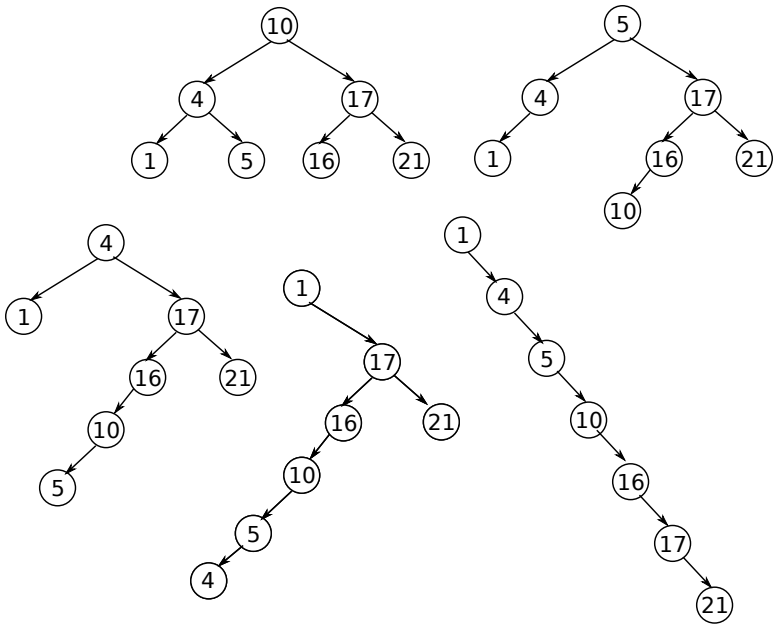
1. The keys and their hash codes are shown in the table on the left. The table on the right shows the contents of the array after all of the keys have been inserted.

key	hash(key)
66	1
54	2
98	7
2	2
38	12
45	6
14	1
17	4
35	9
26	0
88	10
64	12

index	contents
0	26
1	66
2	54
3	2
4	14
5	17
6	45
7	98
8	64
9	35
10	88
11	
12	38

2. There is only one BST of height 2 containing the seven given keys. There are many possibilities for the other trees. Here are some examples of trees of height 2 through 6:

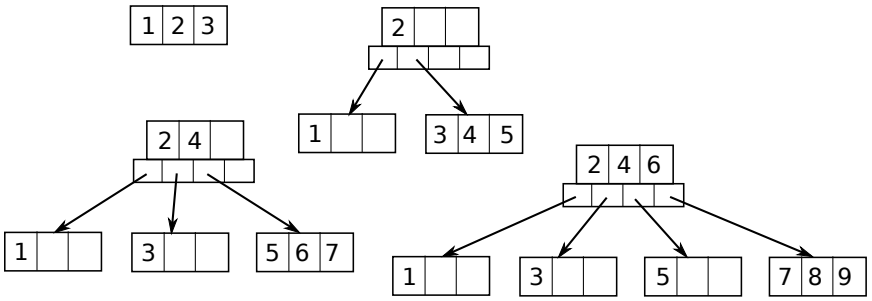


3. Inserting an item into a BST takes time $\Theta(h)$, where h is the height of the tree. The best case for the sorting algorithm would be when the keys are inserted in an order that keeps the tree balanced. In that case, the height is always less than or equal to $\log_2(n)$, and the time for inserting the n items is $\Theta(n * \log(n))$. An inorder traversal of a binary tree containing n nodes only takes time $\Theta(n)$, which is of lower order than $\Theta(n * \log(n))$. The the total best case time is therefore $\Theta(n * \log(n))$. A worst case occurs when the items are inserted in increasing or decreasing order. In that case, the binary tree really has the same form as a linked list, and inserting the k^{th} item takes k steps. To insert all n items takes $1 + 2 + \cdots + n$ steps, which

is $\Theta(n^2)$. The in-order traversal still only takes $\Theta(n)$ time, so the total worst cast run time is $\Theta(n^2)$.

4. If we could build an n-node BST in a run time of lower order than $n * \log(n)$, using comparisons, then we could use the algorithm from Problem 3 to to sort n items in run time of lower order than $n * \log(n)$. This contradicts the known lower bound on sorting using comparisons.

5. A node in the B-Tree can hold 1, 2, or 3 keys. (If it is not a leaf node, then it will hold 2, 3 or 4 pointers to child nodes.) The first three keys will simply be inserted into the root node. Inserting the fourth key will require splitting the root node and making a new root; the fourth key then goes, in this case, into the second child of the root, and the fifth key is placed into the same child. When the sixth key is inserted, it wants to go into that same second child, so the child must be split. There is one more split when the eighth key is inserted. The B-Tree is shown here at several stages of the process:



6. My algorithm reads segments of the file into main memory, sorts them using the standard merge sort algorithm in memory, and then writes the result back to secondary storage in new files (one file for each segment). Make the segments as large as possible, given the size of main memory. The data from the original file has been broken into several files, say K files, and the data in each of the K files is sorted. The algorithm can then merge the K files into one large sorted file, using a merge routine similar to the one from Problem 3b from the previous homework assignment. Note that it is not necessary to read the K files into memory all at once; in fact, it is only necessary to have one page from each file in memory at any given time. This algorithm reads each data item twice and writes it twice. If the original file occupies P pages in secondary storage, then the algorithm reads $2 * P$ pages and writes $2 * P$ pages. The algorithm is essentially just merge sort, and the CPU time is $\Theta(n * \log(n))$, where n is the file size. (Note that to minimize the number of page accesses, it is important to merge the K segments in one pass, rather than merging them two at a time in multiple passes as standard merge sort would do. Each pass in standard merge sort would require reading and writing all of the data from the original file, and there would be a total of $4 * P * \log_2(K)$ page accesses, instead of $4 * P$.)

In the example, the file contains 2^{40} bytes, so at 2^{12} bytes per page, it occupies 2^{28} pages. With 8GB of main memory, we can merge sort a 4GB (2^{33} byte) file segment in main memory, so to cover the whole file, we need $K = 2^7 = 128$ segments. (We probably can't quite do 4GB at a time, so maybe a few more segments.) To read all of the segments into main memory requires 2^{28} page reads. Writing them back to new files takes 2^{28} page writes. Merging the 128 segments requires reading and writing all of the data again, so it adds another 2^{28} page reads and the same number of page writes. The total is therefore 2^{30} page acesses, counting both reads and writes.