ASSIGNMENT NO: 10 DATE: 12/12/2023

## Problem Statement:

(a) Devise a generalised algorithm to insert a sequence of keys one after another into a B-tree of order m.

(b) Devise a generalised algorithm to delete a sequence of keys one after another from a B-tree of order m.

## Brief Description:

B-Tree is a specialized m-way tree that can be widely used for disk access. A B-tree of order m can have atmost m-1 keys and m children. One of the main reasons of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small. In a B-tree of order m, every node contains atmost m children; Every node except the root node contains atleast m/2 children; All the leaf nodes must be on the same level.

While performing operations like insertion or deletion in a B-tree, any property of B-tree may violate such as number of minimum children a node can have. To maintain the properties of B tree, the tree may split or join. Searching in B trees are similar to that in Binary search trees.

Example: m = 4;

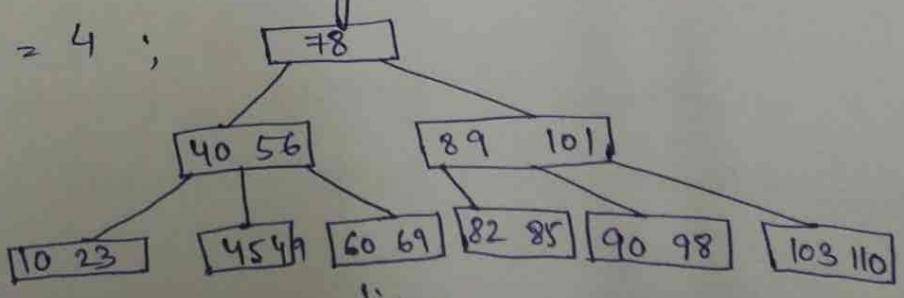



figure.

## Algorithm in Text:

To devise a generalized algorithm for inserting and deleting a sequence of keys into / from a B-tree of order m, we need to peform the following:

### For insertion:

Start at the root of the B-tree. Traverse down the tree to find appropriate leaf node for insertion. If the leaf node has room for the new key, insert it there. If the leaf node is full, split it and promote the median key to the parent node. Repeat until the root is reached, splitting nodes as necessary. If the root is split, create a new root.

### For deletion:

Start at the root of the B-tree. Traverse down the tree to find the node containing the key to delete. If the key is found in a leaf node, remove it. If the key is not found: ① If the current node is a leaf, the key does not exist in the tree. ② If the current node is an internal node, recursively search in the appropriate child. After deletion, rebalance the tree as necessary by merging nodes or redistributing keys. Define a function to merge or redistribute keys between a node and its siblings if necessary. If a sibling has more than 'm' keys, borrow a key from it, otherwise merge the node with it's sibling.

## Algorithm At a glance:

**Input:** Order m and the keys for insertion.

**Output:** Inorder traversal after every insertion and deletion.

For insertion:

```
insert_sequence (root, keys):
    For key in keys:
        if root is full:
            create new_root
            set new_root as parent of root
            split_child (new_root, 0, root)
            root = new_root
        End if
        insert_non_full (root, key)
End for loop and function

insert_non_full (node, key):
    if node is a leaf:
        insert_key_into_leaf (node, key)
    else:
        find_child_index (node, key)
        if child is full:
            split_child (node, child_index, child)
            if key > node.keys [child_index]:
                child_index = child_index + 1
        insert_non_full (node.children [child_index], key)
End function

insert key_into_leaf (leaf, key):
    position = find_position_to_insert (leaf, key)
    // Shift keys to make space accordingly
    leaf.keys [position] = key
    leaf.n = leaf.n + 1
End function

split_child (parent, child_index, child):
    new_child = create_new_node()
    new_child.leaf = child.leaf
    new_child.n = m/2 - 1
```

```
For i = 0 to m/2 - 1 :
    new_child.keys[i] = child.keys[i + m/2]
    child.keys[i + m/2] = null

if not child.leaf :
    For i = 0 to m/2 :
        new_child.children[i] = child.children[i + m/2]
        child.children[i + m/2] = null

For i = parent.n down to child_index + 1 :
    parent.children[i+1] = parent.children[i]

parent.children[child_index+1] = new_child

For i = parent.n - 1 down to child_index :
    parent.keys[i+1] = parent.keys[i]

parent.keys[child_index] = child.keys[m/2]
child.keys[m/2] = null
child.n = m/2 - 1
parent.n = parent.n + 1
```

For deletion:

```
delete_key(root, key):
    if root is not empty :
        delete_from_node(root, key)
        if root has no keys left:
            if root is not leaf :
                set new root as root's only child
            else :
                set root to null

End function
delete_from_node(node, key):
    if key is found in node :
        if node is a leaf :
            remove key from node
```

else:
    if child of node containing key has atleast m/2 keys:
        find predecessor or successor of key
        replace key with predecessor or successor
        recursively delete predecessor or successor from child
    else if sibling of child has more than m/2 keys:
        borrow a key from sibling
        replace key with borrowed key in node
        recursively delete key from child
    else:
        merge child with its sibling
        recursively delete key from merged child

else:
    if key is not in node:
        if node is a leaf:
            key not found in tree
        else:
            find appropriate child for key
            if the child has less than m/2 keys:
                fill the child by borrowing from its sibling or merging
            recursively delete key from appropriate child

Test Run:

Let order (m) = 9

Keys to be inserted: 10, 20, 30, 40, 50, 60, 70, 80, 90

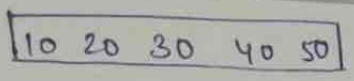
10: [10]

20: [10,20]

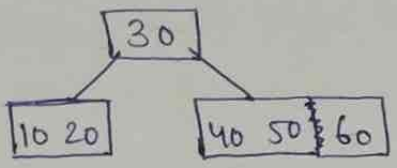
30: [10 20 30]

40: [10 20 30 40]

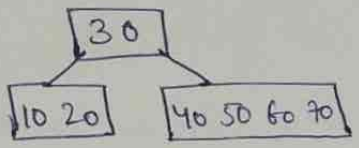
50 :



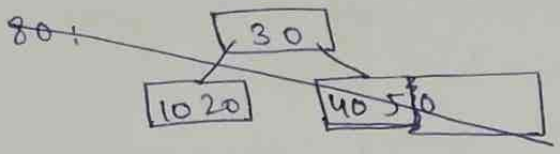

60 :



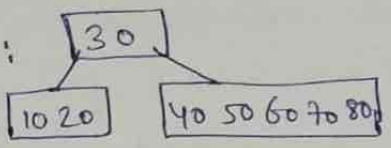

70 :



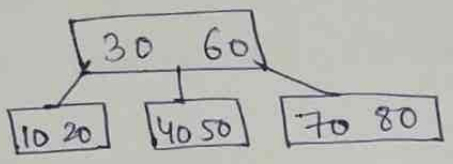

80 :



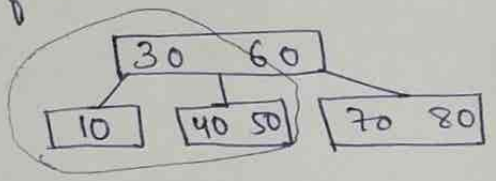

80 :



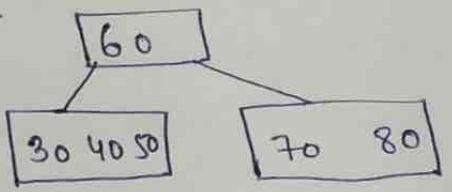

90 :

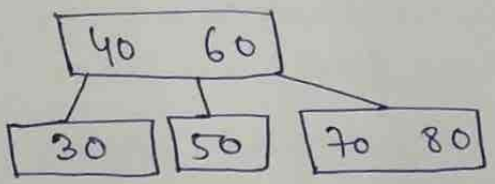



Deletion of 20 :




Deletion of 10 :




As $m = 3$, minimum no: of keys $= \lceil 3/2 \rceil - 1$

$= 1$

maximum number of keys $= 3 - 1$

$= 2$.

## Data Structures Used:

A structure data type is used to represent a node in the B-tree. It contains the following:

n → number of keys currently stored in the node

keys → An array to store the keys

children → An array to store pointers to child nodes

leaf → a flag indicating whether the node is a leaf node (1 indicates leaf, 0 otherwise).

A root node pointer to point to the root node and access the entire tree structure.

## Computational Complexities:

The time complexity for a B-tree to perform insertion and deletion of n elements is O(log n).

## Discussions:

(i) All leaves are at the same level.

(ii) B-trees are particularly efficient for large datasets, as they can store a large number of keys in each node.

(iii) B-trees require additional complexity in terms of maintenance operations, such as node splitting and merging, to ensure balance is maintained.

ASSIGNMENT NO: 11 DATE: 20/12/2023

**Problem Statement**: Implement Heap Sort.

**Brief Description:**

Heap Sort is a comparison based sorting technique based on binary heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

A binary heap is a complete binary tree where items are stored in a special order such that value in a parent node is greater or smaller than the values in its two child nodes depending on whether the heap structure follows max-heap or min-heap. The heap can be represented by an array or a binary tree.

Heap Sort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into sorted part of the list.

**Algorithm in text:**

Heapify procedure can be applied to a node only if its children nodes are heapified. So, the heapifying process must be performed in the bottom-up order.

First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until the size of the heap is greater than 1.

The sorted array is obtained by reversing the order of the elements in the input array.

## Algorithm at a glance:

**Input:** The total number of elements and the data values.

**Output:** The sorted binary tree (traversals).

## Heap sort algorithm for sorting in increasing order:

Step 1: Build a max heap from the input data.

Step 2: At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of the heap by 1. Finally heapify the root of the tree.

Step 3: Repeat step 2 while size of the heap is greater than 1.

Step 4: End.

## Heap Sort algorithm for sorting in decreasing order:

Step 1: Build a min heap from the input data.

Step 2: At this point, the smallest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of the heap by 1. Finally heapify the root of the tree.

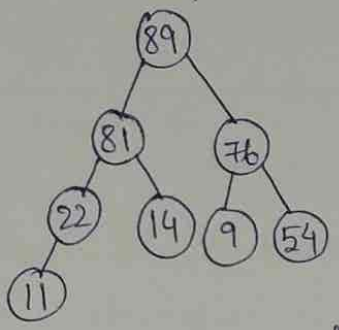
Step 3: Repeat step 3 while size of the heap is greater than 1.

Step 4: End.

## Test Run:

Suppose the array that is to be sorted contains the following elements: 81, 89, 9, 11, 14, 76, 54, 22

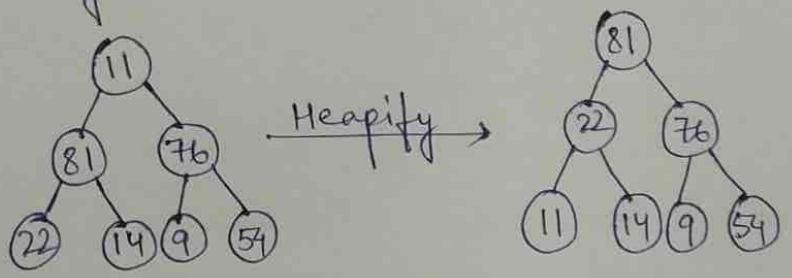
First, we have to build a heap. Here, we are creating a max heap.




After creating the max heap, the array elements are:

| 89 | 81 | 76 | 22 | 14 | 9 | 54 | 11 |
|---|---|---|---|---|---|---|---|

Now, the root 89 is removed to the last location by exchanging it with 11. Finally 89 is eliminated from the heap by reducing the maximum index value of the array by 1. The balanced elements are then rearranged into the heap. The heap and the array looks like:

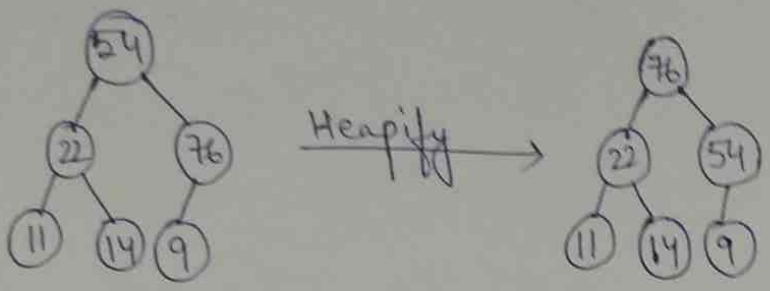



So, the array will be:

| 81 | 22 | 76 | 11 | 14 | 9 | 54 |
|---|---|---|---|---|---|---|

Next, the root 81 is removed to the second last location by exchanging with 54. Finally, 81 is removed from the heap by reducing the maximum index value of the array by 1. The balanced elements are then rearranged into the heap. The heap and the array

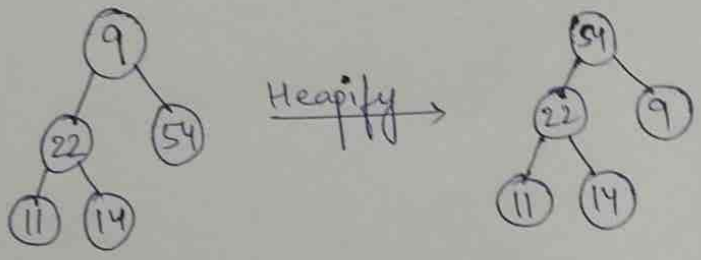
looks like:




So, the array will be:

| 76 | 22 | 54 | 11 | 14 | 9 |
|---|---|---|---|---|---|

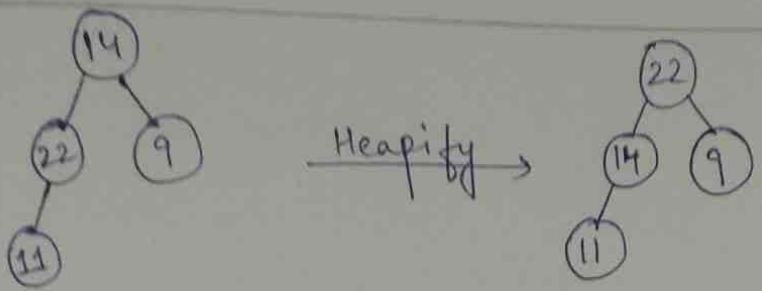
Next, the root 76 is removed to the third last position by exchanging it with 9. Finally 76 is eliminated from the heap and the maximum index value of the array is reduced by 1. The balanced elements are then rearranged into the heap. The heap and the array looks like:




So, the array will be:

| 54 | 22 | 9 | 11 | 14 |
|---|---|---|---|---|

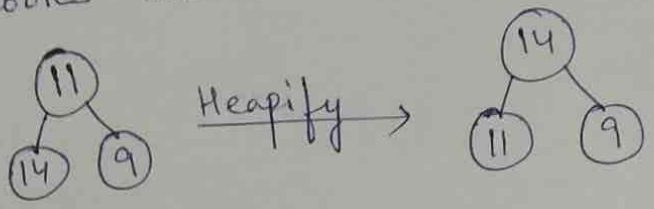
Next, the root 54 is removed to the fourth last position by exchanging it with 14. Finally 54 is removed from the heap and the maximum index of the array is reduced by 1. The balanced elements are then rearranged into the heap. The heap and the array looks like:




So, the array will be:

| 22 | 14 | 9 | 11 |
|---|---|---|---|

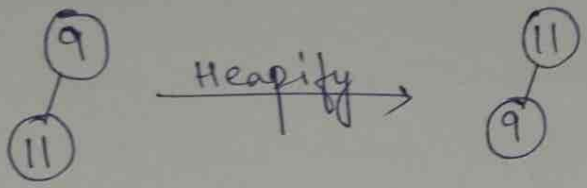
Next, the root 22 is removed to the fifth last position by exchanging it with 11. Finally 22 is eliminated from the heap and the ~~size of array~~ maximum index of array is reduced by 1. The balanced elements are then rearranged into the heap. The heap and the array looks like:




So, the array will be:

| 14 | 11 | 9 |
|---|---|---|

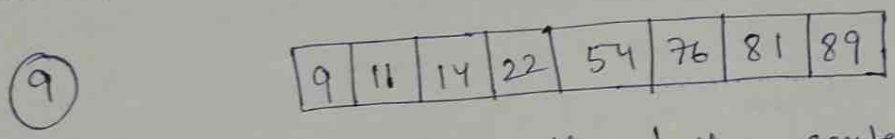
Next, the root 14 will be exchanged with 9 at the third position. Finally 14 is eliminated from the heap and the maximum index of the array is reduced by 1. The balanced elements are then rearranged into the heap. The heap and the array thus becomes:




So, the array will be:

| 11 | 9 |
|---|---|

Next, the root 11 is removed and exchanged with 9 at the second position. Finally 11 is removed from the heap and the maximum index of the array is reduced by 1. The balanced elements are now rearranged into the heap. The heap and the array becomes:




The array above shows the fully sorted array using the heap, thus called the heap sort.

## Data structure Used:

Here, an array is used for arranging a list of elements in order. Heap sort algorithm uses one of the tree concepts called the heap tree. Priority queues are implemented with a heap, a tree like data structure, also used in the heap sort algorithm.

## Computational Complexities:

The worst case time complexity of heap sort is $O(n \log n)$.
The best case time complexity of heap sort for distinct keys is $O(n \log n)$ or $O(n)$ for equal keys.
The average case time complexity of heap sort is $O(n \log n)$.
The worst case space complexity of heap sort is $O(n)$ total $O(1)$ auxiliary.

## Special Features:

(a) Heap sort is an inplace algorithm. It's typical implementation is not stable.

(b) Heap sort algorithm has limited uses because Quick sort and merge sort are better. But the heap data structure itself is hugely used.

(c) It's typical implementation is not stable.

(d) It's simpler to understand because it does not use advanced computer science concepts such as recursion.

ASSIGNMENT NO: 12

DATE: 20/12/2023

Problem Statement: Implementation of Shell Sort.

Brief Description:

Shell sort is the generalization of insertion sort, which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions. In insertion sort, at a time elements can be moved ahead by one position only. To move an element to a far away position, many movements are required, that increases the algorithms execution time. This drawback is overcomed by shell sort as it allows the movement and swapping of far away elements as well.

In shell sort, instead of comparing only adjacent keys, if we compare, the keys far apart then the keys far apart gets sorted. Afterward, the items closer together would be sorted and finally the increment between keys is being compared will be reduced to one to ensure that the list is completely in order.

Shell sort can be used for sorting medium to large datasets. It is an in-place comparison sort.

Algorithm in Text:

Let us consider a one-dimensional array of size n, where n is the number of elements to be sorted using shell sort. The array arr[ ] contains the elements to be sorted at position 0, 1, . . ., n-1.

Let us consider another array gap[ ] of size 3 that contains the required gaps 5, 3 and 1 for sorting the elements.

For every iteration, a gap is chosen from the gap array starting from gap[0], gap[1] and gap[2]. Again for every gap the element at position 0 is checked with the element at a gap of gap[0] and the process continues and if the elements at the respective positions happens to be unordered the elements are swapped. In this way, the elements are first sorted with a gap 5, then with a gap 3 and then at a gap of 1. The final array of sorted elements is then displayed.

Algorithm at a glance:

Input: The number of data elements and the data values.

Output: The sorted array after performing sorting at respective gaps and the final sorted array of elements.

Step 1: Enter the number of elements and store it in n.

Step 2: For i = 0 to n-1 do
enter the elements and store at arr[i].

Step 3: Display the original array of elements.

Step 4: Initialize gap[3] array with values {5, 3, 1}.

Step 5: For g = 0 to 2 do
gap = gap[g]

Step 5(i): For i = gap to n-1 do
temp = arr[i] // temp is a temporary variable

Step 5(ii): For j = i to j >= gap && arr[j-gap] > temp

arr[j] = arr[j - gap];
j = j - gap
End of for loop at 5 (ii)
arr[j] = temp
End of for loop at 5 (i)
Display the currently sorted array with the respective gap
End of for loop at 5

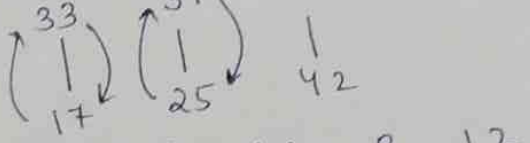
Step 6: Display the final sorted array:
For i = 0 to n-1 do
printf("%d", arr[i]);

Step 7: End.

Example Run:

(i) n = 8
Elements: 33, 31, 40, 8, 12, 17, 25, 42

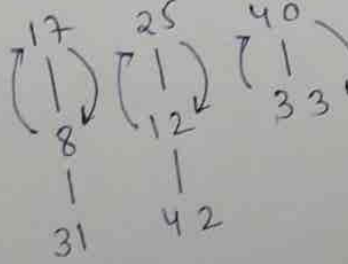
Sorting with gap 5:




17 25 40 8 12
33 31 42

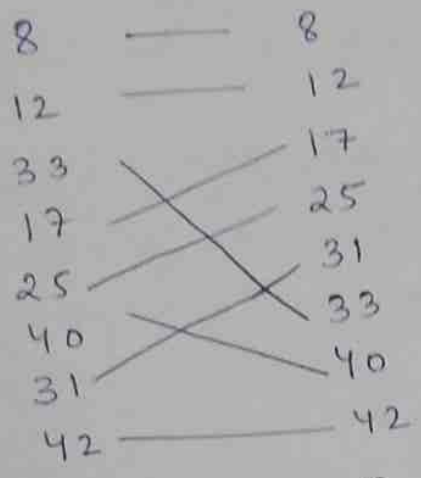
⇒ 17 25 40 8 12 33 31 42

sorting with gap 3:




⇒ 8 12 33 17 25 40 31 42

Sorting with gap 1:

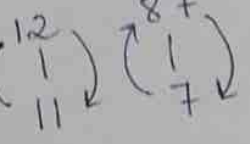



⇒ 8 12 17 25 31 33 40 42

(ii) n = 7

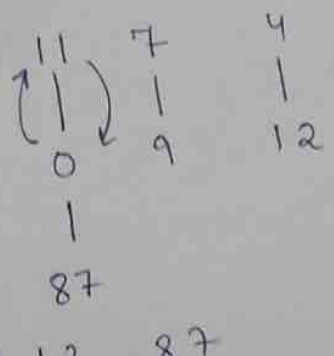
Elements: 12, 87, 4, 0, 9, 11, 7

Sorting with gap 5:



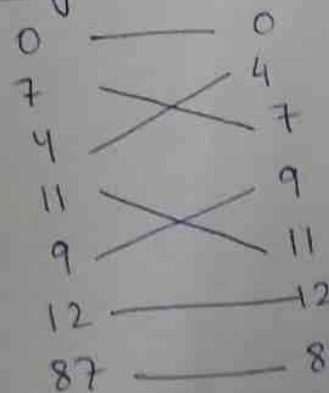

⇒ 11 7 4 0 9 12 87

Sorting with gap 3:




⇒ 0 7 4 11 9 12 87

Sorting with gap 1:




⇒ 0 4 7 9 11 12 87

**Data Structure Used:** Two one-dimensional arrays are used to store the elements to be sorted and the gaps respectively.

**Computational Complexities:**

Time complexity of shell sort is $O(n^2)$ in the worst case if the gap is reduced by half at every iteration where $n$ is the number of elements. The best case complexity is $\Omega(n \log n)$ if the given array is already sorted.

The space complexity of shell sort is $O(1)$.

**Special features:**

(i) There is no mandate for the choice of the gap. Many other choices other than 5, 3, 1 might work as well or better.

(ii) It might be wasteful to choose gaps as powers of 2 as the same keys compared in one pass would be compared again at the next.

(iii) If the increments of gap are chosen close together, it will make necessary to make more passes. If the increments decrease rapidly then fewer but longer pass will occur.

(iv) Shell sort algorithm is significantly slower than merge sort, quick sort and heap sort algorithms.

ASSIGNMENT NO: 13 DATE: 20/12/2023

PROBLEM STATEMENT: Implement Graph Sort.

Brief Description:

Let $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_n\}$ be the given sequence of n unsorted elements. The elements are numbered from 1 through n, from left to right of the list to label their positions in $\Pi$. The position i for some element $\Pi_i$ in $\Pi$ is important in comparing other elements in their respective positions with $\Pi_i$ in $\Pi$. This algorithm sorts the elements in $\Pi$ in non-decreasing order. In computing the RKPian graph, based on a given sequence elements, we do the following:-
For every element in $\Pi$, a vertex is introduced in the graph. Now the graph has n isolated vertices. To introduce edges between the vertices in the graph, for any pair of vertices $\Pi_i$ and $\Pi_j$, we introduce an edge only if $(i-j)(\Pi_i - \Pi_j)$ is non negative. If they are out of order, we do not introduce an edge between the corresponding vertices.

Example: $\Pi = \{5, 4, 7, 1, 2, 3, 6\}$. An edge $\{v_i, v_j\}$ is introduced from $v_i$ to $v_j$ if $\Pi_i \leq \Pi_j$ and $i < j$ in $\Pi$.

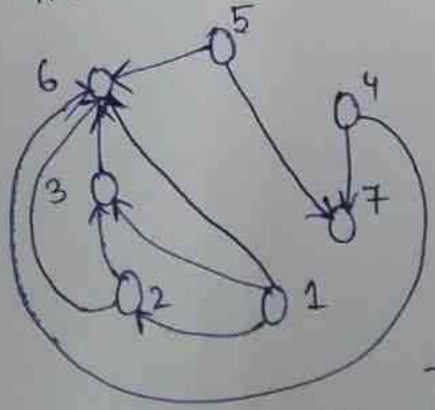



Figure:

## Algorithm in Text:

Let us consider a graph $G(V, E)$ where $|V|$ is the set of vertices. Here it is denoted by $n$ and $|E|$ is the set of edges. Three 1-Dimensional arrays, namely vertices[], adj[][] (it is a 2-Dimensional array to store the adjacency matrix) and indegree[] array to store the the vertices in an order of entry and the indegree of each vertex respectively. Let count keep a track of the count of the indegree of each vertex.

Initially the adjacency matrix is kept as all zeros. Construct the adjacency matrix based on the order of vertices such that, $(\Pi_i - \Pi_j)(i - j) >= 0$, then introduce an edge between the vertices $i$ and $j$, where $\Pi$ denotes the position of $i$ and $j$ respectively in the vertices array. (Here $0 \leq i, j < n$).

Now, calculate the indegree of each vertex and store it in indegree array. Determine the length of the indegree array. Iterate over the 'indegree' array and for each vertex, mark it's adjacent vertices in the adjacency matrix as $-1$ such that the highest positional element in the vertices array is deleted and displayed as the corresponding element of the sorted sequence. The edges associated with that vertex is are also displayed.

## Algorithm at a glance :

**Input :** A sequence of numbers, which are treated as the vertices of the graph.

**Output :** A sorted sequence of the above numbers in a non-decreasing order.

**Step 1 :** Initialize variables n, j, k, count, i and arrays vertices[], adj[][], indegree[]

**Step 2 :** Input the number of vertices and store it in n.

**Step 3 :**
```
For i = 0 to n-1 do
    For j = 0 to n-1 do
        adj[i][j] = 0
    End of for loop
End of for loop
```

**Step 4 :** Input the vertices in an order and store them in the vertices array.

**Step 5 :**
```
For i = 0 to n-1 do
    for j = i+1 to n-1 do
        if ((vertices[i] - vertices[j]) * (i-j)) >= 0 then
            adj[i][j] = 1
    End for loop
End for loop
```

**Step 6 :** Calculate the indegree of each vertex :
```
For i = 0 to n-1 do
    For j = 0 to n-1 do
        if adj[i][j] == 0 then
            count ++
        End if
    if count == n then
        indegree[k++] = ~~vertices[i]~~ i
~~End for loop~~
    count = 0
End of for loop
```

~~Reset count to 0.~~

~~End for loop~~

Step 7: Calculate length of the indegree array, store it in length-indegree.

Step 8:
```
i = length-indegree - 1
while i >= 0 do
    for j = n-1 to 0 do
        for K = 0 to n-1 do
            if vertices[j] == indegree[i] then
                adj[j][K] = -1
```

Step 9:
```
For j = 0 to n-1 do,
    adj[j][indegree[i]] = -1
    adj[indegree[i]][j] = 0
End for loop
```

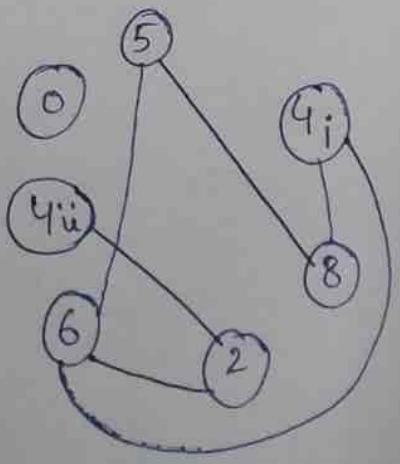
Step 10: Print vertices [indegree[i]] and indegree [i]
// To print the vertex along with it's respective position after being sorted.

Step 11: End.

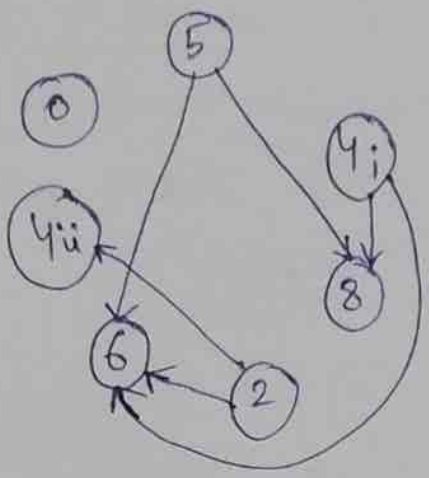
Test Run:

(1) No: of vertices = 7 ; Vertices: 5, 4, 8, 2, 6, 4, 0

step 1:



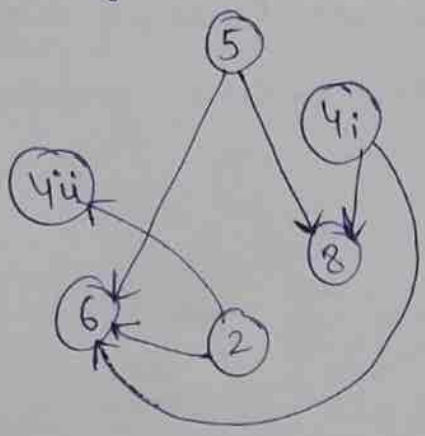
Step 2:




Step 3: Indegree 0 vertices: 5, 4i, 2, 0

Highest positional element: 0

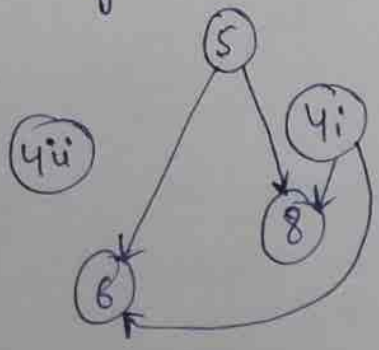
Removing vertex 0:




Step 4: Indegree 0 vertices: 5, 4i, 2

Highest positional element: 2

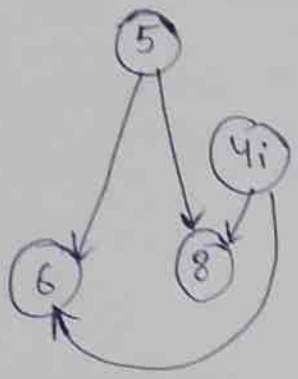
Removing vertex 2:



Step 5: Indegree 0 vertices : 5 , 4i, 4ii

Highest positional element : 4ii

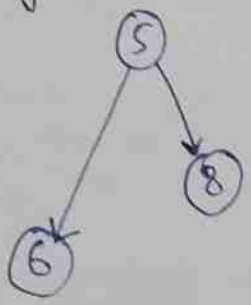
Removing vertex 4ii




Step 6: Indegree 0 vertices : 5 , 4i

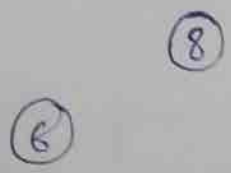
Highest positional element : 4i

Removing vertex 4i




Step 7: Indegree 0 vertices : 5

Removing vertex 5




Step 8: Indegree 0 vertices: 8 , 6

Highest positional element : 6

Removing vertex 6

Step 9: ~~Img~~ Indegree 0 vertices: 8
Remove vertex 8

Sorted ~~elems~~ sequence of elements in non decreasing order: 0, 2, 4ii, 4i, 5, 6, 8

Data Structures Used:

We use ~~three~~ two one dimensional arrays to store the vertices and indegree values of zero. One 2-Dimensional array is used to represent the adjacency matrix of the graph.

Computational Complexities:

The worst case time complexity of Graph Sort is $O(n^2)$ where n is the number of vertices. The space complexity also becomes $O(n^2)$ as the space required to store the adjacency matrix is proportional to the square of the number of vertices. In best case the time complexity becomes (n), when a list of n elements are given in a sorted order.

Discussions:

1) This code efficiently generates the adjacency matrix allowing for quick visualization for the graph's structure.
2) For sparse graphs, using an adjacency matrix might lead to inefficient memory usage.

ASSIGNMENT NO: 14 DATE: 20/12/2023

Problem Statement : Implementation of Sieve Sort.

Brief Description :

If we consider $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_n\}$ as a given sequence of n unsorted elements; the elements are numbered through 1 to n from left to right to label their positions in $\Pi$. The position i for some element in $\Pi_i$ in $\Pi$ is important in comparing other elements in their respective positions with $\Pi_i$ in $\Pi$.

The sorting algorithm Sieve Sort sorts the elements in $\Pi$ in non-decreasing order. In this algorithm, we divide the elements in $\Pi$ and enlist them into P subsequences, $1 \leq p \leq n$; where the value of each of the elements in a subsequence is smaller than that of the first element of the subsequence. Moreover, the values of the first elements in successive subsequences are arranged in non-decreasing fashion. Formally, for a list of n elements in $\Pi$, we assume a header list H[] of size n, where H[1] contains the first element in $\Pi$, i.e., $\Pi_1$. We append $\Pi_2$ after $\Pi_1$ only if the value of $\Pi_2$ is less than the value of $\Pi_1$. Otherwise we update the header list where H[2] that contains $\Pi_2$'s value. For the third element $\Pi_3$, if the value of $\Pi_3$ is less than $\Pi_1$ we append $\Pi_3$ in the list of $\Pi_1$. If value of $\Pi_3$ is greater than $\Pi_1$, we check it's value with $\Pi_2$. If $\Pi_3$'s value is less than that of $\Pi_2$, it is appended with $\Pi_2$'s list otherwise forms a new subsequence. This follows until all the elements are sorted in non-decreasing fashion.

## Algorithm in Text:

We first consider an 1-Dimensional array to hold the 'n' elements, where n is the number of elements and a 2-Dimensional array to store the sublists during sorting.

For performing SieveSort on n elements, the elements are numbered from 1 through n from left to right of the list. This algorithm sorts the elements in non decreasing order. We Divide the elements and enlist them into p subsequences, where the value of each of the element in the subsequence is that of the first element ~~etc~~ of the subsequence. Moreover, the values of the first element of the subsequence successive to each other are arranged in non-decreasing ~~funct~~ fashion.

The algorithm uses sieve sort approach to divide the elements list into sublists and then merge them in sorted order. The sorting direction alternates in each iteration until the array is fully sorted.

## Algorithm at a glance:

**Input:** A sequence of n numbers.

**Output:** A sorted sequence of the n numbers in non-decreasing order.

```
Procedure Sieve (B, A, direction)
/* B is the array of sublists and A is the original
                    array of n elements */

//Initialization
  subtree ← 1
  flag ← 1
  i ← 0
```

```
// Fill sublists
For k = 0 to n-1 do
    j = 0
    f = 0
    while i < subtree and f != 1 do
        if B[i][0] > A[k] then
            while B[i][j] != ' ' do
                j = j+1
            [end while]
            B[i][j] = A[k]
            f = 1
        [end if]
        i = i+1
    [end while]
    if f = 0 then
        subtree = subtree + 1
        B[i][0] = A[k]
    [end if]
[end for]
// Print and Check sorted
for i = 0 to subtree - 1 do
    k = 0
    while B[i][k] != ' ' do
        k = k+1
    [end while]
    For j = k-1 down to 0 do
        print B[i][j]
    [end for]
    Print \n
[End for]
PrintArray (A)
```

```
// Check if sorted
  For i = 0 to n-2 do
      if A[i] > A[i+1] then
          flag = 0
          break from loop
      [End if]
  [End for]
  return flag
End procedure
Algorithm sievesort (array A)
  // Create an array of length n and an array B of n
     empty sublists.
  x = 0
  //Read the elements of A.
  while x != 1 do
      iteration = 1
      while true do
       clearSublists (B)
       if iteration is odd then
           x = Sieve (B, A, 0)
       else
           x = Sieve (B, A, 1)
      [end if]
       if x = 1 then
           break from loop
      [end if]
       iteration ++
  [end while]
  // Print the final sorted elements.
```

## Test Run:

n = 9

Elements: 7 8 5 6 1 9 4 3 2

~~1st~~ Iteration 1: 7 8 5 6 1 9 4 3 2 →

7 5 6 1 4 3 2

8

9

Iteration 2: 7 5 6 1 4 3 2 ←

8

9

⇒ 1 2

3

4

5 6

5

7

8

9

Iteration 3: 1 2 →

3

4

5 6 →

5

7

8

9

Iteration 4:

1

2

3

4

5

6

7

8

9

Sorted sequence: 1 2 3 4 5 6 7 8 9

Data Structures Used:

An one dimensional array to store n elements and a 2-Dimensional array to store the subsequences.

Computational Complexities:

For n elements, the worst case for sorting the list in non-decreasing order is $O(n^2)$ and the best case time complexity is $O(n)$; if the elements are arranged in some fashion.

Discussion:

(i) This is a comparison based sorting algorithm that uses the Divide and Conquer principle. It sorts elements by dividing them into sub lists and merging them in a sorted manner.

(ii) It is not a stable sorting algorithm. It is a multiway Quicksort, which can be used to identify a desired key without sorting the entire sequence.

(iii) The choice of using a two dimensional array to construct the sublists might lead to higher memory usage and potentially slower performance for large inputs.

ASSIGNMENT NO: 15 DATE: 17/01/2024

Problem Statement:

Perform the operation of "Binomial-Heap-Extract-Min" for a Binomial heap that comprises atleast six Binomial trees, where the minimum key (value) is present in a node that is belonging to the largest tree or the next to the largest tree.

Brief Description:

A binomial heap is a type of heap data structure that consists of binomial trees. A binomial tree is a tree that follows a particular structural property: each node has a parent-child relationship with atmost one other node, and the children are ordered according to their degrees in increasing order from left to right.

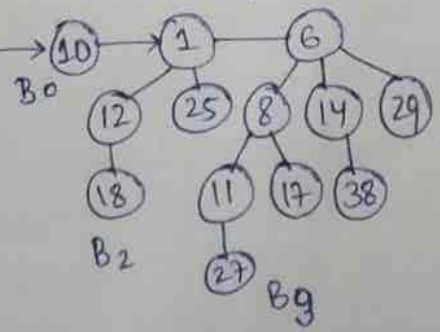
In a binomial heap, the following rules must be followed:-

(i) Each binomial tree in the heap must be a binomial tree of distinct degrees.

(ii) The root of each binomial tree must be containing the minimum element within that tree.

(iii) The degree of any node in the binomial heap is atmost $\log_2 n+1$, where n is the number of elements in the heap.

Example: Heap[H] →




n = 13 (1101)
$B_0 \rightarrow$ 1 node
$B_2 \rightarrow$ 4 nodes
$B_3 \rightarrow$ 8 nodes

Binomial heaps support efficient operations such as insertion, deletion, union and extraction of the minimum element.

The process of extracting the minimum element from a Binomial heap involves several steps :-

(i) Finding the root node containing the minimum element.
(ii) Removing this root node from the binomial heap.
(iii) Combining the children of the removed root node to form a new binomial heap.
(iv) Perform the union of the resulting binomial heap with the original heap and adjust the structure to maintain the binomial heap properties.

Algorithm in Text :

To extract the minimum element from the binomial heap we first create the binomial heap by merging two binomial trees of the same degree, by maintaining the heap structure. Then find the minimum element from the heap and extract the minimum node. Rearrange the heap structure and merge ~~the~~ two Binomial heaps into a single heap while ensuring that the properties of a Binomial heap are preserved.

The algorithm outlines the steps involved in extracting the minimum node from a Binomial heap. It iterates through the heap to find the minimum node, removes it from the heap, reverses the order of its children, and then unions the heap without the minimum node with the

reversed children to maintain the Binomial heap properties. Finally deallocate the memory of the minimum node and returns the updated heap.

Algorithm at a glance:

Input: The number of nodes in the Binomial heap and 'n' integers representing the keys of the nodes in the Binomial heap.

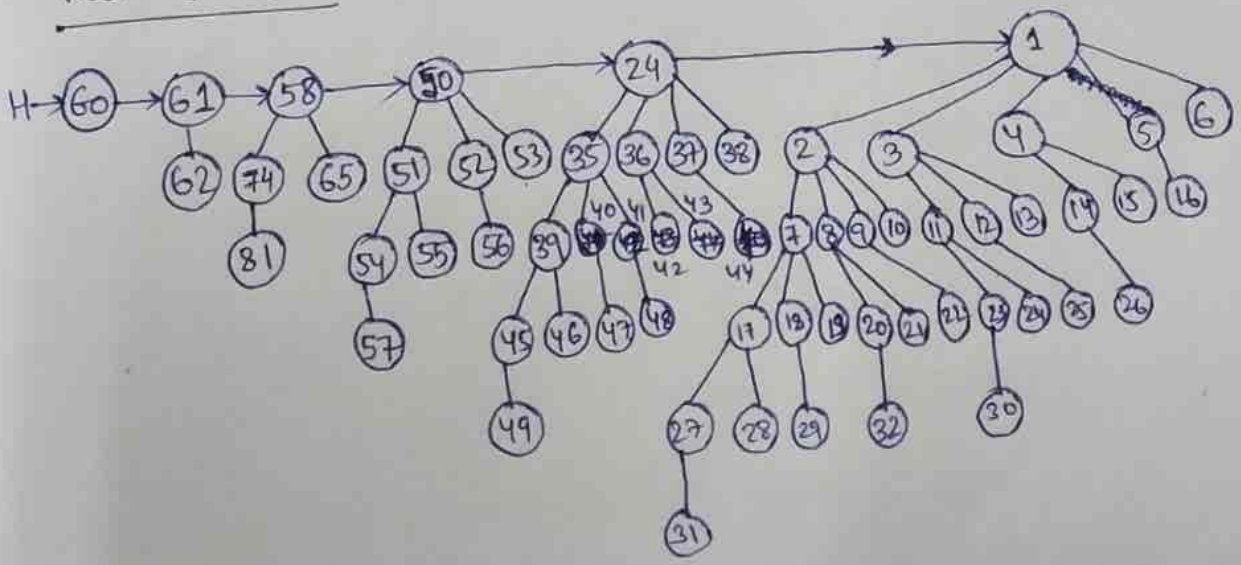
Output: The Binomial heap before and after extraction of the minimum element.

```
Binomial-Heap-Extract-Min:
    if heap is empty, then
        return NULL
    End if
    min_node = findMinimum(heap) /*finds the node
                                   with minimum key value*/

    prev = NULL
    curr = heap
    // Traverse the heap to find the parent of the
    minimum node
    while curr is not equal to min_node do:
        prev = curr
        curr = curr → sibling
    //Remove the minimum node from the heap
    if prev is NULL then,
        heap = heap → sibling
    else:
        prev → sibling = curr → sibling
```

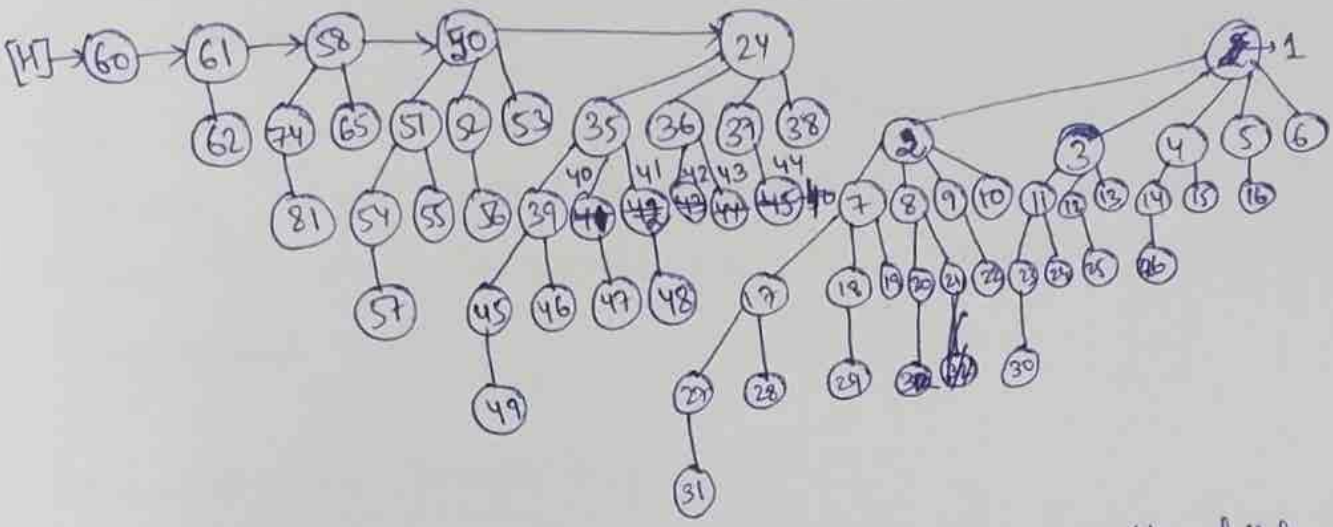
// Reverse the order of the children of the minimum node

```
rev_child = NULL
child = min_node → child
next = NULL
while child is not NULL, do:
    next = child → sibling
    child → sibling = rev_child
    child → parent = NULL
    rev_child = child
    child = next
[End while]
// Union the heap without the minimum node with the reversed children
rev_heap = rev_child
heap = unionBinomialHeap (heap, rev_heap)

free(min_node)
return heap
```

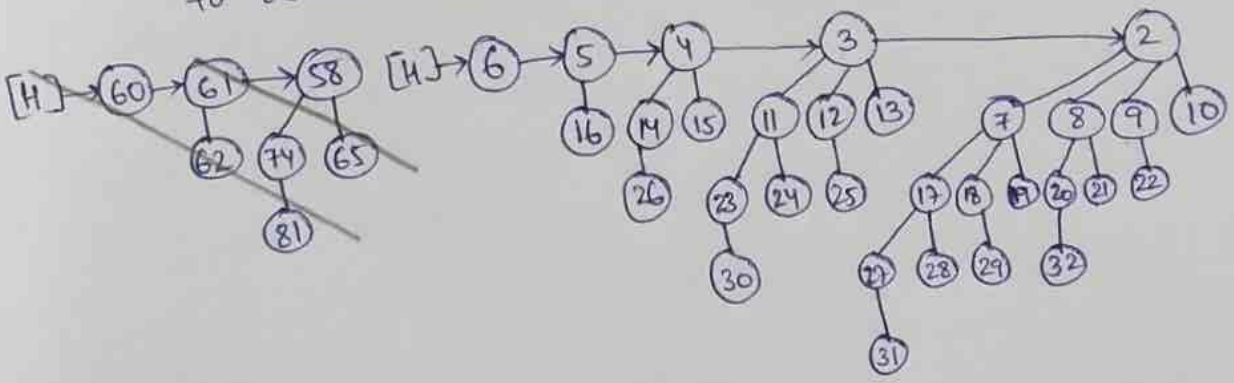
Test Run:



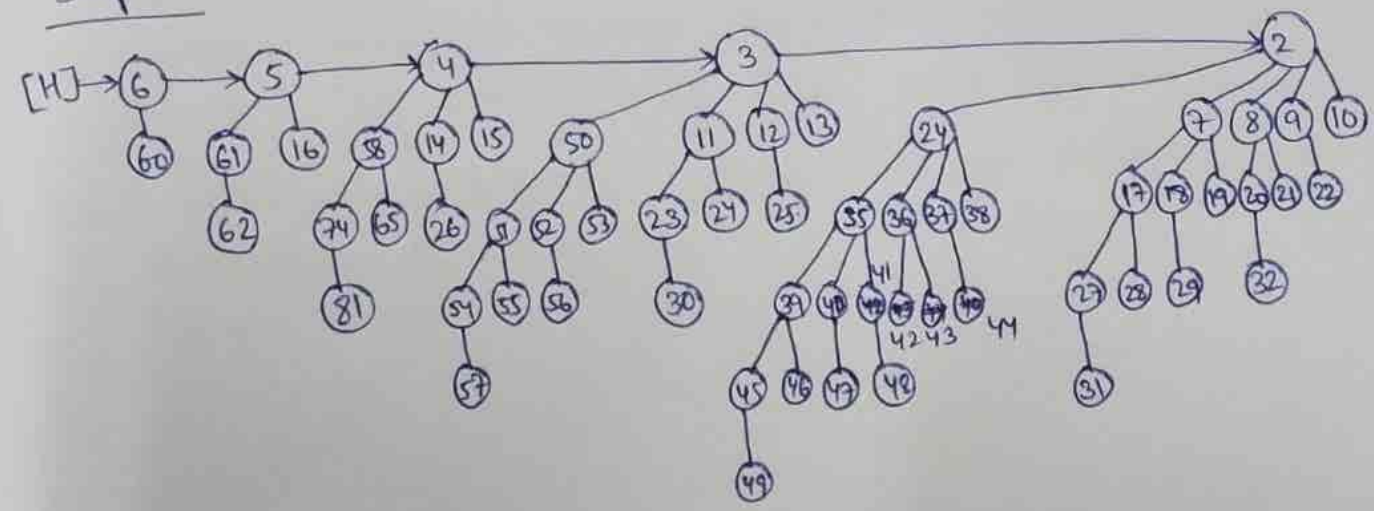

Step 1: Minimum key = 1, separate link from 1



Step 2: Remove 1 from Head list and reverse it's children to obtain new binomial heap H'.



Step 3: Unite H and H' and obtain final result.



Total number of nodes = 62

= 111110

$\Rightarrow 1-B_5, 1-B_4,$

$1-B_3, 1-B_2, 1-B_1$

## Data Structures Used:

For storing the roots of each binomial tree here we use a singly linked list. To hold the binomial trees, linked data structure is used, i.e., tree structure is implemented by linked list.

## Computational Complexity:

If there are 'n' nodes in the binomial heap, then the time complexities for extracting the minimum key are:-

(i) Finding the minimum key: takes $O(\log n)$
(ii) Deleting the minimum key: $O(\log n)$
(iii) Reversing the root list: $O(\log n)$
(iv) Union of the heaps: $O(\log n)$

∴ Overall time complexity: $O(\log n)$.

## Discussion:

(i) Binomial heap is an extension of binary heap that makes union or merge operations and other operations provided by binary heap faster and efficient.

(ii) Implementing binomial heaps can be more complex compared to simpler heap structures like binary heap.

ASSIGNMENT NO: 16 DATE: 22/01/2024

**Problem Statement:** Consider the problem of incrementing n-digit binary integers and compute its amortized cost in terms of n.

**Brief Description:**

Amortized analysis defers for both the kind of analysis, it considers a long sequence of events other than a single in isolation. In particular, amortized analysis defers from average case analysis in that probability is not involved. An amortized analysis of algorithm guarantees the average performance of each operation in the worst case, i.e., it gives a worst case estimate of the cost of a long sequence of event.

Suppose, if we have a sequence of m operations, on a data-structure and $t_i$ is the actual cost of the operation i, $1 \leq i \leq m$ and $c_i$ is the credit balance after operation i. (The credit is chosen in such a way, that it will be large when next operation is expensive and small when the next operation can be done quickly. Thus, the amortized cost $a_i$ of each operation is calculated as $\boxed{a_i = t_i + c_i - c_{i-1}}$ for $i = 1, 2, \ldots m$.

In case of incrementing n-digit binary integers, for credit balance, we take the total number of 1's in the binary integer. The actual cost is taken as the total number of bits flipped for incrementing the binary integer. Thus, the amortized cost can be calculated at each step with the help of the formula stated above.

## Algorithm in Text:

Let us consider a one dimensional array that stores n digits, where n is the number of digits of the binary integer. The binary-integer array is initialized to 0 initially and is incremented later one by one. At the time of incrementing, the total ~~credit~~ cost (total number of bits flipped for incrementing the previous number (binary integer) to its next binary integer.

Now, the current credit is calculated for the current binary integer by counting the total number of integers in it.

Now, the amortized cost is calculated by the formula

$a_i$ = total cost + current credit - previous credit. The previous credit is initially 0 and is updated at each stage by the current credit before going to the next stage.

Thus, the amortized cost for incrementing n-digit binary integers is calculated at each stage and displayed accordingly.

## Algorithm at a glance:

Input: Number of digits in the binary integer.

Output: Total cost, credit balance and the amortized cost at each stage.

Step 1: Enter the number of digits in the binary integer and store it n.

Step 2: Calculate the limit as 2^ n.

Step 3: For i = 0 to n-1, do,
Assign the binary-integer[i] = 0.
End for

Step 4: Initialize total_cost, amortized_cost and prev_credit to 0.

Step 5: For i = 0 to limit, do

(i) Calculate the current_credit by counting the total number of 1's in the binary integer.

(ii) Calculate the amortized cost by the formula total_cost + current_credit - prev_credit.

(iii) Display the total_cost, current_credit and the amortized_cost.

(iv) Assign prev_credit with the current_credit.

(v) Calculate the total_~~credit~~ cost after incrementing the binary ~~digit~~ integer by counting the total number of bits flipped.

End for

Step 6: End.

Step 5.(i): Calculation of current_credit:

```
For i=0 to n-1, do
    if (binary_integer [i] == 1) then
        credit = credit + 1
End if
End for.
```

Step 5.(v):

```
while (k < n && binary_integer[k] == 1) do
    binary_integer[k] = 0;
    total_cost ++
    k++     // Initially k = 0
End while
if (k < n) then
    binary_integer[k] = 1
    total_cost ++
End if
```

## Test Run:

(i) Let $n = 4$

| Step | Integer | total cost | credit | amortized cost |
|---|---|---|---|---|
| 0 | 0000 | — | 0 | — |
| 1 | 0001 | 1 | 1 | 2 |
| 2 | 0010 | 2 | 1 | 2 |
| 3 | 0011 | 1 | 2 | 2 |
| 4 | 0100 | 3 | 1 | 2 |
| 5 | 0101 | 1 | 2 | 2 |
| 6 | 0110 | 2 | 2 | 2 |
| 7 | 0111 | 1 | 3 | 2 |
| 8 | 1000 | 4 | 1 | 2 |
| 9 | 1001 | 1 | 2 | 2 |
| 10 | 1010 | 2 | 2 | 2 |
| 11 | 1011 | 1 | 3 | 2 |
| 12 | 1100 | 3 | 2 | 2 |
| 13 | 1101 | 1 | 3 | 2 |
| 14 | 1110 | 1 | 3 | 2 |
| 15 | 1111 | 1 | 4 | 2 |
| 16 | 0000 | 4 | 0 | 0 |

(ii) Let $n = 2$

| Step | Integer | total cost | credit | amortized cost |
|---|---|---|---|---|
| 0 | 00 | 0 | 0 | — |
| 1 | 01 | 1 | 1 | 2 |
| 2 | 10 | 2 | 1 | 2 |
| 3 | 11 | 1 | 2 | 2 |
| 4 | 00 | 2 | 0 | 0 |

Data Structure Used: One single dimensional array for storing the binary integer is used.

Computational Complexity:

The worst case complexity for incrementing a n bit binary integer is $O(n)$.

Special features:

We use amortized analysis to provide a tight bound on the average cost of operations over a worst case sequence. Even when some operations in that sequence are expensive.