Ans to HW4

11.2-1 **Suppose we use a hash function $\mathbf{h}$ to hash $\mathbf{n}$ distinct keys into an array $\mathbf{T}$ of length $\mathbf{m}$. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{\mathbf{k}, \mathbf{l}\} : \mathbf{k} \neq \mathbf{l} \text{ and } \mathbf{h}(\mathbf{k}) = \mathbf{h}(\mathbf{l})\}$?**
Define $X_{kl} = I\{h(k) = h(l)\}$, where h and l are the indices of two different keys, $1 \leq k, l \leq n$. Then

$$E[X_{kl}] = Pr\{h(k) = h(l)\} = \sum_{i=1}^{m} Pr\{h(k) = i\} \cdot Pr\{h(l) = i\} = m \cdot \frac{1}{m} \cdot \frac{1}{m} = \frac{1}{m}$$

Note M as the number of collisions occurring in the array T. Then

$$E[M] = E[\sum_{k=1}^{n} \sum_{l=k+1}^{n} X_{kl}] = \sum_{k=1}^{n} \sum_{l=k+1}^{n} E[X_{kl}] = \frac{n(n-1)}{2m}$$

11.2-2 **Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $\mathbf{h}(\mathbf{k}) = \mathbf{k} \bmod 9$.**
The hash table is shown in Figure 1.

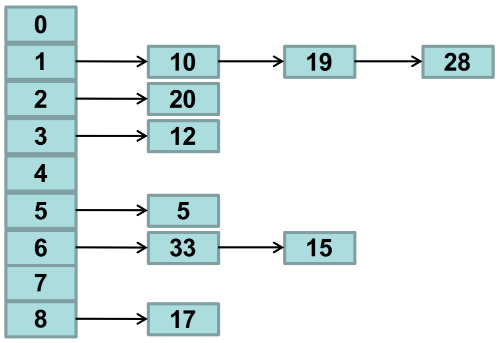
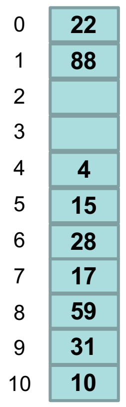


Fig. 1. Hash table

11.4-1 **Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $\mathbf{m} = 11$ using open addressing with the auxiliary hash function $\mathbf{h}'(\mathbf{k}) = \mathbf{k}$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $\mathbf{c_1} = 1$ and $\mathbf{c_2} = 3$, and using double hashing with $\mathbf{h_1}(\mathbf{k}) = \mathbf{k}$ and $\mathbf{h_2}(\mathbf{k}) = 1 + (\mathbf{k} \bmod (\mathbf{m} - 1))$.**
By running a Matlab script to simulate the process of hash mapping, the hash tables by different algorithms can be obtained as shown in Figure 2 to Figure 4.

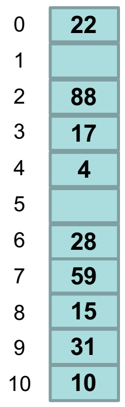
11.4-3 **Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is 3/4 and when it is 7/8.**
According to the knowledge in the slides, the expected upper bound of probe number in an unsuccessful search is $\frac{1}{1-\alpha}$. When $\alpha = \frac{3}{4}$ and $\alpha = \frac{7}{8}$, it is 4 and 8 respectively. The expected upper bound of probe number in a successful search is $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$. When $\alpha = \frac{3}{4}$ and $\alpha = \frac{7}{8}$, it is $\frac{8}{3} \ln 2$ and $\frac{24}{7} \ln 2$ respectively.

12.1-2 **What is the difference between the binary-search-tree property and the min-heap property (see page 153)? Can the min-heap property be used to print out the keys of an $\mathbf{n}$-node tree in sorted order in $\mathbf{O}(\mathbf{n})$ time? Show how, or explain why not.**
The difference lies in: In the binary-search-tree, any node is greater than its left child (as well as any nodes in its left subtree), but smaller than its right child (as well as any nodes in its right subtree). However in the min-heap, any node



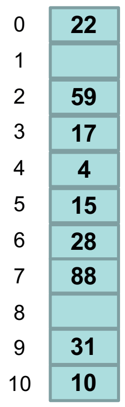
0	22
1	88
2	
3	
4	4
5	15
6	28
7	17
8	59
9	31
10	10

Fig. 2. Linear probing



0	22
1	
2	88
3	17
4	4
5	
6	28
7	59
8	15
9	31
10	10

Fig. 3. Quadratic probing



0	22
1	
2	59
3	17
4	4
5	15
6	28
7	88
8	
9	31
10	10

Fig. 4. Double hash

is no greater than its both children (as well as any nodes in its left sub-heap and right sub-heap both).
The min-heap property cannot be used to print out keys in sorted order in $O(n)$. It can be proofed by reduction to absurdity. Assume there is an min-heap property based algorithm that can print out sorted keys in $O(n)$. Then, one can sorted keys by build-min-heap and this algorithm in $O(n)$. It is contradictory to the fact that any comparison based sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case. So min-heap property cannot be exploited to achieve sorted printing in $O(n)$.

12.2-3 **Write the TREE-PREDECESSOR procedure.**
For a given node x in the binary-search-tree, its predecessor is the greatest key smaller than it. If x has a left subtree, its predecessor is the maximum node in the left subtree, and we should search for the maximum in its left subtree. If x has no left subtree, its predecessor is the first ancestor whose right child is an ancestor of it (or itself), and we should search for the ancestor that meets the requirement. So the Tree-Predecessor procedure for node x can be detailed as follows.

```
Tree-Predecessor(x){
  if x.left$\neq$NIL
    return y=Tree-Maximum(x.left)
  else
    y=x.parent
  end if
  while (y$\neq$NIL and x==y.right)
    x=y
    y=y.parent
  end while
  return y
}
Tree-Maximum(x){
  while x.right$\neq$NIL
    x=x.right
  end while
  return x
}
```

12.2-4 **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.**
As shown in Figure 5, when searching for key 4, the three sets $A = \{2\}$, $B = \{1, 3, 4\}$, $C = \emptyset$. If let $b = 1$ and $a = 2$, then $a > b$. Likewise as shown in Figure 6, when searching for key 1, the three sets $A = \emptyset$, $B = \{4, 2, 1\}$, $C = \{3\}$. If let $b = 4$ and $c = 3$, then $b > c$.

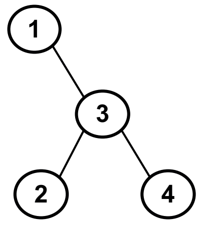


Fig. 5. Linear probing

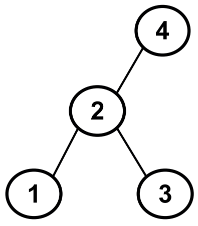


Fig. 6. Quadratic probing

12.2-5 **Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.**

For successor: If a node in binary-search-tree has two children, it means that it has a right subtree, and its successor is the minimum node of the subtree. By proof of contradiction, if the successor still has a left child, there would be an even smaller node in the right subtree. It is contradictory. So the successor cannot has a left child.

Similarly for predecessor: If a node in binary-search-tree has two children, it means that it has a left subtree, and its successor is the minimum node of the subtree. By proof of contradiction, if the successor still has a right child, there would be an even greater node in the left subtree. It is contradictory. So the successor cannot has a right child.

12.3-2 **Suppose that we construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree.**

The former part of Tree-Insert is to find the position where the new key should be inserted. Its strategy is to iteratively compare the value of new key with that of current node, and then decide to search from the left or right in the next. It is performed along the same path as Tree-Search. The latter part of Tree-Insert, i.e. the last step, is to place the new key as a child into the correct position. No new node would be examined in this step. However in the last step of Tree-Search, the searched key would be examined at that last position. So overall, the number of nodes examined in Tree-Search should be one plus the number of nodes examined in Tree-Insert.

12.3-3 **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?**

No matter best case or worst case, the Inoder-Tree-Walk takes $T_2(n) = O(n)$, where n is the total number of elements to be sorted. The difference lies in the binary tree building.

In the best case, the cost of building binary tree is $T_{1b}(n) = O(1) + O(\lg 2) + O(\lg 3) + \cdots + O(\lg n) = O(n \lg n)$.

In the worst case, the cost of building binary tree is $T_{1w}(n) = O(1) + O(2) + O(3) + \cdots + O(n) = O(n^2)$.

Therefore, in the best case, the cost of sorting is $T(n) = T_{1b}(n) + T_2(n) = O(n \lg n)$, while in the worst case, the cost of sorting is $T(n) = T_{1w}(n) + T_2(n) = O(n^2)$.