Chapter 6

Binary Search Trees and Linear Time Sorts.

6.1 Linear Time Sorts

Counting sort. Counting sort assumes that each of the n input elements is an integer with a size at most k. It runs in $\Theta(n+k)$ time, so when $k = O(n)$, counting sort runs in $\Theta(n)$ time. Counting sort first determines, for each input element x, the number of elements less than or equal to x. It then uses this information to place element x directly into its position in the output array. For example, if there are 17 elements less than or equal to x, then x belongs in output position 17. We must modify this scheme slightly to handle the situation where several elements have the **same value**, as we do not want them all to end up in the same position.

```
1  let B and C be new arrays at size n and k
2  for i ∈ [0, k] do
3  |  C_i ← 0
4  end
5  for j ← [1, n] do
6  |  C_Aj ← C_Aj + 1
7  end
8  for i ∈ [1, k] do
9  |  C_i ← C_i + C_i−1
10 end
11 for i ∈ [n, 1] do
12 |  B_C_Aj ← A_j
13 |  C_Aj ← C_Aj−1   // to handle duplicate values
14 end
15 return B
```

Algorithm 1: Counting-sort(A, n, k)

Notice that the Counting sort can beat the lower bound of $\Omega(n \log n)$ only because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array.

An important property of the counting sort is that it is **stable**.

Stable Sort.

We will say that a sorting algorithm is stable if elements with the same value appear in the output array in the same order as they do in the input array.

Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

Radix sort Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column, a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

The Radix-sort procedure assumes that each element in the array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

```
1 for i ∈ [1, d] do
2 |   use a stable sort to sort array A on digit i
3 end
```

Algorithm 2: radix-sort(A, n, d)

Correctness Proof. By induction on the column being sorted.

- Base. Where $d = 1$, the correctness follows immediately from the correctness of our base sort subroutine.
- Induction Assumption. Assume that Radix-sort is correct for any array of numbers containing at most $d - 1$ digits.
- Step. Let A' be the algorithm output. Consider $x, y \in A$. Assume without losing generality that $x > y$. Denote by x_d, y_d their d-digit and by $x_{/d}, y_{/d}$ the numbers obtained by taking only the first $d - 1$ digits of x, y. Separate in two cases:
  - If $x_d > y_d$ then a scenario in which x appear prior to y is imply contradiction to the correctness of our subroutine.
  - So consider the case in which $x_d = y_d$. In that case, it must hold that $x_{/d} > y_{/d}$. Then the appearance of x prior to y either contradicts the assumption that the base algorithm we have used is stable or that x appears before y at the end of the $d - 1$ iteration. Which contradicts the induction assumption.

The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit lies in the range 0 to k–1 (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes $\Theta(n + k)$ time. There are d passes, and so the total time for radix sort is $\Theta\left(d(n + k)\right)$.

Bucket sort. Bucket sort divides the interval [0, 1) into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0, 1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

```
1  let B[0 : n − 1] be a new array for i ← [0, n–1] do
2  |  make B_i an empty list
3  end
4  for i ← [1, n] do
5  |  insert A_i into list B_⌊nA_i⌋]
6  end
7  for i ← [0, n–1] do
8  |  sort list B_i
9  end
10 concatenate the lists B_0, B1, .., B_{n–1} together and
11 return the concatenated lists
```

Algorithm 3: bucket-sort(A, n)

6.2 Binaries Trees and How to Encode Them.

We have already seen, in heaps, that organizing our data in a graph-like structure can offer a speed advantage. For future applications, and in particular for maintaining data in sorted order, we will have to encode our data using binary trees. These trees may not be almost complete and also have to support pointer manipulations, specifically placing a binary tree as a left or right subtree of a given node. To enable this, we will have to treat the **right**, **left**, and **parent** as variables, in contrast to heaps where they are determined completely by the node index. We begin this section by stating definitions.

Definition 6.2.1. *1. Binary Tree: A tree in which any vertex has at most two children.*

2. A descendant of vertex x is a vertex in the subtree whose root is x. A left descendant of vertex x is either a vertex in the subtree whose root is the left child of x, or x's left child.

3. An ancestor of x is a vertex to which x belongs as a descendant.

4. A leaf is a vertex without children.

5. *Height of vertex x is the length of the longest simple path (without cycles) between x and one of the leaves.*

6. *Height of the tree is the height of its root, which is usually denoted by h.*

We encode a binary tree by associating a field to each vertex x, representing its right, left children, and parent. We use the notation x.left to refer to the left child of x, although the physical implementation may differ conceptually. For example, the way binary trees are implemented in Cormen is through 4 arrays. The first stores the value of x, while the others store pointers of specific types. For instance, the array LEFT, where LEFT.x stores the left child of x.

If nothing else has been mentioned, then we can assume that we can add additional fields to the vertices.

6.3 Binary Search Trees.

Binary search tree (BST) is a binary tree which any node x of it:

1. Contains a field key, storing a number x.key.

2. Any left descendant y of x satisfies y.key $\leq x$.key.

3. Any right descendant y of x satisfies y.key $\geq x$.key.

Question. Let T be a binary search tree, Where are the minimum and maximum values of T? (most left and right nodes).

Definition 6.3.1. *Let T be a binary search tree, and let x be a node belonging to it. The predecessor of x will be defined as a vertex y such that y.key $\leq x$.key and y.key is maximal among the nodes satisfying this condition. If we were to set the values of T in sorted order, then the predecessor of x would be located on its left. The successor of x will be defined as y, where x is the predecessor of y.*

Functionalitiy of BST.

1. Search(T, key): returns a pointer to the vertex whose key equals key.

2. Min(T): returns a pointer to the vertex with the minimum value in T.

3. Max(T): returns a pointer to the vertex with the maximum value in T.

4. Predecessor(x): returns a pointer to the predecessor of x.

5. Successor(x): returns a pointer to the successor of x.

6. Insert(T, key): inserts key into T (creates a new vertex).

7. Delete(T, x): removes x from T.

8. Inorder(T): outputs T's keys in sorted order.

Example 6.3.1. *Questions from past exams.*

1. *Write an algorithm that, given a binary search tree T, returns a max heap containing all the elements of T. What is the lower bound for this problem? Explain.*

2. *Write an algorithm that, given a heap H, returns a binary search tree containing all the elements of H. What is the lower bound for this problem? Explain.*

Solution.

1. Remember that any array sorted in reverse order is also a max heap. This is because if $A_i \geq A_j$ for any $i < j$, then it follows that $A_i \geq A_{2i}, A_{2i+1}$ for any i. Therefore, we will output the values of T in reverse order. We do this by applying Inorder in reverse, as given in Algorithm 4. The running time is $\Theta(n)$ and of course it is also the lower bound (as we must read all the inputs for printing it).

Data: r - a vertex in BST
if *r.right is not empty/Null* **then**
  Reverse-Order(r.right)
end
Print r.key
if *r.left is not empty/Null* **then**
  Reverse-Order(r.left)
end

Algorithm 4: Reverse-Order

2. We will initialize an empty binary search tree and then add the elements to it one by one. Each insertion will cost the current height of the tree, so it might sum up to $\Theta(n^2)$. However, next week we will see how those insertions could be done in a way that preserves the balance of the tree, namely guaranteeing that the height of the tree will remain logarithmic.

Example 6.3.2. *Adding fields to BSTs. We would like to support k-smallest element extraction. Suggest a field that will be used for computing the k-smallest element, explain how to maintain it, and write an algorithm that extracts the k-smallest element.*

Solution. We will associate with each vertex of the tree a field that counts the number of nodes whose keys are lower than it (the size of its left subtree). Let's denote it as left-size. The code is given in Algorithm 5. Guideline for proving correctness: if the root has strictly greater than $k-1$ elements, then the k-smallest element is in its left subtree and is also the $k-1$ smallest element in that subtree. If the root has strictly fewer than $k-1$ elements, then it is clear that the k-smallest element is in its right subtree. However, in contrast to the previous case, here the order of the k-smallest element of the whole tree in the subtree will be the substitution between k and the number of nodes in the left subtree $\cup$ the root.

Data: r - a vertex in BST, k-stat parameter
1 $m \leftarrow r$.left-size
2 **if** $m = k - 1$ **then**
3 | **return** r.key
4 **end**
5 **if** $m > k - 1$ **then**
6 | **return** Get-Order(r.left, k)
7 **end**
8 **if** $m < k - 1$ **then**
9 | **return** Get-Order(r.right, $k - (m + 1)$)
10 **end**

Algorithm 5: Get-Order

6.4 Appendix - BST Methods Source: A Mixture of Cormen, Wikipedia, and Codeforces

Data: T - tree, key - key to search for
Result: pointer to vertex with key equal to key
1 **if** T *is empty* **then**
2 | **return** null;
3 **end**
4 **if** T*.key equals key* **then**
5 | **return** T;
6 **end**
7 **if** *key is less than* T*.key* **then**
8 | **return** Search(T.left, key);
9 **end**
10 **else**
11 | **return** Search(T.right, key);
12 **end**

Algorithm 6: Search

Data: T - tree
Result: pointer to vertex with minimum value in T
if T *is empty* **then**
  return null;
end
if $T.left$ *is empty* **then**
  return T;
end
return Min(T.left);

Algorithm 7: Min

Data: T - tree
Result: pointer to vertex with maximum value in T
if T *is empty* **then**
  return null;
end
if T.*right is empty* **then**
  return T;
end
return Max(T.right);

Algorithm 8: Max

Data: x - vertex
Result: pointer to predecessor of x
if x.*left is not empty* **then**
  return Max(x.left);
end
$y \leftarrow x.parent$;
while y *is not empty and* x *is* y.*left* **do**
  $x \leftarrow y$;
  $y \leftarrow y$.parent;
end
return y;

Algorithm 9: Predecessor

Data: x - vertex
Result: pointer to successor of x
if x.*right is not empty* **then**
  return Min(x.right);
end
$y \leftarrow x$.parent;
while y *is not empty and* x *is* y.*right* **do**
  $x \leftarrow y$;
  $y \leftarrow y$.parent;
end
return y;

Algorithm 10: Successor

```
Data: T - tree, key - key to insert
Result: inserts key into T
1  newNode ← create new vertex with key key;
2  y ← null;
3  x ← T;
4  while x is not empty do
5  |   y ← x;
6  |   if key is less than x.key then
7  |   |   x ← x.left;
8  |   end
9  |   else
10 |   |   x ← x.right;
11 |   end
12 end
13 newNode.parent ← y;
14 if y is null then
15 |   T ← newNode;
16 end
17 else if key is less than y.key then
18 |   y.left ← newNode;
19 end
20 else
21 |   y.right ← newNode;
22 end
```

Algorithm 11: Insert

```
Data: T: The input tree
1  if T is not empty then
2  |   Inorder(T.left);
3  |   Output T.key;
4  |   Inorder(T.right);
5  end
```

Algorithm 12: Inorder(T)