I've implemented the skip list in two template classes, EventPillar and Event List.

The EventPillar Class

For EventPillar class, each eventPillar stored all events with a given key(For example, all events happened in year 1800 are in one eventPillar, and these events are stored in the vector called “events” which is a public parameter of that eventPillar). Therefore, each eventPillar represented unique year. Each eventPillar also storest an array of forward pointers as “fwdPillars” these pointers point to nodes farther down the skip list. This is also how the pillars were singly linked.

EventPillar has two constructors : 1) **EventPillar(Event* e , int h)** was generally used; “Event* e” was pushed back into vector events; int h passed into pillarHeight; 2) **EventPillar(int)** was used for heads and tails, which initialized the vector events of heads and tails as nullptr.

EventPillar has two methods : 1) **int getHeight(void)** returned the pillarHeight
2) **int getKey(void)** returned the key(the year) of that pillar. For heads and tails, it returned 1000000, which could be regarded as + ∞ in this project.

The EventList Class

While EventPillar objects represent the individual pillars in a skip list, the EventList class manages their insertion(**insert**), deletion(**remove**), and retrieval(**findMostRecent/findRange**). EventPillar has three public fields: head, tail, and maxHeight. The heads and tails pillars are created in the constructor(The constructor sets all forward pointers in the head pillar to the address of the tail pillar. The tail node's forward pointers are set to nullptr.); maxHeight is also the current height of the skip list and initially set as 1.

Search

All operations on a skip list rely on the search algorithm.

The search starts at the topmost forward pointer in the head, which is indicated by maxHeight, and controlled by the outermost for loop. The code always retrieves the value of the key in the next pillar at this level. If the key in the pillar being compared is less than the key (cmpKey), then the next pillar becomes the current pillar, and the comparison key pointer is reset to the value of the key in the next forward node (handled in the while loop). This process continues until the code finds a key that is less than the search key. At this point, the search drops down a level in the current node, and the cmpKey pointer is set to the key value of the next forward node.

Eventually, the search ends up at level 1, in the node just prior to the node less than or equal to the key being sought .It is at this point where all add, delete, and insert operations take place. The updateVec pointers always point to the most previous node prior to the insertion/deletion point at each level**.**

Insert

The insertion uses search algorithm to select a location for a new pillar.

After it locates the perfect spot, it checks if the new event's year already exists. If so, it will push back the new event into the corresponding pillars' "events". Otherwise, randomHeight() will generate new pillar's height. If the height of the new pillar is greater than maxHeight, the insertion will keep doubling the size of skip list until the resulting pillars are at least as tall as the new nodes.

For the method **resize()**, the only thing we need to care is head and tail pillar since other pillars' position won't change. First, the old head is copied into a temporary pillar. Then new head and tail are created and connected. Then new head's fwdPillars on 1 to old maxHeight level were set to the old head's fwdPillars.

For the insertion on each level, the new pillar acquires from updateVec the forward pointers from its previous pillars.

Remove

The search algorithm also allocates the spot for the remove. When the node you want to remove is located, its forward pointers are copied into the forward pointers of the previous nodes.
After the object is deleted, it will check whether the node is the tallest in the list. If so, maxHeight--.

findRange

The search algorithm is used to allocate the "perfect spot" for first and last separately. Then all the events between these two spots were copied into allEvents vector and returned.

findMostRecent

The search algorithm is used to allocate the "perfect spot" for the input year.
There are three possibilities in this method.

1) input year is really small, none of years in skip list is equal or bigger than it. This case tmp == head, return nullptr
2) tmp.fwdPillars[1].year == input year, return tmp.fwdPillars[1].events
3) other case, return tmp.events.