3.

```
20      //start building index of the contents of book
21      clock_t startTime = clock();
22      //read word to temp
23      while (inFile >> temp)
24      {
25          if (temp != "----------------------------------------")
26          {
27              //all transform to lowercase
28              transform(temp.begin(), temp.end(), temp.begin(), ::tolower);
29              //add to IndexMap
30              bookContent.add(temp, bookPage, wordNum);
31              wordNum++;
32          }
33          else
34          {
35              //advance page number; reset wordNum to 1
36              bookPage++;
37              wordNum = 1;
38          }
39      }
40      clock_t endTime = clock();
41      double seconds = static_cast<double>(endTime - startTime)/ CLOCKS_PER_SEC;
42      cout << "The time (1 trial) for this function is: " << seconds << " seconds." << endl << endl;
43      cout << "Number of keys stored in the index (distinct words): " << bookContent.numKeys() << "." << endl << endl;
44      IndexRecord fathers = bookContent.get("fathers");
45      cout << fathers << endl << endl;
```

Code for building the index using Hash Table

BigO Analysis (in general cases):

- **total words in external text file: m = 186,000 (given)**
- **number of distinct words in index: n = 11,325 (from program)**

BigO Analysis (build the index through Hash Table)

Line 23 : O(m); read through all the words in the external text file

Line 30 (**add()**) : **average O(1);** derived below;

Rest of the Line : O(1)

Overall BigO : O(m)

Time required (1 trial) : 0.188s

3a**. BigO Analysis (build the index through Self-balancing BST)**

BigO for insert for BST : O(logn)

This BST will only insert distinct records; n refers to the number of distinct records as stated above.

Overall BigO (BST ver.) : O(m * log(n))

Time required: : 186000*log(11325) * 0.188 / 186000

= 0.762s

3b. **BigO Analysis (build the index vector)**

BigO for insert for vector/array : O(n)

Vector approach follows similar idea, it will first search the entire vector (O(n) through linear search). If it is not found, it will insert at the back of the vector (O(1)).

Overall BigO (vector ver.) **: O(m * n); //m total words * n searches**

Time required: **: 186000*11325 * 0.188 / 186000**

= 2129s

```
107  //Add indicated location to the map.
108  //  If the key does not exist in the map, add an IndexRecord for it
109  //  If the key does exist, add a Location to its IndexRecord
110  void IndexMap::add(const std::string& key, int pageNumber, int wordNumber)
111  {
112      if(key == EMPTY_CELL || key == PREVIOUS_USED_CELL)
113          throw invalid_argument("Invalid key");
114
115      int bucketNumber = getLocationFor(key);
116
117      if (contains(key))
118          buckets[bucketNumber].addLocation(IndexLocation(pageNumber, wordNumber));
119      else
120      {
121          if (keyCount > MAX_LOAD * numBuckets)
122              grow();
123
124          while(buckets[bucketNumber].word != EMPTY_CELL && buckets[bucketNumber].word != PREVIOUS_USED_CELL)
125          {
126              if(bucketNumber == numBuckets - 1)
127                  bucketNumber = 0;
128              else
129                  bucketNumber++;
130          }
131          buckets[bucketNumber].word = key;
132          buckets[bucketNumber].addLocation(IndexLocation(pageNumber, wordNumber));
133          keyCount++;
134      }
135  }
```

Code for add()

BigO for add()

Line 112-113 : O(1)

Line 115 (**getLocationFor()**) : O(1)

Line 117 (**contains()**) : avg O(1)

it follows the BigO of random access from vector. However once collision occurs, it will move further steps to check if the cell is valid (maximum steps is around tableCapacity * 0.7; as it will grow once the keySize reaches the threshold)

Line 118 (**addLocation()**) : O(1)

Line 122 (**grow()**) : *O(numBuckets * total number of records in locations)*

for this hash map, **grow()** will scan through the entire buckets[numBuckets], once the cell contains valid data, it will copy the data (words in string and locations in array).

Other Line : O(1)

Overall BigO for add() **: avg O(1)**

Although BigO of grow() is more significant than the other functions, grow() will only be implemented in specific situation. When we are considering the average BigO for add(), most of the time it will cost O(1) only. Therefore, add() should cost in average O(1).

```
31  //hash function to return the position of the bucket
32  unsigned int IndexMap::getLocationFor(const std::string& key) const
33  {
34      std::hash<string> hasher;
35      unsigned int hashValue = static_cast<unsigned int>(hasher(key));
36
37      //return that mapped onto table
38      return hashValue % numBuckets;
39  }
```

Code for getLocationFor(); BigO: O(1)

```
84  //Returns true of indicated key is in the map
85  bool IndexMap::contains(const std::string& key) const
86  {
87      if(key == EMPTY_CELL || key == PREVIOUS_USED_CELL)
88          throw invalid_argument("Invalid key");
89
90      int bucketNumber = getLocationFor(key);
91
92      while(buckets[bucketNumber].word != EMPTY_CELL && buckets[bucketNumber].word != PREVIOUS_USED_CELL)
93      {
94          if(buckets[bucketNumber].word == key)
95              return true;
96          else
97          {
98              if(bucketNumber == numBuckets - 1)
99                  bucketNumber = 0;
100             else
101                 bucketNumber++;
102         }
103     }
104     return false;
105 }
```

Code for contains(); BigO: avg O(1)

```
63  void IndexRecord::addLocation(const IndexLocation& loc)
64  {
65      locations.push_back(loc);
66  }
```

Code for addLocation(); BigO: O(1)

```
12  //handle resizing the hash table into a new array with twice as many buckets
13  void IndexMap::grow()
14  {
15      IndexRecord* oldPtr = buckets;
16      int oldNumBuckets = numBuckets;
17      numBuckets = 2 * numBuckets;
18      buckets = new IndexRecord[numBuckets];
19      keyCount = 0;
20      for (int i = 0; i < oldNumBuckets; i++)
21      {
22          if(oldPtr[i].word != EMPTY_CELL && oldPtr[i].word != PREVIOUS_USED_CELL)
23          {
24              for(int j = 0; j < oldPtr[i].locations.size(); j++)
25                  add(oldPtr[i].word, oldPtr[i].locations[j].pageNum, oldPtr[i].locations[j].wordNum);
26          }
27      }
28      delete [] oldPtr;
29  }
```

*Code for grow(); BigO: O(numBuckets * total number of records in locations)*