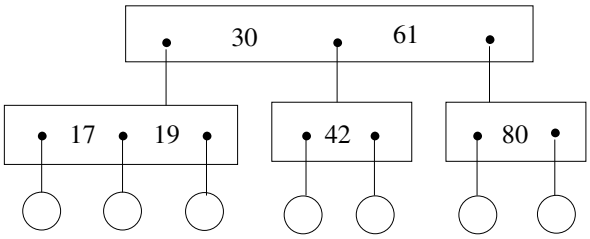
Step 5



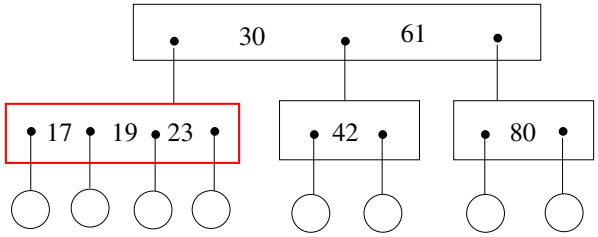
Step 5.5



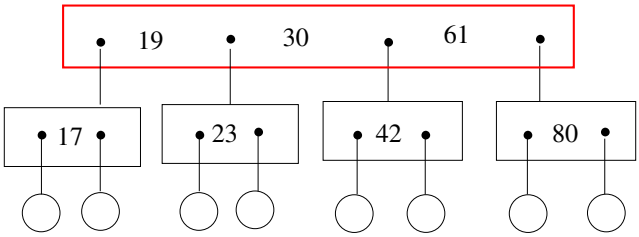
Step 6



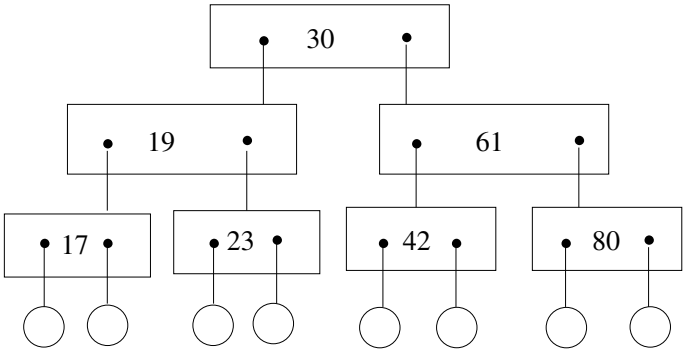
Step 6.1



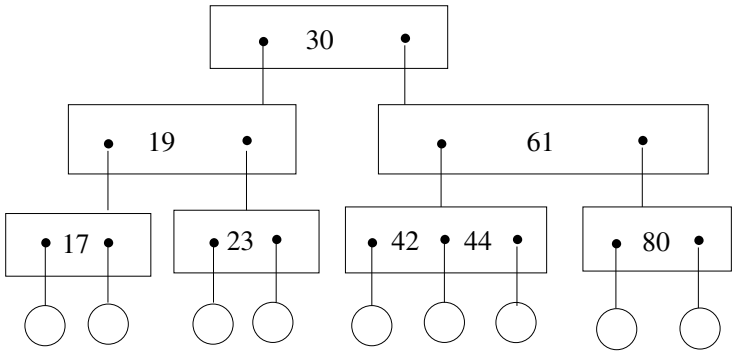
Step 6.2



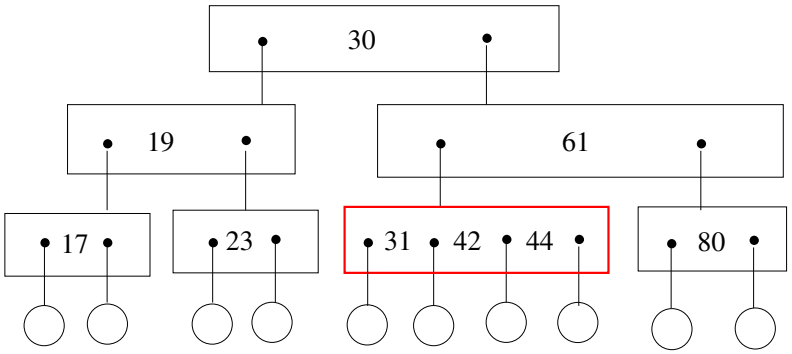
Step 7



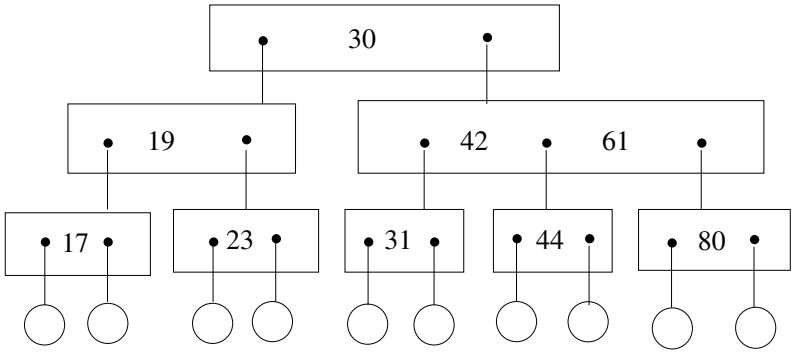
Step 8



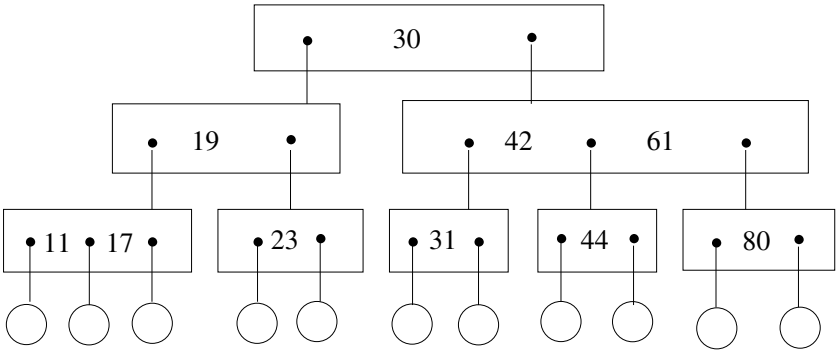
Step 8.5



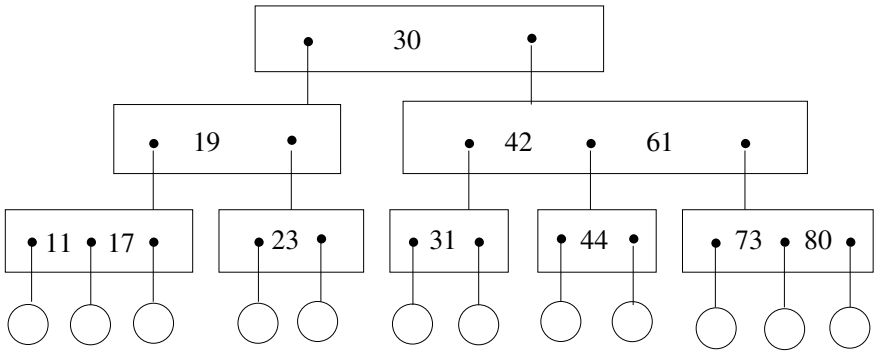
Step 9



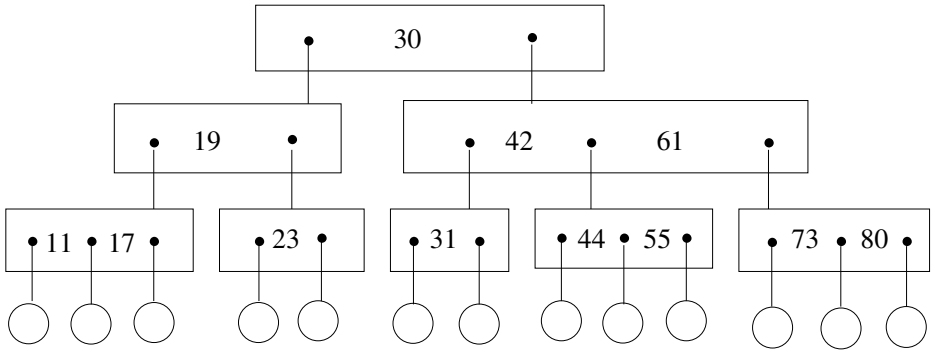
Step 10



Step 11



Step 12



5. Describe the technique known as "polynomial hashing".

   Answer: A character string is a sequence of ASCII (type **char**) values: $\mathbf{a} = [\, a_0, a_1, a_2, ..., a_{n-1} \,]$. Fortunately, type **char** and type **int** are compatible in C++. We define a polynomial using the a_i as coefficients. I.e.,

   $$p_{\mathbf{a}}(x) \;=\; a_0 \;+\; a_1 x \;+\; a_2 x^2 \;+\; a_3 x^3 \;+\; \ldots \;+\; a_{n-1} x^{n-1} \tag{1}$$

   The polynomial in equation (1) is used to define a hash function as follows:

   $$h(\mathbf{a}) \;=\; p_{\mathbf{a}}(x) \text{ modulo } N \tag{2}$$

   where x is a small prime number (e.g., 7, 11, 13) and N is the table size.

6. In the context of open addressing for hash tables, what is **double hashing**?

   Answer: Double hashing is a technique used with open addressing to ensure that when two keys collide, they follow different probe sequences. This avoids clustering.

   Let $h_1()$ and $h_2()$ denote two different hash functions. We define the probe sequence as:

   $$(\; h_1(\,\mathbf{a}\,) \;+\; i h_2(\,\mathbf{a}\,) \;) \text{ modulo } N \tag{3}$$

   where $0 \leq i < N$.

7. Describe the advantages and disadvantages of open addressing for resolving collisions in hash tables.

   Answer: Advantages

   - simplicity
   - it can be efficient, especially when the table is relatively sparse.
   - It can be easily implemented in programming languages which do not support pointers.

   Disadvantages include:

   - The table size limits the number of keys that can be stored.
   - Open addressing is not efficient when the table is nearly full.
   - Deleting from the table is not practical.

8. Refer to the handouts for the depth-first search of a graph and for the breadth-first search of a tree. Design an algorithm to do a breadth-first search of a graph.

   Answer: Breadth first search of a graph is similar to a breadth first traversal in a tree, except

   - We must re-start the search if there are nodes we do not reach from our starting point.
   - We must mark the nodes as we go so we do not endlessly to follow a cycle.

   n algorithm for depth-first search including depth-first numbering is given below:

   Input: Undirected Graph $G = (V, E)$

```
bfs( G )
    dfnumber = 0 ;
    for all v ∈ V do
        visited[v] = false ;
    for all v ∈ V do
        if ( not visited[v] ) {
            bf_explore( G, v )
        }
```

   Input: Undirected Graph $G = (V, E)$ and starting node u

```
bf_explore( G, u )
    Q.enqueue( u ) ;
    while ( Q.not_empty() )  {
        v = Q.dequeue() ;
        visited [ v ] = true ;
        for each edge (v, w) ∈ E  {
            if ( not visited[w] )  {
                Q.enqueue( w )
            }
        }
    }
```

9. Describe two data structures to represent an undirected graph $G = (V, E)$.

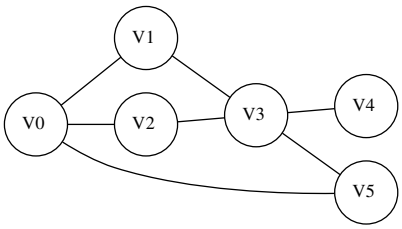
Answer:

An undirected graph G can be represented by an **adjacency matrix** or an **adjacency list**.

An **adjacency matrix**, A, for an undirected graph, G, with n nodes is an $n \times n$ matrix:

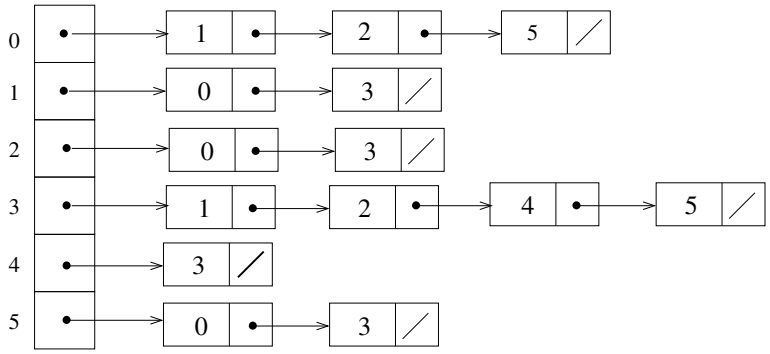
$$A_{i,j} = \begin{cases} 1 \text{ if edge } (i,j) \in E \\ 0 \text{ otherwise} \end{cases}$$

For example, an undirected graph and the corresponding adjacency matrix are shown below:



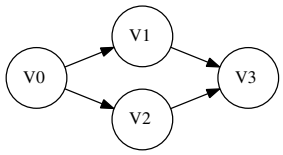
$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Adjacency lists use less memory for sparse graphs. For each vertex, we keep a list of connected vertices. For example, an adjacency list for the graph in shown above is illustrated below:



10. Design an algorithm to detect if a directed graph has a cycle.

Answer: This is a slightly tricky problem. We might try to modify a depth-first search and declare a cycle when we encounter a marked node while searching neighbors. This would be correct for an undirected graph, but it is not correct for a directed graph. For example:



A depth first search will reach v_3 by the path $v_0 \to v_1 \to v_3$ and mark it. The depth first search then "backs up" to v_0 and encounters v_3 again via the path $v_0 \to v_2 \to v_3$. We encounter node v_3 twice, but there is no cycle.

Also, a modified breadth-first search does not work with a directed graph.

However, we have one more trick: a directed graph has a cycle if and only if there exists a back edge. Recall the classification of edges using their pre- and post- numbers. This leads us to the following easy algorithm.

Input: A directed Graph $G = (V, E)$

Output: Returns **true** is a cycle exists, **false** otherwise.

Method:

bool has_cycle(G)

1. Perform a depth first search of G and assign pre- and post-numbers to each node.
2. Classify each edge in E using the pre- and post- numbers from step 1.
   If a back edge is found, return **true**.
3. If no back edge was found in step 2, return **false**.

11. Given the directed acyclic graph (DAG) in Figure 4.

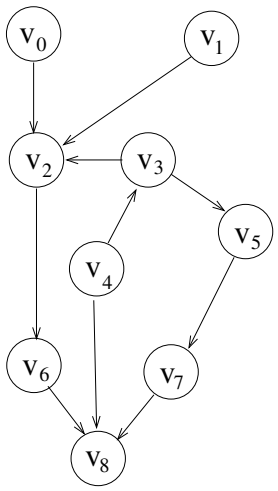
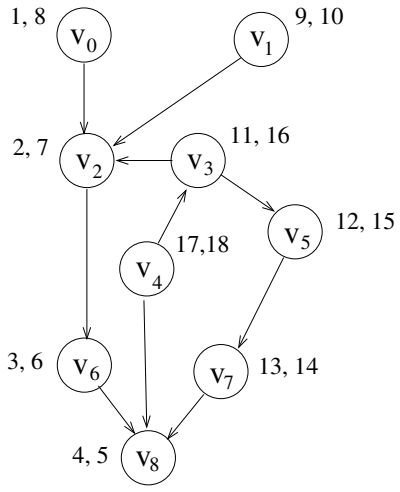


Figure 4: Example Directed Acyclic Graph

(a) Perform a depth-first search and assign pre- and post- numbers.

Answer:



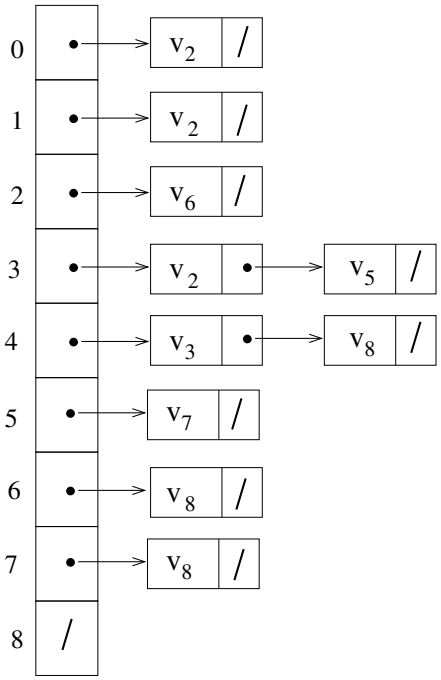
(b) Give a topological sort of the DAG.

Answer:

$$v_4, v_3, v_5, v_7, v_1, v_0, v_2, v_6, v_8$$

(c) Draw a diagram illustrating an adjacency list for the graph in Figure 4

Answer:



12. Give a "Big O" upper bound for the time taken by the following code segment.

```
t = 0 ;
for ( i = 1 ; ( i * i ) <= n  ; i++ ) {
   for ( j = 1 ; j <= n  ; j++ ) {
       t = t + i * j ;
   }
}
```

For simplicity, assume that n is a perfect square, e.g., 4, 9, 16, 25, ...

Answer: We observe that the statement `t = t + i * j ;` takes a constant number of steps (time), and we denote this as c. The inner loop repeats the statement n times. We conclude that the time $T_{\text{inner}}(n)$ for the inner loop is given by:

$$T_{\text{inner}}(n) \;=\; cn$$

We also observe that the outer loop will be repeated for values of i in the set $\{\,1, 2, 3, ..., \sqrt{n}\,\}$. We add the time taken by the inner loop for all values of the control variable i in the outer loop. Therefore,

$$T_{\text{outer}}(n) \;=\; \sum_{i=1}^{\sqrt{n}} T_{\text{inner}}(n) \;=\; \sum_{i=1}^{\sqrt{n}} cn \;=\; cn^{\frac{3}{2}} \;\in\; \mathcal{O}(n^{\frac{3}{2}})$$

13. Suppose you have a collection of records, each containing **name**, **phone number**, **date of birth**, and additional text information. You can assume that each type of information is stored as a character string. Draw a diagram illustrating a collection of such records. Using hash tables, include appropriate structures so that the collection of records can be searched either by name, or by phone number.

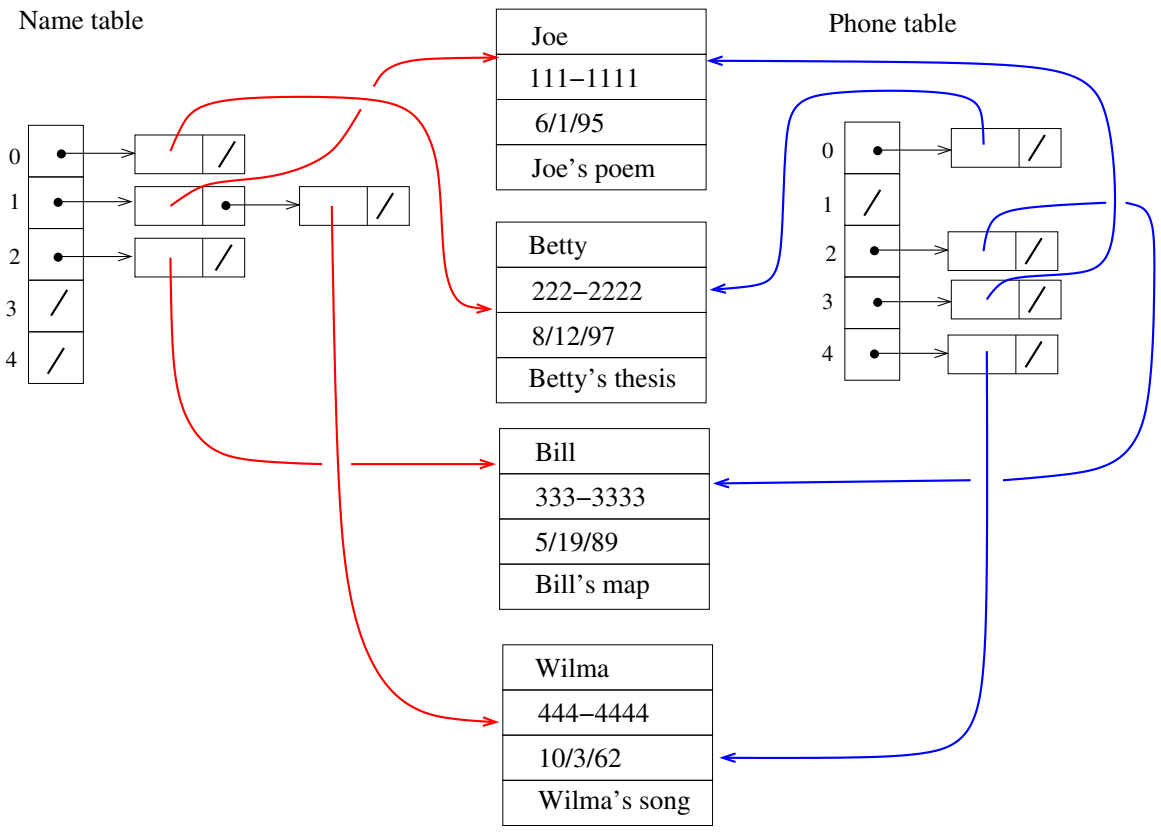
Answer: We illustrate the two-index structure with example data:

Joe	111-1111	6/1/95	Joe's poem
Betty	222-2222	8/12/97	Betty's thesis
Bill	333-3333	5/19/89	Bill's map
Wilma	444-4444	10/3/62	Wilma's song

The names and phone numbers have the following hash values (with table size 5).

String:	Joe	Betty	Bill	Wilma	111-1111	222-2222	333-3333	444-4444
Hash Value:	1	0	2	1	3	0	2	4

The structure is illustrated below:



14. Give declarations for C++ classes needed to implement the indexing scheme in problem 13.

Answer:

In this design, we use only one class type for the hash tables. We declare two different hash table objects in `main()`. The table itself keeps track of the type of search can be done within it. The search type of a table is set by the constructor and can not be changed after the hash table object is created.

```
class record{
   private:
      char * name ;
      char * number ;
      char * dob ;
      char * info ;
   public:
      record() { name = NULL ; number = NULL ; dob = NULL ; info = NULL ; }
//
      void set_name( char * the_name ) ;    // Deep copy the_name.
      void set_phone( char * the_phone ) ;  // Deep copy the_phone.
      void set_dob( char * the_dob ) ;      // Deep copy the_dob.
      void set_info( char * the_info ) ;    // Deep copy the_info.
//
      char * get_name( ) ;
      char * get_phone( ) ;
      char * get_dob( ) ;
      char * get_info( ) ;
} ;

enum searchby { byname, byphone } ;

class listcell {
   private:
      record * rp     ;
      listcell * next ;
   public:
      listcell() { rp = NULL ; next = NULL ; }
} ;

class list {
   private:
       listcell * head  ;
       int count        ;
   public:
       list() {  head = NULL ; count = 0 ; }
       listcell * search( char * target, searchby type ) ;
       void prepend( record * new_record ) ;
} ;
```

```
class hash_table {
    private:
       // Private data:
          searchby htype ;  // Keeps what type of index table this is.
          list * table   ;  // Dynamically allocated array of lists.
          int hcount     ;
       // Private functions:
          int hash_function( char * s ) ;  //  Gives hash value.
    public:
       hash_table(int table_size, searchby type) ;  // Constructor.
       listcell * hash_search( char * target ) ;
       void add_to_table( record & rp  ) ;
} ;



// Design idea:   At the time the table is created, it is given
//                a table type by the constructor.  For example,
//                the main() program could declare two tables.
//
//    const int N = 97 ;
//
//    int main()
//    {
//        hash_table nametab( N, byname ) ; // Constructor call.
//        hash_table phonetab( N, byphone ) ; // Constructor call.
//            .
//            .
//            .
//    }
//
//  Each table "knows" the type of search to do.  The method
//  hash_table::hash_search() searches for the target by passing
//  the search type into list::search() ;
//
```