Tutorial 11

Topics

- Searching
- Sorting
- Java Collection Framework
- Tutorial Problems
- Design Exercises

Searching

The objective of searching is to locate a specific record identified through one or more keys. There are several approaches to searching, which are different in the access time for very large databases.

Linear search
Linear search is the sequential process of scanning through the records, starting at the first record, until either a match is found or the search is completed unsuccessfully. Linear search is an appropriate option if the records are in random order or if the data is stored in a linear linked list. This approach may be reasonable if the size of the data is small and the content of the database is highly dynamic (e.g. frequent deletion).

Binary search
Binary search is a straightforward procedure for locating an element in an already sorted array. It is directly analogous to the bisection method for root finding. It operates by splitting the data set in half, determining which half the desired element must be in (if it exists) and then cutting it in half again, etc. That is, binary search is a "divide and conquer" strategy. Binary search is much more efficient than linear search since it requires $O(\log n)$ iterations in the worst case when there is no match. However, it requires preprocessing/sorting the data.

Search with a binary search tree
A binary search tree is a binary tree with the property that the left offspring of any node is "less" than the root node and the right offspring is "larger" than the root node. This property applies recursively to each subtree of the complete tree. To search for an item in a binary tree, we do the following:

- Compare the search key with the root key.
- If it is equal, then the search is completed.
- Else determine if it is smaller than the root key.
- If so, then it must be in the left subtree.
- If it is larger, then it must be in the right subtree.

Therefore at each time, either the left or the right subtree is eliminated from consideration. Eventually the procedure finds the key in the tree or comes to a NULL node and concludes the key is not stored in the tree. The performance of the algorithm depends on the shape of the binary tree.

Hashing
Linear search, binary search, and search with a binary search tree all involve examining records in some systematic sequence until the record being sought is found, or the search is determined to be unsuccessful. Also, the efficiency of these approaches are directly dependent on the size of the data. For very large or wide-ranging data sets, such searching methods may require examining many records. To reduce the time required for a search in large data sets, an alternative approach called hashing may be employed. The trade-off is that hashing requires more memory than the previously described methods. The additional memory used by hashing helps to reduce the number of records examined in any search. It also facilitates the adding and deleting of records.

Hash Table

Hash tables are a quick way to index and search data associated with arbitrary key value types, i.e. names to phone numbers. The solution that hash tables offer is almost like a hack to speed up the search process. It is often time consuming to search a database with complex keys, such as strings, because comparing them is slow, as opposed to comparing integers. Also, traversing linked-lists is slow, compared to accessing arrays. So the trick in hash tables is, take your complex key value, map it uniquely to an integer value, then use that value as an index into an array where you store the data. This then reduces the computation time to be just the time it takes to map your complex key into an integer value because accessing arrays is fast.

Terminology

The following are the hash-table buzzwords:

- **Hash function**: The function that takes your key and maps it into an integer. All hash functions look something like this:
  ```c
  int HashFunction(Key key).
  ```
- **Collision**: When more than one key map or hash to the integer value.
- **hash code**: The value returned by the hash function.
- **Collision resolution**: When you find some way to differentiate between keys when collision occurs. For example, chaining and double hashing.

Chaining

Chaining is a simple way of resolving collisions. Basically, when more than one set of key-value pairs produce the same hash code, you put them all in the same slot. When you go search for them, you just have to search through that list for the right one.

Drawbacks

Although hash tables are fast, it is often hard to find a good hash function that will distribute your data evenly throughout the array and avoid collision. Another problem with hash tables is that it may potentially waste a lot of memory. For example, if your hash function produces values from 1 to 5000, but you only have 1000 items, then you
would have wasted at least 4000 array elements. The most important thing, therefore, is to choose your hash function carefully.

Hash Function Example

Here we give you a simple modulo operator as a hash function, which will only work as a hash function if the objects to be hashed are integers and in that case it isn't usually a good choice. The general hash function is modulo operator(\%). For an integer key, the function

$$h(key\_value) = key\_value \% M$$

will provide an index in the desired range.

Efficiency

Let us consider the efficiency of these searching methods. For example, MIT's registrar's office keeps records of the 9000 students. They access each record by using MIT ID numbers. Suppose the registrar stored the records in an array of 9000 elements in no particular order, and performed a linear search on the records whenever he needed information for a student. In the event of an unsuccessful search, it would have taken 9000 comparisons. Suppose the records were stored in ascending order by MIT ID number. An unsuccessful binary search would require 14 comparisons (log2(9000) is about 14). Now, if hashing is used, an unsuccessful search may take as little as one or two comparisons. Suppose a hash table with space for 20000 entries is used. The hash function might be (MIT ID number)\%20000. Since few ID numbers would map to the same position, far fewer comparisons would be required.

Example

We will look at a small example to see how hashing works, and then discuss some collision resolution schemes. Let's say that there are 5 students in a class and each student has a unique 4 digit ID number. We can store this information in a hash table in the following way. Create a hash table of size 20, and use the hash function ID\%20 to store and search for elements. Let's say our student list is as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>ID Number</th>
<th>ID%20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>1235</td>
<td>15</td>
</tr>
<tr>
<td>Jane</td>
<td>1472</td>
<td>12</td>
</tr>
<tr>
<td>Gary</td>
<td>9209</td>
<td>9</td>
</tr>
<tr>
<td>Cindy</td>
<td>7427</td>
<td>7</td>
</tr>
<tr>
<td>Paula</td>
<td>8328</td>
<td>8</td>
</tr>
</tbody>
</table>

Our hash table would look like:

|   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|   |   |   |   |   |   |   |   |   | 7427|   |   | 8328|   |   | 9209|   |   | 1472|   |   | 1735|   |
Suppose we were searching for a student who had an ID number of 2003 which produces 3 when hashed. We see that there is no entry at position 3, so we can immediately determine that the search is unsuccessful.

Now, how do we deal with collisions if two records have the same hash codes (the result of a hash function is called a hash code)? For example, let's say that Tom, who has an ID number of 5567, joins the class. The hash function would indicate that his record belongs at position 7. However, Cindy's record is already there. Therefore we must deal with this collision. There are three different collision resolution schemes: linear probing, chaining, and double hashing.

Linear probing

When a collision occurs, the new record is simply inserted into the next free position in the hash table. So, in the example above, Tom will be inserted in position 10.

Chaining

Each position in the hash table stores a linked list. Whenever a record is being added to a non-empty table position, it is simply added to the linked list. For above example, Tom will be appended to the linked list at position 7.

Double hashing

When a collision occurs, a secondary hash function is called with the new element's search key. Say this returns the value x. Then, starting from the position indicated by the first hash function, every xth position in the hash table is checked, until an empty position is found. The new record is placed at this position. Notice that the secondary hash function should not return zero, otherwise this scheme will fail.

Here, we construct the secondary hash function as

$$h'(k) = M - 2 - k%(M-2)$$

where M is the the hash table size. The value of the second hash function would then be added to the first hash function result to produce the new hash table index. For above example, M=20. The key is ID number, so for Tom, k=5567, h'(k)=13, so Tom will be inserted in position (13+7)%20 = 0.

Sorting

Sorting is generally used to organize large volumes of data such as billing records, where having data sorted makes it easy to search and manipulate the data. There are several sorting algorithms.

Insertion Sort

This is a technique much like picking up a set of cards. First, pick up one card. Then pick up another card and place it before or after the first, so that the cards in hand are in order. Next, pick up a third card and put it in the position which keeps the cards in hand in order. This process continues until all of the cards have been correctly placed in order. For example, for sorting the array
insertion sort works as follows: start with 7. Then grab the next item, 4. 4 < 7, so 4 should be placed before the 7:

4 7 8 2 1 3
Next, place the 8:
4 7 8 2 1 3
then the 2:
2 4 7 8 1 3
and so on:
1 2 4 7 8 3
1 2 3 4 7 8
until the array of numbers has been sorted.

One possible implementation of insertion sort is:

```java
public void sort( Object[] d, int start, int end )
{
    Object key;
    int i, j;
    // j indexes the element to be inserted
    // i indexes the possible positions where element j could be inserted
    for ( j = start + 1; j <= end; j++ )
    {
        key = d[ j ];
        for ( i = j - 1; i >= 0 && compare( key, d[ i ] ) < 0; i-- )
            d[ i + 1 ] = d[ i ];
        d[ i + 1 ] = key;
    }
}
```

Mergesort

Mergesort is one divide-and-conquer sorting algorithm. The basic idea is to divide the data into two equal or nearly equal data sets. At each level, sort the subsets and then merge the result. This algorithm is easily implemented using recursive method. The base case is the case that just one element need to sort. When you merge the result. You need to compare the corresponding elements in two subsets and use extra memory to store the temporary result.

Still using the example above, we want to sort the array:

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>8</th>
<th>2</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
</table>

divide into two groups:

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>8</th>
<th>^</th>
<th>2</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
</table>

divide again:

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>^</th>
<th>^</th>
<th>8</th>
<th>^</th>
<th>2</th>
<th>1</th>
<th>^</th>
<th>3</th>
</tr>
</thead>
</table>

divide again:

| 7 | ^ | ^ | ^ | 4 | ^ | ^ | 8 | ^ | 2 | ^ | ^ | 1 | ^ | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

cannot divide again, so merge one level:
merge again:

| 4 | 7 | ^^ | 8 | ^ | 1 | 2 | ^^ | 3 |

and merge again:

| 1 | 2 | 3 | 4 | 7 | 8 |

Quick sort

A good sorting technique that requires $O(n \log_2 n)$ running time on average, and $O(n^2)$ running time in the worst case, where $n$ is the number of data. Quicksort is also a “divide and conquer” method for sorting. The basic idea is to partition the data set into two parts and then sort them independently. The following quicksort implementation is simpler but has flaws that lead to worst case performance in situations the algorithm is likely to encounter. (please notice this implementation is different from what we learned from the lecture).

```java
public void sort( Object[] d, int start, int end )
{
    Object o;
    if ( start >= end )
        return;
    int p = partition( d, start, end );
    sort( d, start, p );
    sort( d, p+1, end );
}

private int partition( Object[] d, int start, int end )
{
    //The pivot is initially declared to be the value of the leftmost element of the subarray.
    Object pivot = d[start];
    Object o;
    int low = start - 1;
    int high = end + 1;
    while ( true )
    {
        //Scans left from the right end of the subarray looking for an element less than or equal to the pivot.
        //Note that the body of the while is empty. All the action is done in the --high.
        //Note that since we are using a predecrement operator, high is initialized to end+1.
        while ( compare( pivot, d[--high] ) < 0 ) ;
        //Similarly scans right from the left end looking for an element greater than or equal to the pivot.
        while ( compare( pivot, d[++low] ) > 0 ) ;
        //At this point, if low and high haven't crossed, then we have an element
```
indexed by low that
// belongs above the pivot and an element indexed by high that belongs
below the pivot.
    if ( low < high )
        { o = d[ low ]; d[ low ] = d[ high ]; d[ high ] = o; } //So we swap them.

    //If low and high have crossed we are done and the current position of high is
returned as the pivot, that is as the upper end of the lower partition.
    else
        return high;
    }
}

You should work through it following the partition() method. Note that the upper
partition does not get completely sorted, and in this version of quicksort, the partition
value does not end up separating the partitions.
In order to avoid the worst case for the Quick sort, we can choose the median of the
first, middle and last element as the pivot. Put the smallest of these three as the first
one (start), the largest as the last one (end), the median as the second one, then we
only need partition element[start+1] to element[end-1].

Introducing Java Collection Framework
The following Collection tutorial is compiled from Sun Microsystems online tutorials.

Definitions

A collection is simply an object that groups multiple elements into a single unit.
Collections are used to store, retrieve and manipulate data, and to transmit data from
one method to another. Collections typically represent data items that form a natural
group, like a poker hand (a collection of cards), a mail folder (a collection of letters),
or a telephone directory (a collection of name-to-phone-number mappings).
A collections framework is a unified architecture for representing and manipulating
collections. All collections frameworks contain three things:
Interfaces: abstract data types representing collections. Interfaces allow collections
to be manipulated independently of the details of their representation. In object-
oriented languages like Java, these interfaces generally form a hierarchy.
Implementations: concrete implementations of the collection interfaces. In essence,
these are reusable data structures.
Algorithms: methods that perform useful computations, like searching and sorting, on
objects that implement collection interfaces. These algorithms are said to be
polymorphic because the same method can be used on many different
implementations of the appropriate collections interface. In essence, algorithms are
reusable functionality.

Interfaces

The Collection Interface

A Collection represents a group of objects, known as its elements. The primary use of
the Collection interface is to pass around collections of objects where maximum
generality is desired. For example, by convention all general-purpose collection
implementations (which typically implement some subinterface of Collection like Set or List) have a constructor that takes a Collection argument. This constructor initializes the new Collection to contain all of the elements in the specified Collection. This constructor allows the caller to create a Collection of a desired implementation type, initially containing all of the elements in any given Collection, whatever its subinterface or implementation type. Suppose you have a Collection, c, which may be a List, a Set, or some other kind of Collection. The following one-liner creates a new ArrayList (an implementation of the List interface), initially containing all of the elements in c:

List l = new ArrayList(c);

The Collection interface is shown below:

```java
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

The interface does about what you'd expect, given that a Collection represents a group of objects. It has methods to tell you how many elements are in the collection (size, isEmpty), to check if a given object is in the collection (contains), to add and remove an element from the collection (add, remove), and to provide an iterator over the collection (iterator).

The add method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call. Similarly, the remove method is defined to remove a single instance of the specified element from the Collection, assuming the Collection contains the element, and to return true if the Collection was modified as a result.

The Set Interface
A set is a collection that cannot contain duplicate elements. Set models the mathematical set abstraction. The Set interface extends Collection and contains no methods other than those inherited from Collection. It adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set objects with different implementation types to be compared meaningfully. Two Set objects are equal if they contain the same elements.

The Set interface is shown below:

```java
public interface Set {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

**The List Interface**

A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for:

- Positional Access: manipulate elements based on their numerical position in the list.
- Search: search for a specified object in the list and return its numerical position.
- List Iteration: extend Iterator semantics to take advantage of the list’s sequential nature.
- Range-view: perform arbitrary range operations on the list.

The List interface is shown below:

```java
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element);    // Optional
    void add(int index, Object element);       // Optional
    Object remove(int index);                  // Optional
}
```
abstract boolean addAll(int index, Collection c); // Optional

// Search
int indexOf(Object o);
int lastIndexOf(Object o);

// Iteration
ListIterator listIterator();
ListIterator listIterator(int index);

// Range-view
List subList(int from, int to);

The Map Interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. The Map interface is shown below:

public interface Map {
    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}

Implementations

Set
The two general purpose **Set** implementations are **HashSet** and **TreeSet**. It's very straightforward to decide which of these two to use. HashSet is much faster (constant time vs. log time for most operations), but offers no ordering guarantees. If you need to use the operations in the SortedSet, or in-order iteration is important to you, use TreeSet. Otherwise, use HashSet. It's a fair bet that you'll end up using HashSet most of the time.

**List**

The two general purpose **List** implementations are **ArrayList** and **LinkedList**. Most of the time, you'll probably use ArrayList. It offers constant time positional access, and it's just plain fast, because it does not have to allocate a node object for each element in the List, and it can take advantage of the native method System.arraycopy when it has to move multiple elements at once. Think of ArrayList as Vector without the synchronization overhead.

**Map**

The two general purpose **Map** implementations are **HashMap** and **TreeMap**. The situation for Map is exactly analogous to Set. If you need SortedMap operations or in-order Collection-view iteration, go for TreeMap; otherwise, go for HashMap. Everything else in the **Set section** also applies to Map so just re-read it.

**Algorithms**

**Sorting**
The sort algorithm reorders a List so that its elements are ascending order according to some ordering relation. Two forms of the operation are provided. The simple form just takes a List and sorts it according to its elements' **natural ordering**.

<table>
<thead>
<tr>
<th>Class</th>
<th>Natural Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>signed numerical</td>
</tr>
<tr>
<td>Character</td>
<td>unsigned numerical</td>
</tr>
<tr>
<td>Long</td>
<td>signed numerical</td>
</tr>
<tr>
<td>Integer</td>
<td>signed numerical</td>
</tr>
<tr>
<td>Short</td>
<td>signed numerical</td>
</tr>
<tr>
<td>Double</td>
<td>signed numerical</td>
</tr>
<tr>
<td>Float</td>
<td>signed numerical</td>
</tr>
<tr>
<td>BigInteger</td>
<td>signed numerical</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>signed numerical</td>
</tr>
<tr>
<td>File</td>
<td>system-dependent lexicographic on pathname.</td>
</tr>
</tbody>
</table>
The sort operation uses a slightly optimized *merge sort* algorithm.

Here's a little program that prints out its arguments in lexicographic (alphabetical) order.

```java
import java.util.*;
public class Sort {
    public static void main(String args[]) {
        List l = Arrays.asList(args);
        Collections.sort(l);
        System.out.println(l);
    }
}
```

Let's run the program with the argument "i walk the line" and it will output the following

```
i, line, the, walk
```

The second form of sort takes a *Comparator* in addition to a List and sorts the elements with the Comparator. Details can be referred to [Collections Tutorial](http://java.sun.com) on Sun's website.

**Searching**

The `binarySearch` algorithm searches for a specified element in a sorted List using the *binary search* algorithm. There are two forms of this algorithm. The first takes a List and an element to search for (the "search key"). This form assumes that the List is sorted into ascending order according to the natural ordering of its elements. The second form of the call takes a Comparator in addition to the List and the search key, and assumes that the List is sorted into ascending order according to the specified Comparator. The sort algorithm (described above) can be used to sort the List prior to calling `binarySearch`.

The return value is the same for both forms: if the List contains the search key, its index is returned. If not, the return value is \((-\text{insertion point}) - 1\), where the *insertion point* is defined as the point at which the value would be inserted into the List: the index of the first element greater than the value, or `list.size()` if all elements in the List are less than the specified value. This admittedly ugly formula was chosen to guarantee that the return value will be >= 0 if and only if the search key is found. It's basically a hack to combine a boolean ("found") and an integer ("index") into a single int return value.

The following idiom, usable with both forms of the `binarySearch` operation, looks for the specified search key, and inserts it at the appropriate position if it's not already present:

```java
int pos = Collections.binarySearch(l, key);
if (pos < 0)
    l.add(-pos-1, key);
```
Tutorial Problems

Problem 1
State whether each of the following statement is true or false. If false, explain why.

a. In an insertion sort, items are selected one at a time from a sorted list and inserted in an unsorted set.
   b. Mergesort and quicksort are both divide-and-conquer sorting algorithms.
   c. The basic idea of quicksort is to partition the data set into two parts and then sort each of them with quicksort recursively.
   d. A Collection represents a group of objects. All the Collection implementations are unordered.
   e. A Set is a collection that cannot contain duplicate elements.

Problem 2

Part A
Given a list as follows
10    20   30    40    25    35
Consider the Insertion Sort algorithm discussed in class
public void sort( Object [] d, int start, int end )
{
    Object key;
    int i, j;
    // j indexes the element to be inserted
    // i indexes the possible positions where element j could be inserted
    for ( j = start + 1; j <= end; j++ )
    {
        key = d[ j ];
        for ( i = j - 1; i >= 0 && compare( key, d[ i ] ) < 0; i-- )
            d[ i + 1 ] = d[ i ];
        d[ i + 1 ] = key;
    }
}

(1) How many comparisons would this array require the Insertion Sort to perform a complete sorting?

(2) Show the state of the array after each comparison
(3) We have two loops in this Insertion Sort method, in your answer to (2), mark the state of the array at the end of each iteration of the outermost loop.

**Part B**

Simulate the execution of quicksort on the following array, using the simpler Quick Sort algorithm discussed in class to get the first two subarrays.

6 12 18 9 15 3

Two subarrays are: and

Problem 3

Given `toString()` method of class TreeMap.Entry is defined as follows,

```java
public class TreeMap extends AbstractMap implements SortedMap, {...{
    ...
    private static class Entry implements Map.Entry {
        Object key;
        Object value;
        ...
        public String toString() {return key + "=" + value;} }
}
```

Refer to the Java Online Documentation, write the output of the following piece of code.

```java
import java.util.*;

public class MapExample {
    public static void main(String args[]) {
        Map map = new HashMap();

                          "Andrew", "Melissa", "Alice", "Steve");
        for (int i=0, n=names.length; i<n; i++) {
            String key = name[i];
            Integer frequency = (Integer)map.get(key);
            if (frequency == null) {
                frequency = new Integer(1);
            } else {
                int value = frequency.intValue();
                frequency = new Integer(value + 1);
            }
            map.put(key, frequency);
        }
        Map sortedMap = new TreeMap(map);
    }
}
```
Design Exercises

The 1.00 TAs decided to build a student information system. The main purpose of the system is to allow fast lookup of a student’s grade. The TAs are considering using the following data structures for their student look up system.

Linked List (with each student stored as a node in a linked list)
Hash Tables
Binary Search Tree

Answer the following questions about the above data structures.

Part 1. If binary search trees were used to store students’ grades, answer the following questions:

A. Which of the following should be used as the key if the main purpose is to allow fast look up of a particular student's grade?
   (a) Student's grade.
   (b) Student's ID.
B. Justify your answer above by comparing the efficiency of the student lookup process using the above 2 choices as the key.

Part 2. Which data structure do you recommend the TAs use and why? Justify your answer by comparing the performance of the search time using the 3 data structures when a large number of students are stored. In your answers, clearly specify what will be used as the keys for hashtables and binary search trees respectively.

Part 3. Which of the following statement(s) is/are true when comparing the above 3 data structures:

a) Linked list gives the fastest search time (time it takes to look up a particular student's grade) no matter how many students there are in the linked list.
b) In the worst-case, binary search trees have the same search time (i.e., time to look up a particular item) as linked lists.
c) If one wishes to output the data items in sorted order, hash tables are better than binary search trees.
d) In order to locate an item with a given key in a hashtable, one must compare the key with the keys in all buckets.

Part 4. Give an example input sequence of 5 integer values that will result in a binary search tree that has worst case search time. Use 5 integer values for your example input sequence in the space provided below.

1. _____ 2. _______ 3._________ 4. ________ 5. ________

Draw a diagram of the binary search tree constructed using your example input sequence:
Part 5. The TAs have written the following class Student to be stored as a key in the hash table.
A. Help the TAs finish the below Student class by writing the code for equals so that students with the same IDs are considered equal.

```java
public class Student {
    String name;
    int id;
    int grade;

    public int hashCode() {
        return id;
    }

    public boolean equals(Object s) {
        //insert your code below
    }
}
```

B. Assume that chaining is used as a collision resolution strategy by the hash table. Further assume that the hashtable uses the following hashfunction to map items to a bucket:
\[ h \mod m, \]
where \( h \) is the item's hash code (i.e., key value), and \( m \) is the table size. Give an example sequence of 5 ID values that will result in a hash table with the worst-case search behavior using the above Student class as keys to the hash table when \( m \) is 100.