Advanced Data Structures and Algorithms

Assignment 1

Shubhama Kaushik
15/0131

Q1: For the one-dimensional version of the closest-pair problem i.e. for the problem of finding two closest numbers among a given set of n real numbers, design a divide and conquer algorithm. Analyse the running time of the algorithm.

Ans: For the one dimensional version of the closest pair problem, we will first sort the number in ascending order and then we will walk through the sorted list, computing distance from each point to the one that comes after it. It is easy to see that one of these distances must be the minimum one.

Running time:

Time taken to sort the numbers $= O(n \log n)$

Total time to compute distances $= O(n)$

Total time $= O(n \log n) + O(n)$

$= O(n \log n)$.

Q2: In the divide and conquer algorithm for the closest pair problem, we are required to sort the points acc. to their y-coordinates for the two subproblems generated when we divide the points into two equal sized sets acc. to their x-coordinates. In the beginning all the points are sorted acc. to their x-coordinates, list of which is maintained in $P(x)$ and acc. to y-coordinates, list of which is maintained in $P(y)$. Write an $O(n)$ algorithm to

sort the subsets of points during subsequent problems arising in solving the problem.

Ans: Initially we have P_x with all the points sorted according to x-coordinates and P_y with all the points sorted according to y-coordinates. We divide P_x into Q and R in such a way that Q_x contains first half of P_x and R_x contains second half of P_x and this division takes $O(1)$ time. Then we take the last point of Q_x and make $x^* = x$-coordinate. Now, to compute Q_y & R_y, we pass through P_y and compute $x \leq x^* \; \forall \; x \in P_x$ if x is less than or equal to x^*, point is in Q_y and if it is more than x^*, it is in R_y. For each point in P_y, this comparison takes $O(1)$ time & we have to do this for n points.

$\therefore$ Total time $= O(n)$

Also, since P_y is already sorted $\therefore$ Q_y and R_y are also sorted.

Q3: What happens if one doubles the table size when an item is inserted into a full table and halve the size when a deletion would cause a tabl to become less than half full? Determine a bad scenario for Table-insert and Table-Delete operations Can you think of a better steategy? Explain how the suggested steategy will improve the amortised cost of the operations.

The strategy in which one doubles the table siz when an item is inserted into a full table and halve the size when a deletion would cause the table to become less than full, would guarantee

that the load factor of the table never drops below 1/2, but unfortunately, it can cause the amortized cost of an operation to be quite large.

Consider the following scenario, we perform n operations on a table T, where n is an exact power of 2. The first $n/2$ operations are insertions, which by our previous analysis cost a total of $\Theta(n)$. At the end of this sequence of insertion, $T.num = T.size = n/2$. For the second $n/2$ operations, we perform the following sequence: insert, delete, delete, insert, delete, delete, insert, insert....

The first insertion causes the table to expand to size n. The two following deletions cause the table to contract back to size $n/2$. Two further insertions cause another expansion and so forth. The cost of each expansion and contraction is $\Theta(n)$ and there are $\Theta(n)$ of them. Thus, the table cost of n operations is $\Theta(n^2)$, making the amortised cost of an operation $\Theta(n)$.

We can improve upon this strategy by allowing the load factor of the table to drop below 1/2. Specifically, we continue to double the table size upon inserting an item into a full table, but we halve the table size when deleting an item causes the table to become less than 1/4 full, rather than 1/2. The load factor of the table is therefore bound below by the constant 1/4.

Q4. We studied the following potential function to analyse the cost of table insertion and deletion for dynamic tables:

$$\phi(T) = \begin{cases} 2\,num[T] - size[T] & \text{if } \alpha(T) \geq 1/2 \\ size[T]/2 - num[T] & \text{if } \alpha(T) < 1/2 \end{cases}$$

Can the potential function $\phi'(T)$ below be used in place of $\phi(T)$?

$$\phi'(T) = \begin{cases} 2\,num[T] - 2\,size[T] & \text{if } \alpha(T) \geq 1/2 \\ size[T]/8 - num[T] & \text{if } \alpha(T) < 1/2 \end{cases}$$

Why or why not?

Ans: We know that we need a potential func^n such that potential grows to T.num when load factor is either 1 or 1/4 so that we have enough potential to pay for expansion and contraction. Also when $\alpha = 1/2$ potential should be 0. Suppose that current size of table is $n/2$ and it is full i.e. $\alpha = 1$. So potential acc. to the given potential function is

$$\phi = 2\,num[T] - 2\,size[T]$$
$$= 2\left(n/2\right) = 2\left(n/2\right) = 0$$

so if we insert an item to the table causing it to expand we won't have potential to pay for the expansion. So this potential func^n can't be used. Also suppose the current size is n and table is 1/4 full i.e. $\alpha = \frac{1}{4}$. So potential according to the given potential func^n is

$$\phi = size[T]/2 - num[T]$$
$$= n/2 - n/4$$
$$= -n/4$$

the potential is -ve. If we delete an item, which causes table to contract, we won't have enough potential to pay for the contraction. ∴ The given potential func^n can't be used.

Q5. Execute the following operations on an initially empty fibonacci heap:

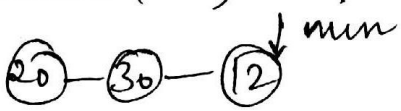
insert(12); insert(20); insert(30), insert(10), insert(25), insert(28), insert(8), insert(26), insert(35), Insert(40), insert(5), insert(50), deleteMin(), decreasekey(26,7), decreasekey(30,6), deleteMin(). For all intermediate steps, illustrate the resulting fibonacci heap. New elements should always be inserted to the left of the current minimum. The consolidation operation after deleteMin() starts with the next element on the right hand size of the deleted minimum.

Ans:

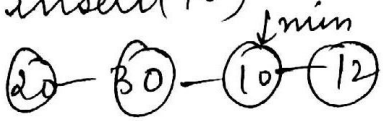
insert(12)

insert(20)

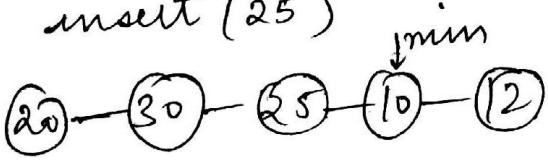
insert(30)



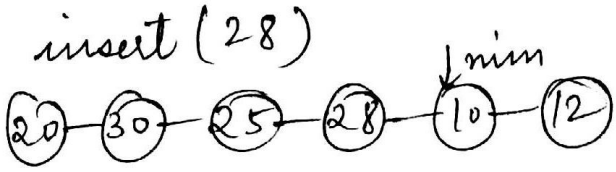
insert(10)



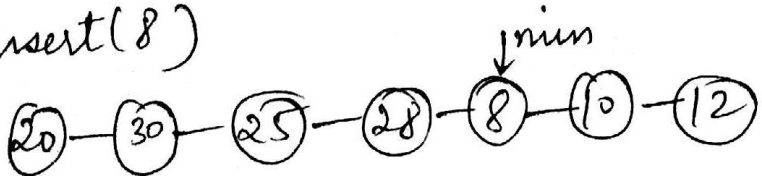
insert(25)



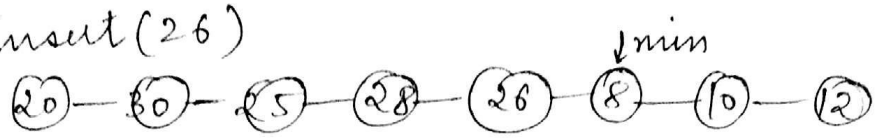
insert(28)



insert(8)

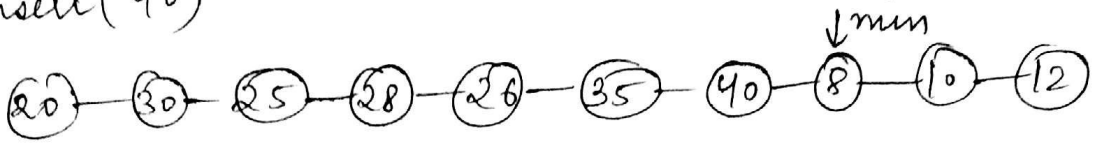


insert(26)

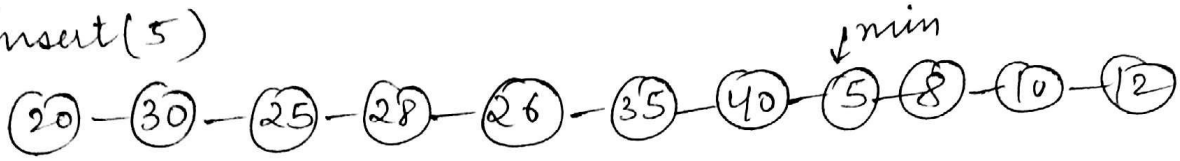


insert (35)

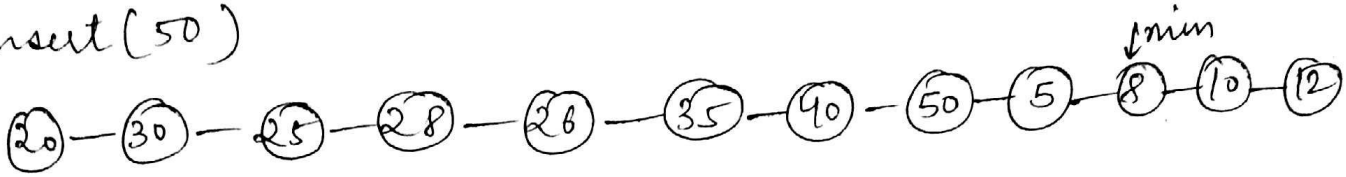
insert(40)



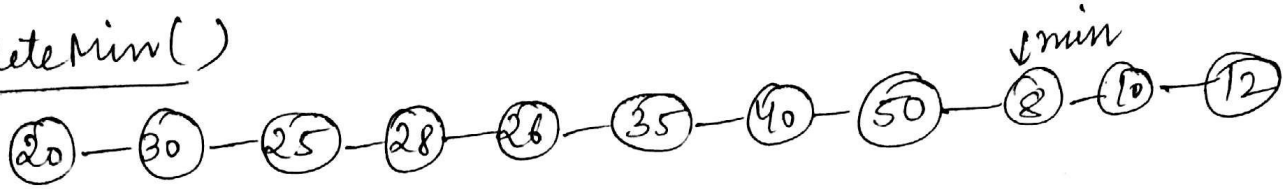
insert(5)



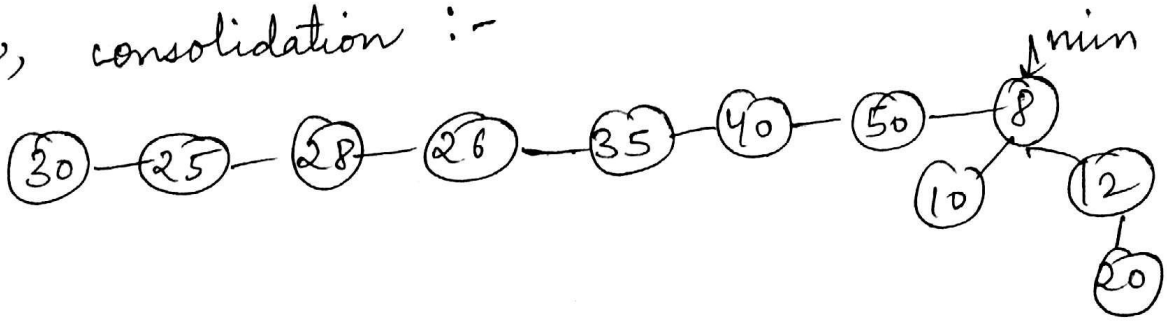
insert(50)

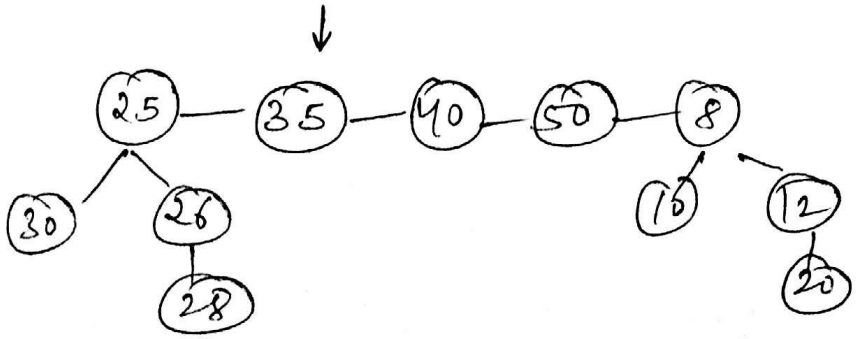


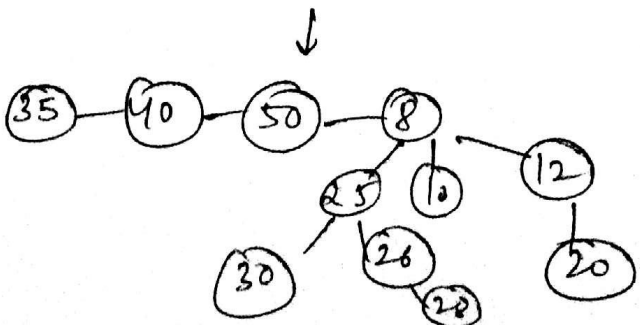
deleteMin()

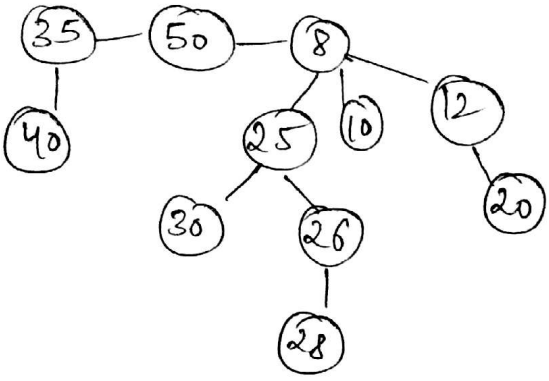


Now, consolidation :-

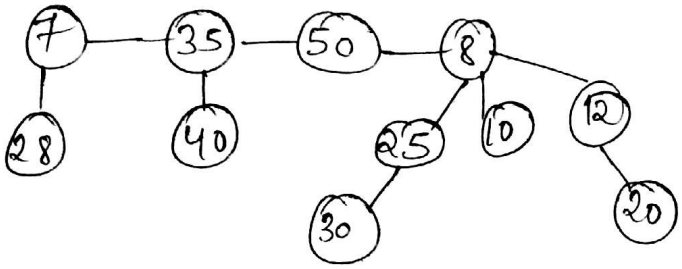




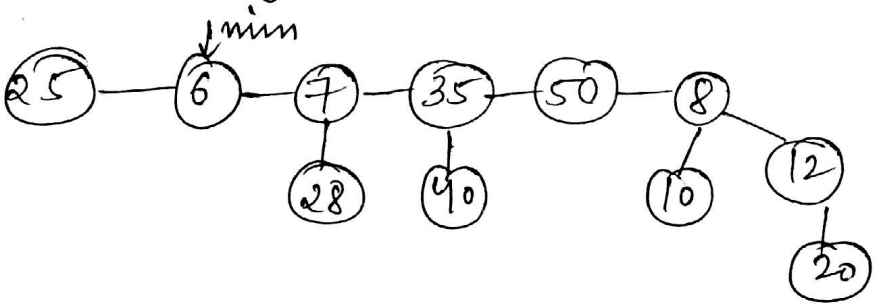




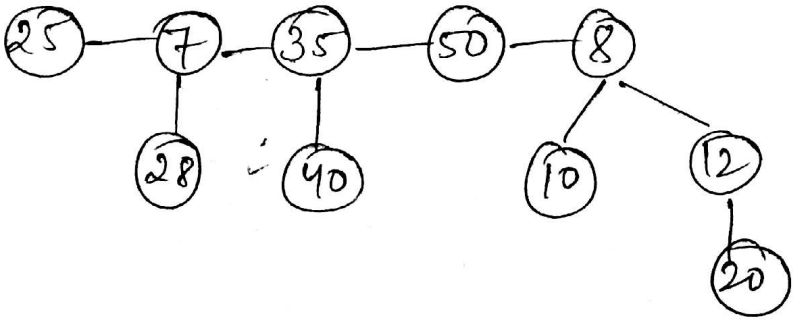
decreasekey (26,7)



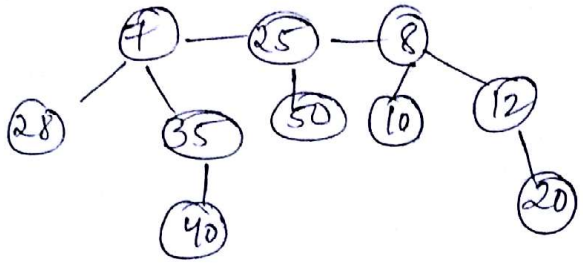
decreasekey (30,6)

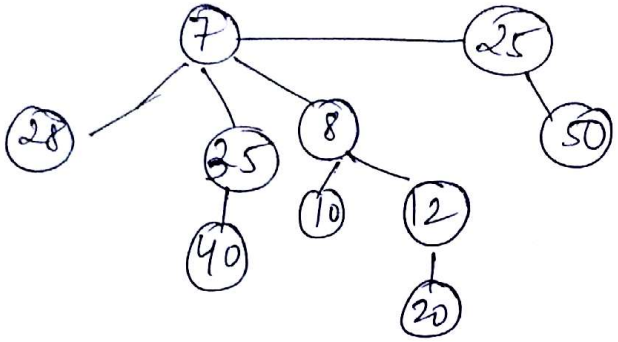


deleteMin ()



Consolidation





Q8. What is the maximal length of a code word possible in Huffman encoding of an alphabet of n characters?

Ans: Maximal length of a code word is the sum, over all the letters $x \in S$, of the number of times x occurs time the length of the bit string $y(x)$ used to encode x.

$$= \sum_{x \in S} n f_x |y(x)| = n \sum_{x \in S} f_x |y(x)|$$

Q9. Indicate whether each of the following properties are true for every Huffman code.

(a) The codewords of two least frequency characters have the same length.

Ans: True, consider a leaf v in T^* whose depth is as long as possible. leaf v has a parent u and

. we know that T^* is a full binary tree, so u must have another child. So w and v are siblings. Now, we show that w is a leaf. If w were not a leaf, there would be some leaf w' in the subtree below it. But then w' would have a depth greater than that of v, contradicting our assumption that v is a leaf of maximum depth in T^*. So v and w are sibling leaves as deep as possible in T^*. Thus our algo will get to the level containing v and w first. The leaves v and w are siblings and will get the two lowest frequency letters of all ∴ The code length of two lowest frequency letters will be same as these letters are labeled at sibling leaves v and w.

(b) The codeword's length of a more frequent character is always smaller than or equal to the codeword's length of a less frequent one.

Ans: True, suppose that u and v are leaves in T^* such that depth(u) < depth(v). Suppose T^* gives optimal prefix code, leaf u is labeled with $y \in S$ & leaf v is labeled with $z \in S$. Then $f_y \geq f_z$. Suppose $f_y < f_z$, consider code obtained by exchanging labels at node u and v. On ABL, the effect of change is as follows: the multiplier on f_y increases (from depth(u) to depth(v)) and the multiplier on f_z decrease by the same amount (from depth(v) to depth(u).

$$ABL = \sum_{x \in S} f_x \, depth_x - (depth(u) f_y + depth(v) f_z) + (depth(u) f_z + depth(v) f_y)$$

$$= \sum_{x \in S} f_x \, depth_x - u f_y - v f_z + u f_z + v f_y$$

$$= \sum_{x \in S} f_x \, depth_x + u (f_z - f_y) - v (f_z - f_y)$$

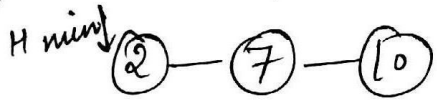
$$= \sum_{x \in S} f_x \, depth_x + (u - v)(f_z - f_y)$$

$$f = \sum_{x \in S} f_x \, depth_x + (v - u)(f_y - f_z)$$

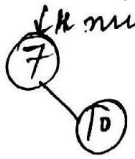
If $f_y < f_z$, this change is a -ve no., contradicting the supposed optimality of the prefix code that we had before the exchange.

Q6. Show that the following claim is not true: The maximum height of a tree within a fibonacci heap with n nodes is $O(\log n)$.

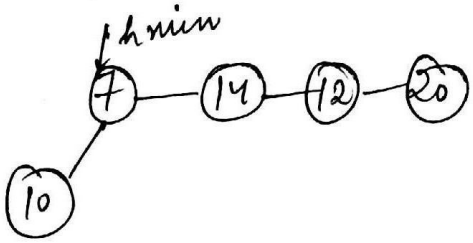
1) Insert 2, 7, 10



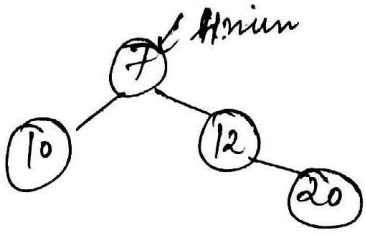
2) Extract min



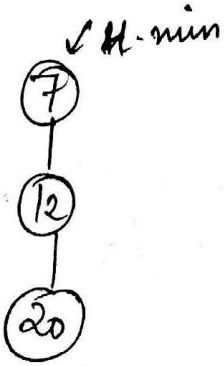
3) Insert 14, 12, 20



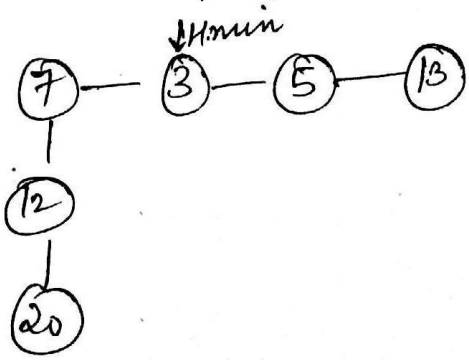
4) delete 14



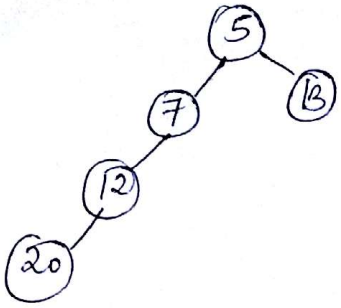
5) delete 10



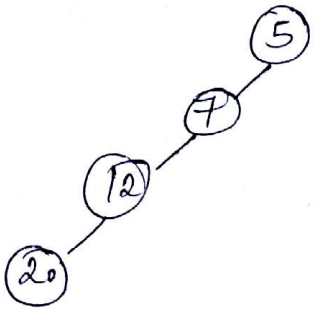
6) insert 3, 5, 13



7) Extract min



8) Delete 13



In step 8, we obtained a fibonacci heap consisting of 1 tree which is a linear chain of n nodes where $n = 4$.

Height of the tree = 3

i.e. $\text{height} > \log_2 4$

$\text{height} > 2$

$\Rightarrow \text{height} > \log n$

∴ max. height is not $O(\log n)$.

Q7 Suppose algorithm A requires $O(n^2)$ decrease-key operations and $O(n)$ delete-min operations; all remaining steps takes $O(n)$ time. Suppose the decrease key and delete min operations are implemented using fibonacci heaps. True or false: Algorithm A takes $O(n^2)$ amortized time. Explain.

Ans: In algorithm A,

1. time required for a decrease key operation using fibonacci heap $= O(\log n)$ i.e $O(n)$.

Total no. of decrease key operations $= O(n^2)$

Total time taken for $O(n^2)$ decrease key operations

$= O(n^2)$ —— (1)

2. Time required for a delete min operation using fibonacci heap =

In algorithm A

1. Amortized cost for a decrease key operation using fibonacci heap = $O(1)$
Total no. of decrease key operations = $O(n^2)$
∴ Total amortized cost for $O(n^2)$ decrease key operations = $O(n^2)$ — (1)

2. Amortized cost for a delete-min operation using fibonacci heap = $O(\log n)$
Total no. of delete min operations = $O(n)$
∴ Total amortized cost for $O(n)$ delete min operations $= O(n \lg n)$ — (2)

3. Cost for remaining steps = $O(n)$

∴ Total time taken by all operations in algorithm A

$$O(n^2) + O(n \lg n) + O(n) = O(n^2) \quad (\text{from } ①, ②, ③)$$

$$\text{Amortized cost} = \frac{O(n^2)}{n} = O(n)$$

∴ false, Algorithm A doesn't takes $O(n^2)$ amortized cost

Q9. If the symbols are sorted by frequency, can we say that algorithm for the generation of Huffman code be implemented in linear time? Explain.

Ans. Yes, the algorithm for the Huffman code generation can be implemented in linear time i.e. $O(n)$ if symbols are already sorted by frequency and stored in array, say P (ascending frequency).

Total no. of iterations on $P = n$

In each iteration :—

1) 2 symbols with lowest frequencies from top are extracted. Time taken = $O(1)$.

2) Time taken to merge 2 symbols into a single letter with frequency of equal to combined frequency of both = $O(1)$

∴ Total time in each iteration = $O(1)$

Total time taken in n iterations of scanning array $P = O(n)$ i.e. linear time.