11.4-1

Problem

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \mod (m - 1))$.

Solution

With linear probing, we use the hash function $h(k, i) = (h'(k) + i) \mod m = (k + i) \mod m$.

$h(10, 0) = (10 + 0) \mod 11 = 10$. Thus we have $T[10] = 10$.

$h(22, 0) = (22 + 0) \mod 11 = 0$. Thus we have $T[0] = 22$.

$h(31, 0) = (31 + 0) \mod 11 = 9$. Thus $T[9] = 31$.

$h(4, 0) = (4 + 0) \mod 11 = 4$. Thus $T[4] = 4$.

$h(15, 0) = (15 + 0) \mod 11 = 4$. Since $T[4]$ is occupied, we probe again, $h(15, 1) = 5$. Thus $T[5] = 15$.

$h(28, 0) = (28 + 0) \mod 11 = 6$. Thus $T[6] = 28$.

$h(17, 0) = (17 + 0) \mod 11 = 6$. Since $T[6]$ is occupied, we probe again. $h(17, 1) = 7$. Thus $T[7] = 17$.

$h(88, 0) = (88 + 0) \mod 11 = 0$. $T[0]$ is occupied, so probe again. $h(88, 1) = 1$. Thus $T[1] = 88$.

$h(59, 0) = (59 + 0) \mod 11 = 4$. $T[4]$ is occupied. $h(59, 1) = 5$, $h(59, 2) = 6$, $h(59, 3) = 7$ are all occupied. Probing the fourth time, $h(59, 4) = 8$ works. Thus $T[8] = 59$. 


Figure 1: (a) Hash table using linear probe, with hash function $h(k, i) = (k + i) \mod m$. (b) Hash table using quadratic probe, with hash function $h(k, i) = (k + i + 3i^2) \mod m$. (c) Hash table using double hashing, with hash function $h(k, i) = (k + i + 3i^2) \mod m$. 
The final hash table is as shown in figure 1a.

With quadratic hashing, we use the hash function \( h(k, i) = (h'(k) + i + 3i^2) \mod m \). 

\( h(10, 0) = (10 + 0 + 0) \mod 11 = 10 \). Thus we have \( T[10] = 10 \).

\( h(22, 0) = (22 + 0 + 0) \mod 11 = 0 \). Thus we have \( T[0] = 22 \).

\( h(31, 0) = (31 + 0 + 0) \mod 11 = 9 \). Thus \( T[9] = 31 \).

\( h(4, 0) = (4 + 0 + 0) \mod 11 = 4 \). Thus \( T[4] = 4 \).

\( h(15, 0) = (15 + 0 + 0) \mod 11 = 4 \). Since \( T[4] \) is occupied, we probe again, \( h(15,1) = (15 + 1 + 3) \mod 11 = 8 \). Thus \( T[8] = 15 \).

\( h(28, 0) = (28 + 0 + 0) \mod 11 = 6 \). Thus \( T[6] = 28 \).

\( h(17, 0) = (17 + 0 + 0) \mod 11 = 6 \). Since \( T[6] \) is occupied, we probe again. \( h(17, 1) = 10 \). Thus \( T[10] = 17 \).

\( h(88, 0) = (88 + 0 + 0) \mod 11 = 0 \). \( T[0] \) is occupied, so probe again. \( h(88,1) = 4 \), \( h(88,2) = 3 \), \( h(88,3) = 8 \), \( h(88,4) = 8 \), \( h(88,5) = 3 \), \( h(88,6) = 4 \), \( h(88,7) = 0 \) do not work. We finally succeed with \( h(88,8) = 2 \). Thus \( T[2] = 88 \).

\( h(59, 0) = (59 + 0 + 0) \mod 11 = 4 \). \( T[4] \) is occupied. We probe again. \( h(59,1) = 7 \). Thus \( T[7] = 59 \).

The final hash table is as shown in figure 1b.

With double hashing, we use the hash function \( h(k, i) = (h_1(k) + ih_2(k)) \mod m = (k + \{1 + k \mod (m − 1)\}) \mod m \).

\( h(10, 0) = (10 + 0.h_2(10)) \mod 11 = 10 \). Thus we have \( T[10] = 10 \).

\( h(22, 0) = (22 + 0.h_2(22)) \mod 11 = 0 \). Thus we have \( T[0] = 22 \).

\( h(31, 0) = (31 + 0.h_2(31)) \mod 11 = 9 \). Thus \( T[9] = 31 \).

\( h(4, 0) = (4 + 0.h_2(4)) \mod 11 = 4 \). Thus \( T[4] = 4 \).

\( h(15, 0) = (15 + 0.h_2(15)) \mod 11 = 4 \). Since \( T[4] \) is occupied, we probe again, \( h(15,1) = (15 + 1.h_2(15)) \mod 11 = (15 + (15 \mod 10)) \mod 11 = 10 \). Since \( T[10] \) is occupied we probe again. \( h(15,2) = 5 \). Thus \( T[5] = 15 \).

\( h(28, 0) = (28 + 0.h_2(28)) \mod 11 = 6 \). Thus \( T[6] = 28 \).

\( h(17, 0) = (17 + 0.h_2(17)) \mod 11 = 6 \). Since \( T[6] \) is occupied, we probe again. \( h(17, 1) = 3 \). Thus \( T[3] = 17 \).

\( h(88, 0) = (88 + 0.h_2(88)) \mod 11 = 0 \). \( T[0] \) is occupied, so probe again. \( h(88,1) = 9 \) which is occupied. \( h(88,2) = 7 \). Thus \( T[7] = 88 \).
\( h(59,0) = (59 + 0 \cdot h_2(59)) \mod 11 = 4 \). T[4] is occupied. We probe again. \( h(59,1) = 3 \) does not work. \( h(59,2) = 2 \). Thus \( T[2] = 59 \).

The final hash table is as shown in figure 1c.

### 11.4-2

**Problem**

Write pseudocode for HASH-DELETE as outlined in the text, and modify HASH-INSERT to handle the special value DELETED.

**Solution**

**HASH-DELETE(T,k)**

\[
\text{i} = 0 \\
\text{repeat} \\
\quad j = h(k, i) \\
\quad \text{if (} T[j] == k \text{)} \\
\quad \quad T[j] = \text{DELETED} \\
\quad \quad \text{return} \\
\quad i = i + 1 \\
\text{until } T[j] == \text{NIL or } i == m \\
\text{return}
\]

Note that the Deletion code cannot simply mark a slot as empty by storing NIL in it. If we did so, key retrieval will fail for any key \( k \) for which insertion code found the slot occupied and probed beyond it. This issue is solved by marking the slot with a special DELETED value. Inserts can treat such slots as empty, while search simply skips this slot.

The Hash-Insert() code that handles this modified case is as follows.

**HASH-INSERT(T, k)**

\[
\text{i} = 0 \\
\text{repeat} \\
\quad j = h(k, i) \\
\quad \text{if (} T[j] == \text{NIL or } T[j] == \text{DELETED} \text{)} \\
\quad \quad T[j] = k \\
\quad \quad \text{return } j \\
\quad \text{else } i = i + 1 \\
\text{until } i == m \\
\text{error } "\text{hash table overflow}"
\]
12.1-5

Problem

Argue that since sorting n elements takes $\Omega(n \log n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \log n)$ time in the worst case.

Solution

The value of the nodes in the tree can be printed in sorted order in $O(n)$ time using an inorder traversal of the tree. Thus any algorithm that builds a binary tree can be used to solve a sorting problem. Now if it were possible to devise an algorithm that can construct a binary tree with a worst case time bound better than $\Omega(n \log n)$, then we would have a sorting algorithm that has better bound than $\Omega(n \log n)$. Since binary tree construction also uses key comparisons, the $\Omega(n \log n)$ bounds must apply to such an algorithm too. The existence of such an algorithm for constructing a binary tree in $o(n \log n)$ time would thus contradict the lower bound for sorting.

12.2-4

Problem

Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A, b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professors claim.

Solution

The claim is wrong. A simple counter example is shown in figure 2. In the figure, the search is being done for leaf node 3, so the set $B = \{8, 4, 3\}$. There is nothing to the left of the path and so set $A = \{\phi\}$. Set $C$ is all elements to the right of the path, so set $C = \{6\}$. For any element $a \in A$, and $b \in B$ the claim is true, since A is an empty set. But if set $b = 8$ and $c = 6$, the claim fails to hold.
Figure 2: Counter example to Professor Bunyan’s claim. The set A is empty. The search path, where search is performed for the key 3 is marked. The search proceeds from root 8 to the node 4 and then to node 3. So \( B = \{8, 4, 3\} \). Set C is the only key to the right of the path, i.e., \( C = \{6\} \).

12.3-3

Problem

We can sort a given set of \( n \) numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

Solution

Tree-Sort(A)

// let T be an empty binary search tree
for i <- 1 to n
  do Tree-Insert(T, A[i])
Inorder-Tree-Walk(root[T])

Worst case of \( \Theta(n^2) \) occurs when a linear chain of nodes results from the repeated insert operations. It can be easily verified that this follows from the following recurrence. \( T(n) = T(n - 1) + cn \), i.e., to insert \( n \) nodes, the cost is \( T(n - 1) \) (the cost of inserting n-1 nodes) and the cost of inserting the \( n^{th} \) node. Solving this recurrence, we get the \( \Theta(n^2) \) as runtime cost.

Best case of \( \Theta(n \log n) \) occurs when a binary tree of height \( \Theta(\log n) \) results from the insert operations. When inserting \( n^{th} \) node, we are inserting it into a tree with height \( \log(n) \) since the tree is perfectly balanced. Thus the runtime cost is \( \sum_{i=1}^{n} (\log i + d) = \Theta(n \log n) \).
12.3-4

Problem
Is the operation of deletion commutative in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

Solution
The deletion is not a commutative operation. A counter example is shown in the figure 3.

13.1-6

Problem
What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

Solution
Note that the black height $bh(x)$ is defined as number of black nodes on any path from node $x$ to a leaf, not including $x$.

The smallest possible number of internal nodes is $2^k - 1$, which occurs when every node is black. This is produced by a a complete binary tree with k levels
with all nodes black. This tree has 1 root at level 0, 2 internal nodes at level 1 so on. Adding up we get, total internal nodes = \( \sum_{l=0}^{k} 2^l = 2^k - 1 \).

The largest possible number of internal nodes is \( 2^{2k} - 1 \) which occurs when every other node in each path is a black node. This is produced by a complete binary tree which has alternating levels of black and red nodes. Since the black height is \( k \), the height of the tree is \( 2k \). Using similar calculations as before, we find that total number of internal nodes is \( 2^{2k} - 1 \).