

## Chapter 6

# Linear Time Sorts and Lower Bounds in the Comparison Model.

### 6.1 Heapsort (David's group).

We will start by introducing the heap-sort algorithm and providing a proof of its correctness.

```
1  $A \leftarrow \text{Build-Heap}(A)$ 
2 for  $i \in [n]$  do
3   | swap  $A_1 \leftrightarrow A_{n-i+1}$ 
4   | heapsize( $A$ )  $\leftarrow n - i$ 
5   | heapify( $A, 1$ )
6 end
7 return  $A$ 
```

**Algorithm 1:** Heap-sort( $A$ )

Correctness. We are going to prove the following statement.

**Claim 6.1.1.** *At the end of the  $i$ th iteration,  $A_{n-i+1}, A_{n-i+2}, \dots, A_n$  are the  $i$  largest elements of  $A$  placed in order, and  $A_1, A_2, \dots, A_{n-i}$  is a maximum heap.*

*Proof.* By induction.

1. Base.  $A_n$  is set in line (3) to be the root of the heap, and therefore is the maximum of  $A$ . After line (4),  $A_1$  is the parent of the heap's roots as line (4) excludes  $A_n$  from the heap (So the heap inequality holds for any  $j \in [2, (n-1)/2]$ ). Therefore, by the correctness of heapify, we get that after line (5),  $A_1, A_2, \dots, A_{n-1}$  is a heap.
2. Assumption. Assume the correctness of the claim for any  $i' < i$ .
3. Step. Consider the  $i$ th iteration. By the induction assumption,  $A_1$  is a root of the heap  $A_1, A_2, \dots, A_{n-i+1}$  and therefore is their maximum. So after the swapping in line (3), we get that  $A_{n-i+1}$  is the element which is greater than

$n - i$  elements in  $A$ . By using the induction assumption again, we know that it is also less than  $A_{n-i+2}, A_{n-i+3}, \dots, A_n$ , so after line (3) and by the fact that  $A_{n-i+2}, A_{n-i+3}, \dots, A_n$  are the  $i - 1$  largest elements placed in order, we have that  $A_{n-i+1}, A_{n-i+2}, A_{n-i+3}, \dots, A_n$  are the  $i$  largest elements placed in order.

By repeating the same arguments in the base case, we can conclude, based on the correctness of heapify, that after line (5),  $A_1$  is either the root of a heap or the heap inequality did not hold for some  $i \in [2, (n - i)/2]$ . In the latter case, this would contradict the induction assumption (since before line (3),  $A_1 \dots A_{n-i+1}$  were heaps).

□

## 6.2 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. It first determines, for each input element  $x$ , the number of elements less than or equal to  $x$ . Then, it 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  $x$  will be placed in position 17. However, we need to make some adjustments to this method to handle cases where multiple elements have the **same value**. We do not want all of them to end up in the same position.

```

1 let B and C be new arrays at size  $n$  and  $k$ 
2 for  $i \in [0, k]$  do
3   |  $C_i \leftarrow 0$ 
4 end
5 for  $j \leftarrow [1, n]$  do
6   |  $C_{A_j} \leftarrow C_{A_j} + 1$ 
7 end
8 for  $i \in [1, k]$  do
9   |  $C_i \leftarrow C_i + C_{i-1}$ 
10 end
11 for  $j \in [n, 1]$  do
12   |  $B_{C_{A_j}} \leftarrow A_j$ 
13   |  $C_{A_j} \leftarrow C_{A_j} - 1$  // to handle duplicate values
14 end
15 return B

```

**Algorithm 2:** 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.

Note 1: That introduction was copied word by word from a web source. Do not use for commercial purposes.

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

**Algorithm 3:** 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  $\Theta(d(n + k))$ .

Note 2: We will return to analyze the expected (average) running time after the lecture on probability.

**Bucket sort.** [COMMENT] Only if there is time. 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
2 for i ← [0, n-1] do
3   | make Bi an empty list
4 end
5 for i ← [1, n] do
6   | insert Ai into list B[nAi]
7 end
8 for i ← [0, n-1] do
9   | sort list Bi
10 end
11 concatenate the lists B0, B1, ..., Bn-1 together and
12 return the concatenated lists
```

**Algorithm 4:** bucket-sort( $A, n$ )

### 6.3 Sorting in Comparison Model.

We have learned in the lecture that the runs of any sorting algorithm, which does not assume anything about the input's structure except the ability to compare elements in pairs, can be modeled as a binary tree. At each node, the algorithm compares two elements and, based on the result, moves to either the left or right child. Eventually, we reach the leaf of the tree and output the sorted elements. Notice that the nodes only exist in our imagination; the algorithm is not aware of their existence. We will name that tree the comparison tree.

The height of the comparison tree is (at least) the running time of the algorithm. We are going to demonstrate how to use these ideas to lower-bound the running time of the find-peak problem in the comparison model.

**Example 6.3.1.** Prove that no algorithm can find a peak in less than  $\Theta(\log n)$  time.

*Proof.* Consider the following inputs for the problem,  $A^j$  will be a tringle that get's is point at position  $j$ . Namely:

$$A_i^j = \begin{cases} i & i < j \\ j + j - i & \text{else} \end{cases}$$

Now, let's assume, in contradiction, that there exists an algorithm  $\mathcal{A}$  in the comparison model that runs in less than  $\log n$  time. This implies that the comparison

tree representation of its running has a height less than  $\log n$  and the number of its leaves, each associated with a possible output, is less than  $n$ . Remember that we have  $n$  inputs of the form  $A^j$ .

By applying the pigeonhole principle, we can conclude that there are distinct  $j$  and  $j'$  such that  $\mathcal{A}$  reaches the same leaf when given  $A^j$  and  $A^{j'}$  as inputs. Since their peaks are set at different positions, it follows that the algorithm will output a wrong answer for at least one of them.  $\square$

**Example 6.3.2.** *Prove that no algorithm, in the comparison model, can merge two sorted arrays in time  $\Theta(n^{1-\varepsilon})$  for some  $\varepsilon > 0$ .*

*Proof.* Assuming, for contradiction, that there exists an algorithm  $\mathcal{A}$  in the comparison model that merges two given sorted arrays in time  $\Theta(n^{1-\varepsilon})$ . Now, consider the sorting algorithm obtained by replacing the merge subroutine in merge-sort with  $\mathcal{A}$ . On one hand, our new algorithm is a sorting algorithm in the comparison model (otherwise it would contradict the correctness of  $\mathcal{A}$ ), while on the other hand, its running time is equal to:

$$T(n) = n^{1-\varepsilon} + 2T(n/2) \Rightarrow T(n) = \Theta(n)$$

This contradicts our  $\Omega(n \log n)$  lower bound for sorting in the comparison model.  $\square$

## 6.4 Binaries Trees and How to Encode Them. [COMMENT] Only if there is time.

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 6.4.1.** 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  $x.\text{left}$  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  $\text{LEFT}.x$  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.

## 6.5 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  $x.\text{key}$ .
2. Any left descendant  $y$  of  $x$  satisfies  $y.\text{key} \leq x.\text{key}$ .
3. Any right descendant  $y$  of  $x$  satisfies  $y.\text{key} \geq x.\text{key}$ .

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

**Definition 6.5.1.** 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  $y$ .

### Functionality 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 6.5.1.** 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 5. 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).
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 6.5.2.** 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.

**Data:**  $r$  - a vertex in BST

```

1 if  $r.right$  is not empty/Null then
2   | Reverse-Order( $r.right$ )
3 end
4 Print  $r.key$ 
5 if  $r.left$  is not empty/Null then
6   | Reverse-Order( $r.left$ )
7 end
```

**Algorithm 5:** Reverse-Order

**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 6. Guideline for proving correctness: Consider the root, and let's separate into cases based on how many elements the root is greater than.

1. If the root is 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.
2. If the root is greater than strictly fewer than  $k - 1$  elements, then it is clear that the  $k$ -smallest element is also greater than it and located 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  $\{ \text{left-subtree} \cup \text{root} \}$ .
3. The third case is when the root is greater than exactly  $k - 1$  elements, which by definition sets it as the  $k$ -smallest element.

In the first two cases, we are calling Get-Order on trees (BST with our additional field) with smaller height. Therefore, it is clear how one would prove correctness by induction. We will see in the next recitation how to maintain the tree, update the field, and guarantee that the height of the tree will remain logarithmic.

**Data:**  $r$  - a vertex in BST,  $k$ -stat parameter

```

1  $m \leftarrow r.left\text{-size}$ 
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 6:** Get-Order



## 6.6 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 7:** Search

**Data:**  $T$  - tree

**Result:** pointer to vertex with minimum value in  $T$

```

1 if  $T$  is empty then
2   | return null;
3 end
4 if  $T.left$  is empty then
5   | return  $T$ ;
6 end
7 return Min( $T.left$ );
```

**Algorithm 8:** Min

**Data:**  $T$  - tree  
**Result:** pointer to vertex with maximum value in  $T$

```

1 if  $T$  is empty then
2   | return null;
3 end
4 if  $T.right$  is empty then
5   | return  $T$ ;
6 end
7 return Max( $T.right$ );

```

**Algorithm 9:** Max

**Data:**  $x$  - vertex  
**Result:** pointer to predecessor of  $x$

```

1 if  $x.left$  is not empty then
2   | return Max( $x.left$ );
3 end
4  $y \leftarrow x.parent$ ;
5 while  $y$  is not empty and  $x$  is  $y.left$  do
6   |  $x \leftarrow y$ ;
7   |  $y \leftarrow y.parent$ ;
8 end
9 return  $y$ ;

```

**Algorithm 10:** Predecessor

**Data:**  $x$  - vertex  
**Result:** pointer to successor of  $x$

```

1 if  $x.right$  is not empty then
2   | return Min( $x.right$ );
3 end
4  $y \leftarrow x.parent$ ;
5 while  $y$  is not empty and  $x$  is  $y.right$  do
6   |  $x \leftarrow y$ ;
7   |  $y \leftarrow y.parent$ ;
8 end
9 return  $y$ ;

```

**Algorithm 11:** Successor

**Data:**  $T$  - tree, key - key to insert  
**Result:** inserts key into  $T$

```

1 newNode  $\leftarrow$  create new vertex with key  $key$ ;
2  $y \leftarrow \text{null}$ ;
3  $x \leftarrow T$ ;
4 while  $x$  is not empty do
5    $y \leftarrow x$ ;
6   if  $key$  is less than  $x.key$  then
7      $x \leftarrow x.\text{left}$ ;
8   end
9   else
10     $x \leftarrow x.\text{right}$ ;
11  end
12 end
13 newNode.parent  $\leftarrow y$ ;
14 if  $y$  is null then
15    $T \leftarrow \text{newNode}$ ;
16 end
17 else if  $key$  is less than  $y.key$  then
18    $y.\text{left} \leftarrow \text{newNode}$ ;
19 end
20 else
21    $y.\text{right} \leftarrow \text{newNode}$ ;
22 end

```

**Algorithm 12:** Insert

**Data:**  $T$ : The input tree

```

1 if  $T$  is not empty then
2   Inorder( $T.\text{left}$ );
3   Output  $T.\text{key}$ ;
4   Inorder( $T.\text{right}$ );
5 end

```

**Algorithm 13:** Inorder( $T$ )