Syllabus: Module 7

Heaps as priority queues, Binary heaps, binomial and Fibonacci heaps, Heaps in Huffman coding, Extendible hashing.

Heaps as priority queue

Priority Queue

- Priority queue contains data items which have some preset priority.
- While removing an element from a priority queue, the data item with the highest priority (low or high value) is removed first.
- In a priority queue, insertion is performed in the order of arrival and deletion is performed based on the priority.

Priority Queue -Operations

- insert / enqueue – add an item to the rear of the queue.
- remove / dequeue – remove an item from the front of the queue based on priority.

Priority Queue -Strength

- Quickly access the highest-priority item.
- Priority queues allow you to peek at the top item in O(1)

Priority Queue -Application

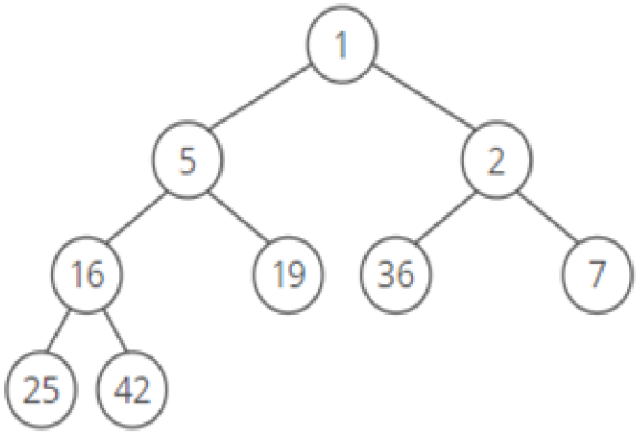
- Imaging a big list of bugs for an engineering team to tackle. You want to keep the highest-priority bugs at the top of the list.

- Operating system schedulers may use priority queues to select the next process to run, ensuring high-priority tasks run before low-priority ones.

Priority Queue -Implementation- Binary Heap

Priority queues are often implemented using binary heaps.

Notice how the highest priority is at the top of the heap, ready to be grabbed in O(1) time.



- To **enqueue** an item, add it to the heap using the priority as the key. ($O(\log(n))$ time)
- To **peek** at the highest priority item, look at the item at the top. ($O(1)$ time)

Heap

A heap is a data structure that stores a collection of objects (with keys), and has the following properties:

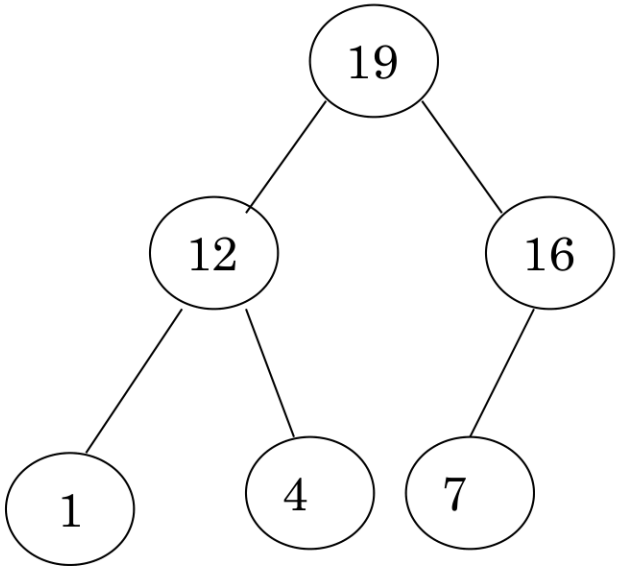
- Structural property (Complete Binary tree)
- Heap order property (min/max heap)

Binary Heap

- **Binary Heap**
  - A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible
  - 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 children nodes.
  - The former is called as max heap and the latter is called min-heap.
  - The heap can be represented by a binary tree or array.

Heap

- Max-heap

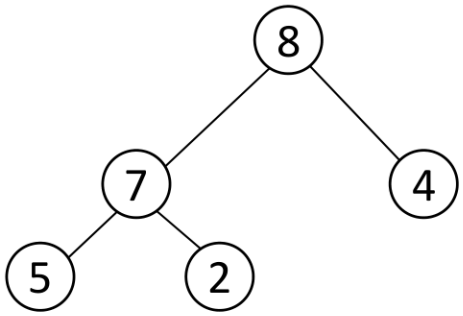


19	12	16	1	4	7

Array A

Heap

- Two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x, $Parent(x) \geq x$



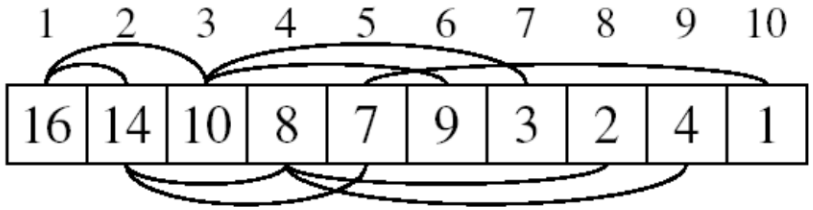
Heap

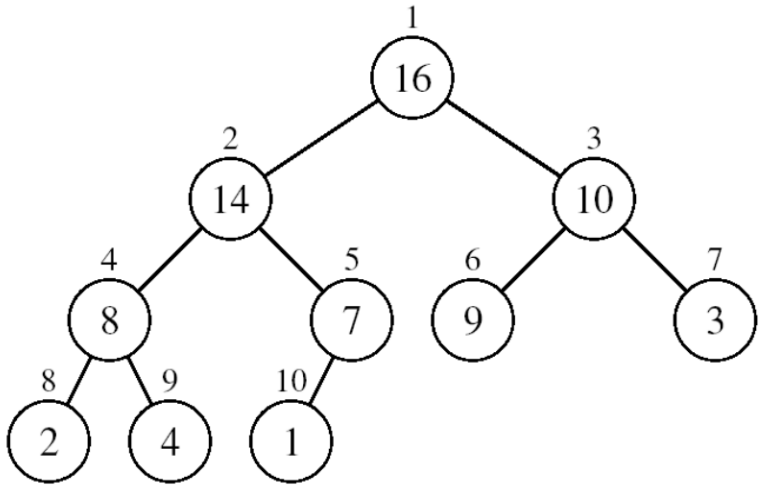
From the max heap property, it follows that:
"The root is the maximum element of the heap!"

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$





Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

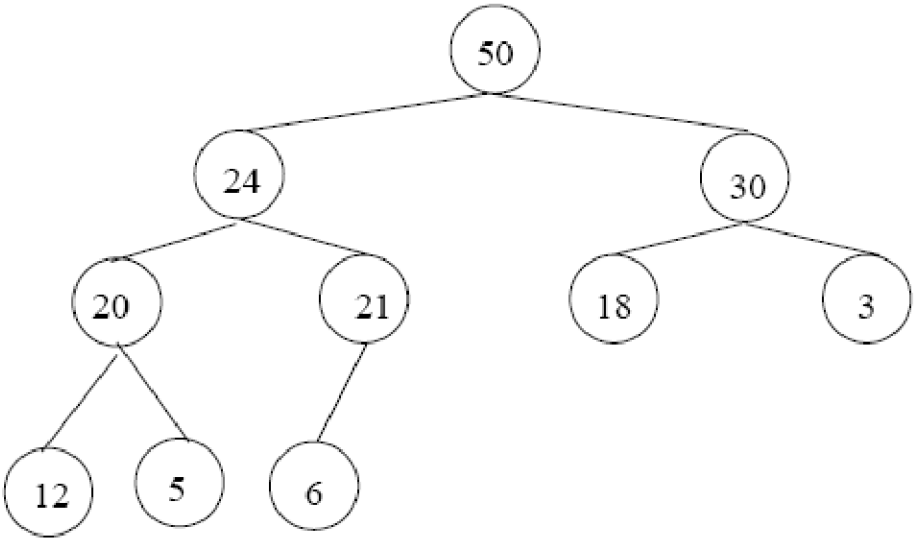
$$A[PARENT(i)] \geq A[i]$$

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

$$A[PARENT(i)] \leq A[i]$$

Adding/Deleting Nodes in Heap

- New nodes are inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

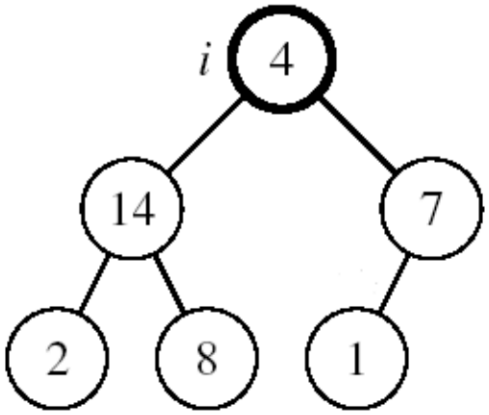


Operations on Heaps (Eg: MaxHeap)

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT

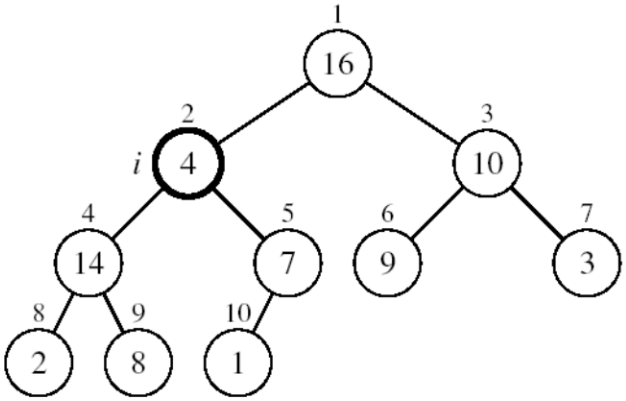
Maintaining the Max Heap Property

- Suppose a node is smaller than a child in a max heap
  - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



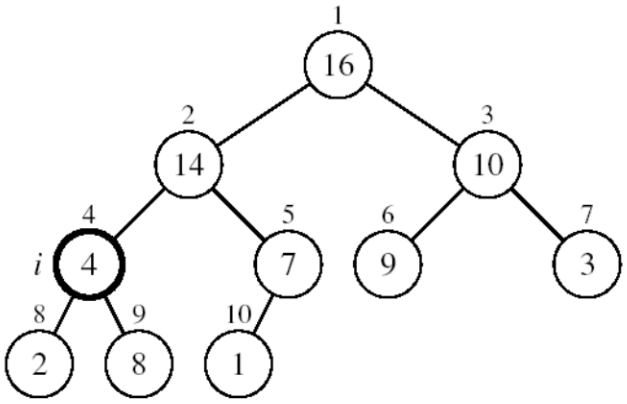
Example

MAX-HEAPIFY(A)



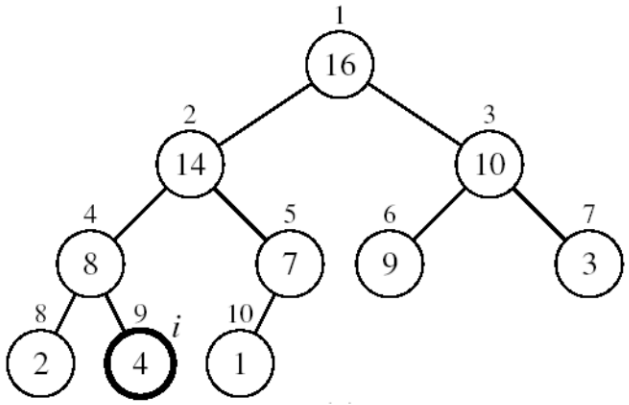
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

A[4] ↔ A[9]



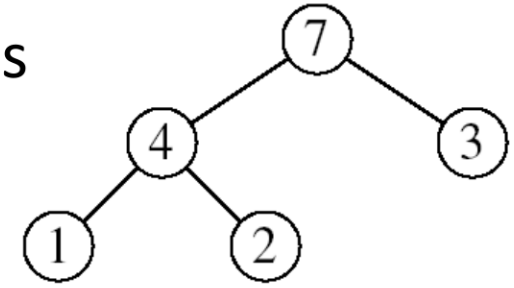
Heap property restored

Heap Sort

- Heap sort is a comparison based sorting technique based on Binary Heap data structure.

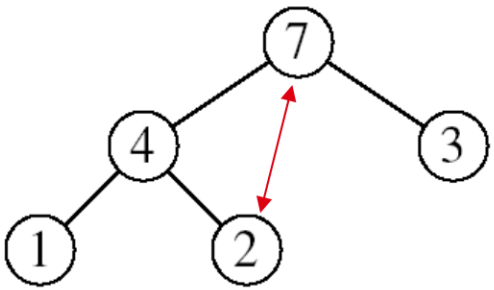
Heapsort

- Goal:
  - Sort an array using heap representations

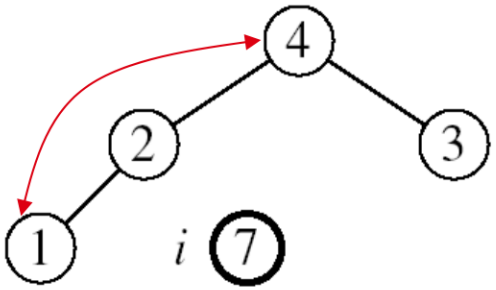


- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

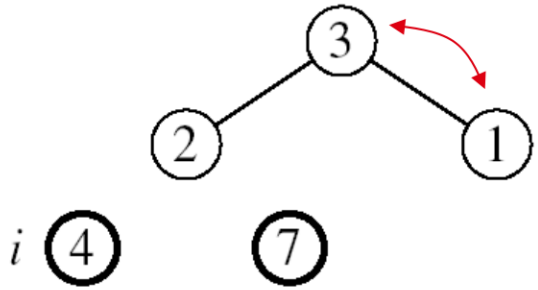
Example: A=[7, 4, 3, 1, 2]



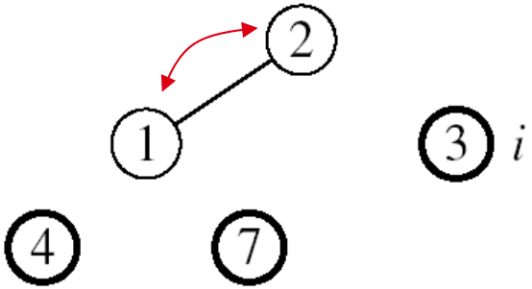
MAX-HEAPIFY(A)

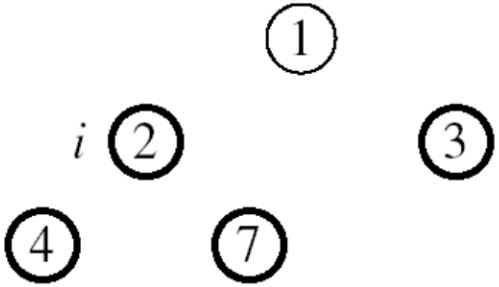


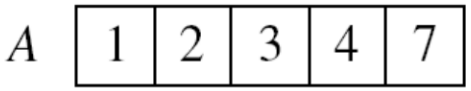
MAX-HEAPIFY(A)



MAX-HEAPIFY(A)







MAX-HEAPIFY(A)

HEAPSORT Algorithm

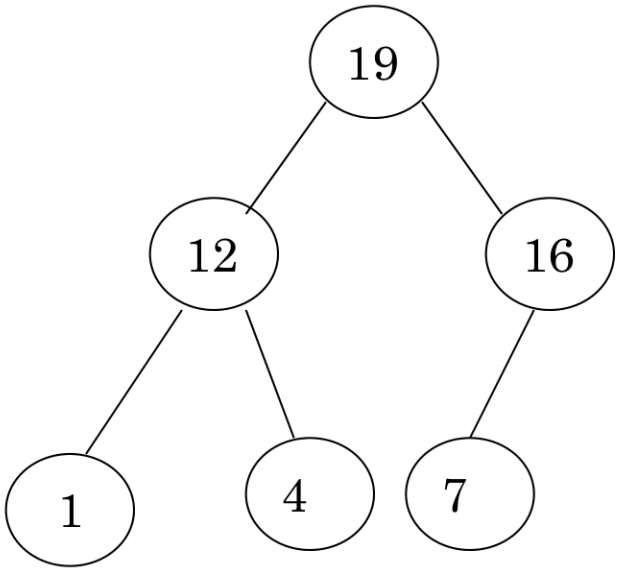
1. BUILD-MAX-HEAP(A)
2. **for** $i \leftarrow length[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. MAX-HEAPIFY(A, 1, i - 1)

Time Complexity Analysis

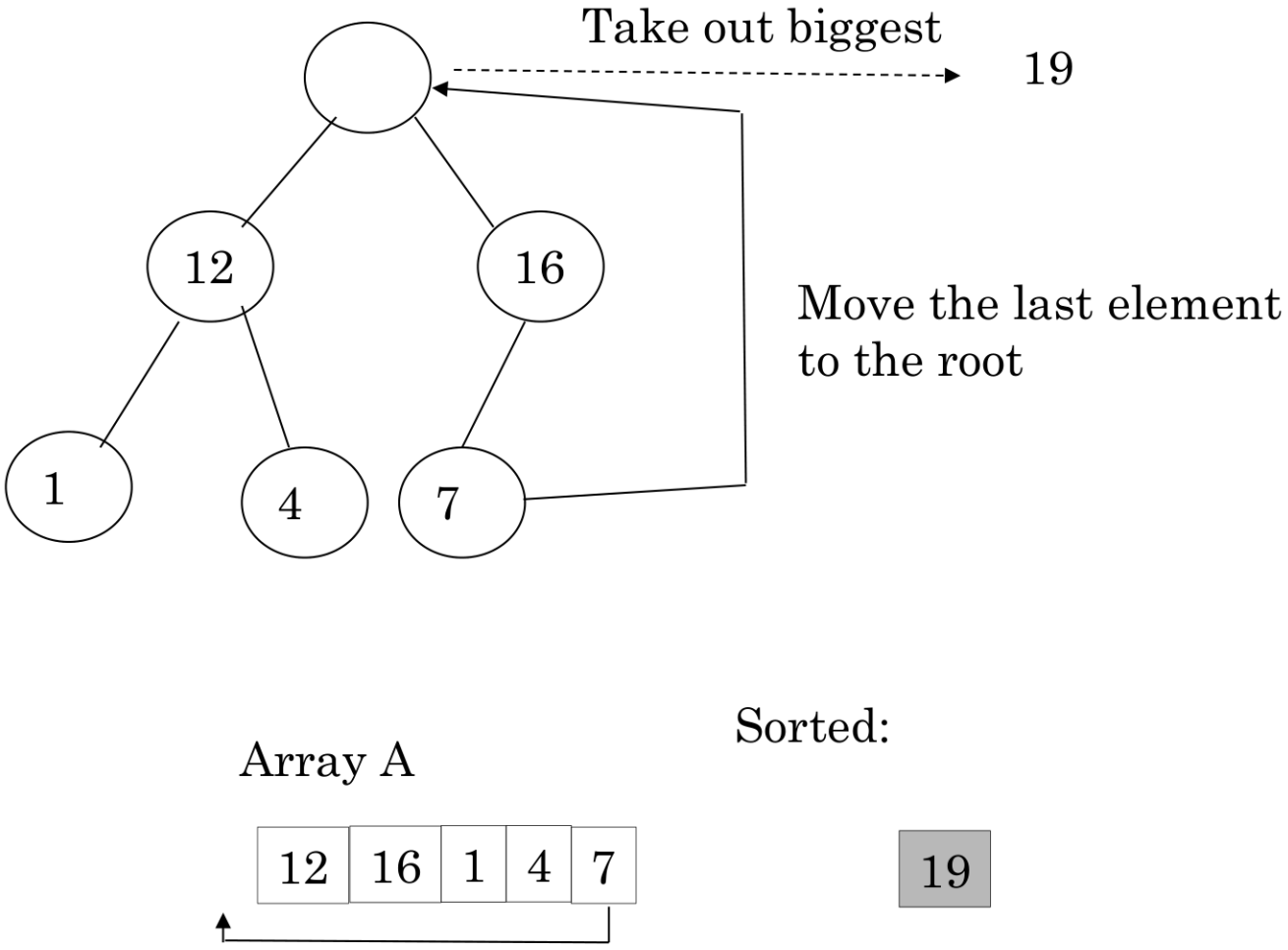
- Build Heap Algorithm will run in O(n) time
- There are *n*-1 calls to Heapify each call requires O(log *n*) time
- Heap sort program combine Build Heap program and Heapify, therefore it has the running time of O(n log n) time
- Total time complexity: O(n log n)

Heap Sort

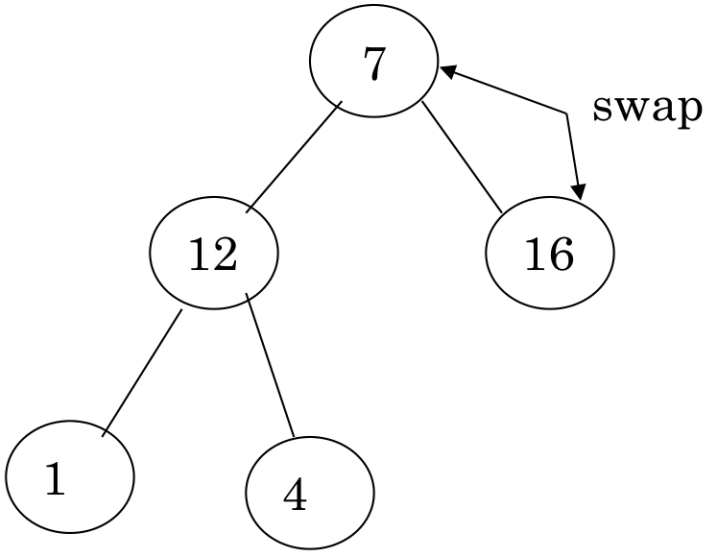
- The heapsort algorithm consists of two phases:
  - build a heap from an arbitrary array
  - use the heap to sort the data
- It is an in-place sorting technique
- To sort the elements in the decreasing order, use a min heap
- To sort the elements in the increasing order, use a max heap



Another Example of Heap Sort



HEAPIFY()

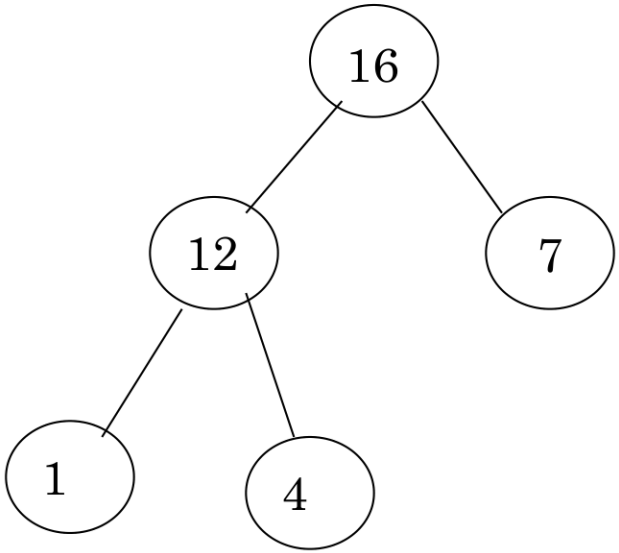


Array A

7	12	16	1	4

Sorted:

19

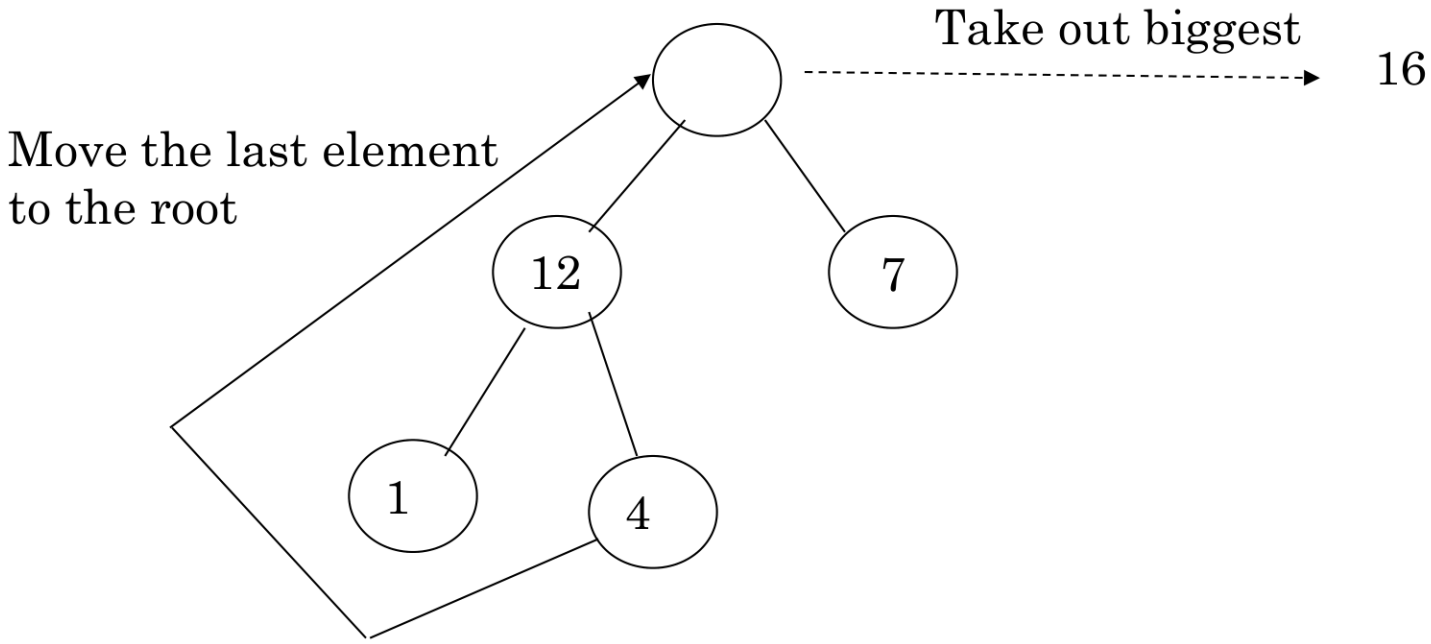


Array A

16	12	7	1	4

Sorted:

19

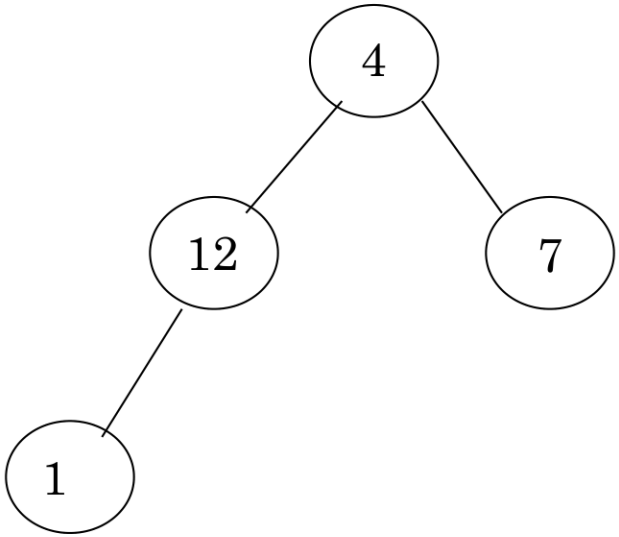


Sorted:

Array A

12	7	1	4

16	19



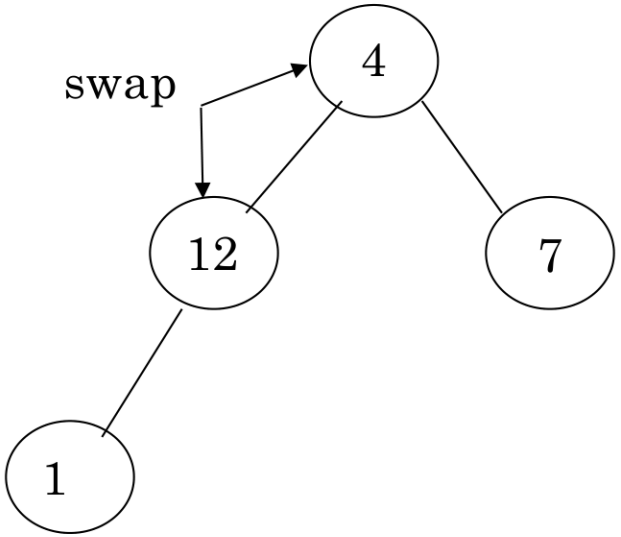
Array A

4	12	7	1

Sorted:

16	19

HEAPIFY()

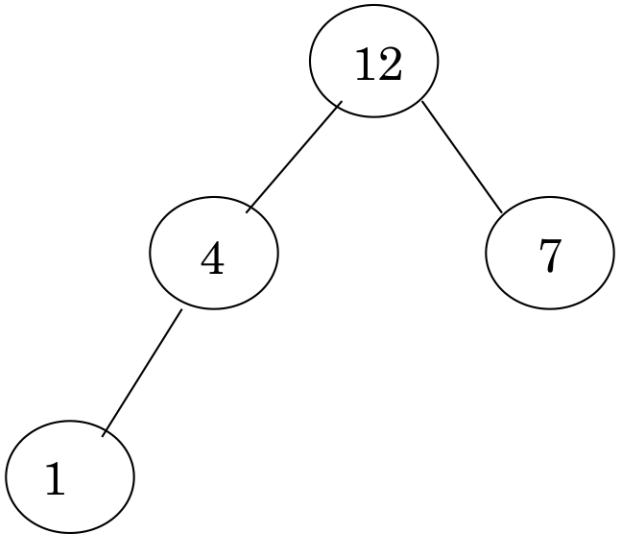


Array A

4	12	7	1

Sorted:

16	19

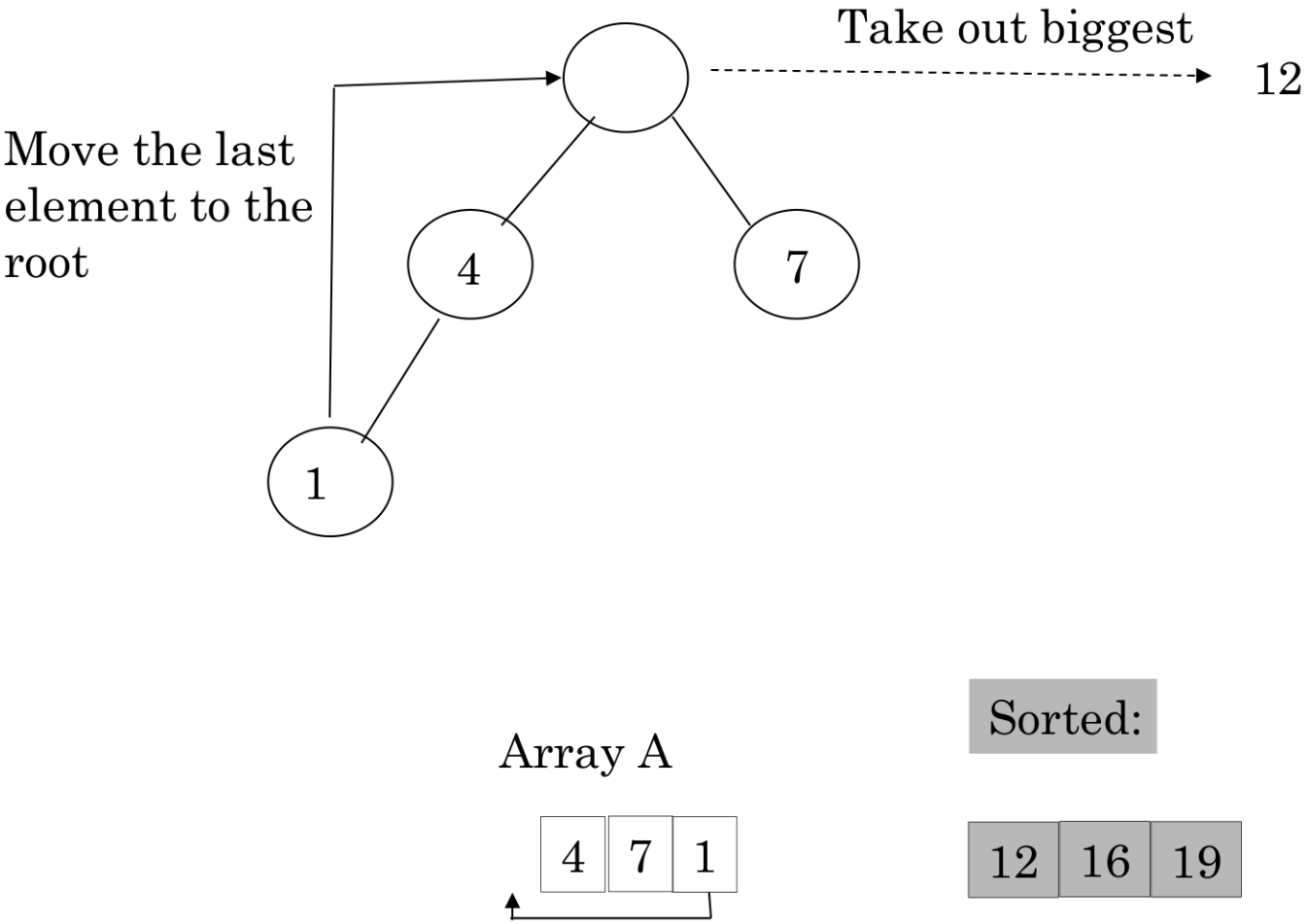


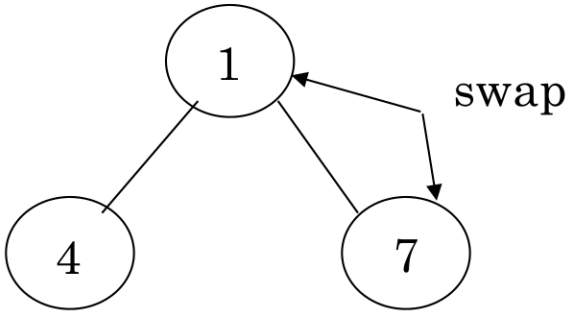
Array A

12	4	7	1

Sorted:

16	19


Take out biggest
12
Move the last element to the root
4
7
1
Array A
4
7
1
Sorted:
12
16
19

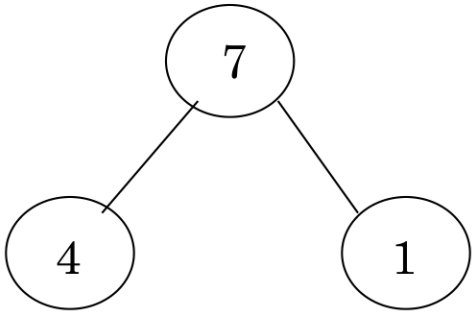


Array A

1	4	7

Sorted:

12	16	19

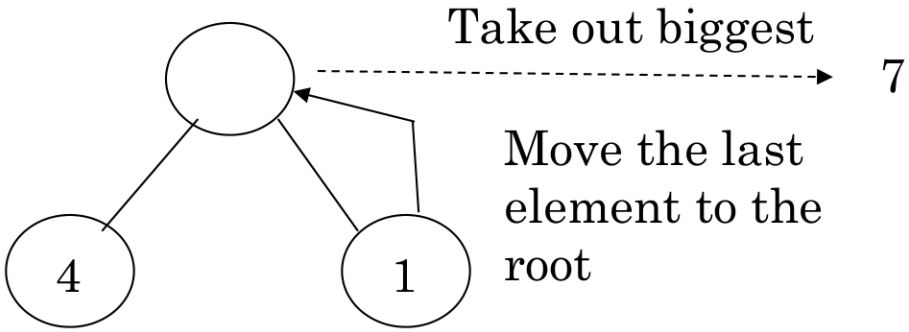


Array A

7	4	1

Sorted:

12	16	19



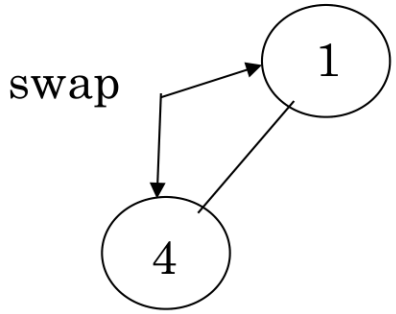
Array A

1	4

Sorted:

7	12	16	19

HEAPIFY()



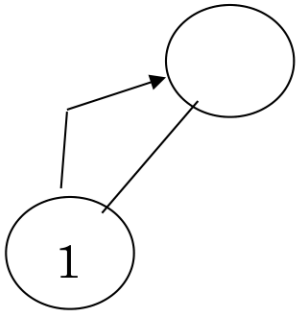
Array A

4	1

Sorted:

7	12	16	19

Move the last element to the root



Take out biggest

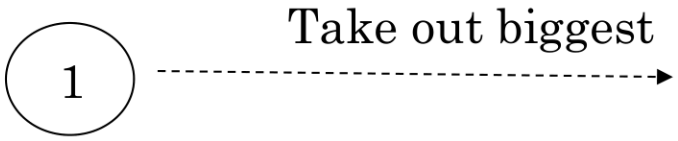
4

Array A

1

Sorted:

4	7	12	16	19

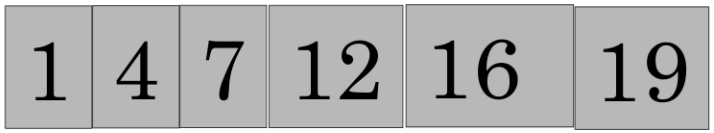


Array A

Sorted:

1	4	7	12	16	19

Sorted:



Exercise

- Demonstrate, step by step, the operation of Build-Max Heap on the array and perform Heap sort on the built heap

  A=[5, 3, 17, 10, 84, 19, 6, 22, 9]

Binomial Heap

Binomial Heap

Understanding Binomial tree is essential to understand binomial heap

Binomial Heap

- A binomial heap can be defined as the collection of binomial trees that satisfies the heap properties, i.e., min-heap.
- Mainly, Binomial heap is used to implement a priority queue.
- It is an extension of binary heap that gives faster merge or union operations along with other operations provided by binary heap.

Binomial Heap

Binomial Tree:

- Ordered tree
- Not a binary tree
- Not balanced
- If the tree is binomial tree of order k, the representation is B_k → Binomial tree with order k

Binomial Heap

- **Binomial Tree Structure**: A binomial tree of order k is defined recursively:
  - A binomial tree of order 0 is a single node.
  - A binomial tree of order k consists of two binomial trees of order k-1, where one tree is linked as the leftmost child of the other.

- **Union/Merge Operation**: A binomial heap supports a union operation that efficiently merges two binomial heaps in O(log n) time by combining binomial trees of the same order

Binomial Heap

Operations:

- **Insert**: Insertion of an element involves creating a new binomial heap with a single node and merging it with the existing heap.
- **Find Minimum**: Traverse the root list to find the minimum key.
- **Delete Minimum**: Remove the root with the minimum key and reinsert its children as separate binomial trees.
- **Union**: Merging two binomial heaps involves combining the binomial trees of the same order.

Binomial Heap

B_0

(with single node)

(0 children at the root)

Properties

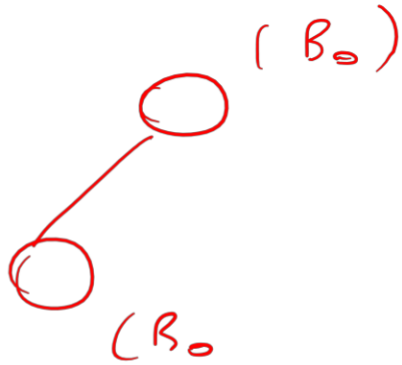
* B_k has k children at the root
* B_k can be formed using two B_{k-1} trees
* The root of one B_{k-1} will be the leftmost child of the root of other B_{k-1}

Binomial Heap

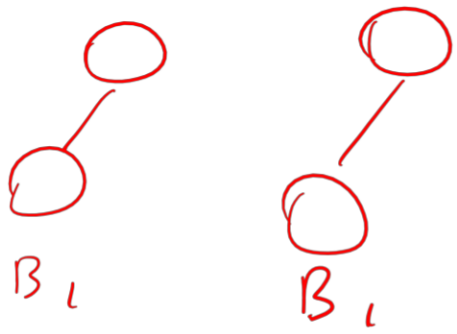
B_1

(1 child at root)

(B_1 is formed using two B_0s.)



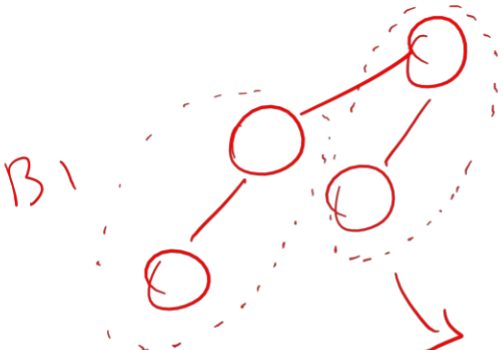
B_2



Combine two B_1s to form B_2

→ root of one B_1 will become the leftmost child of another B_1

Binomial Heap



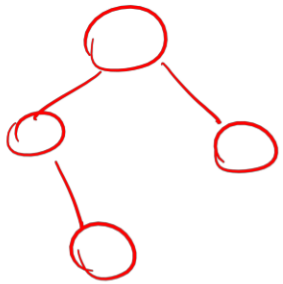
B_2 has two children at the root

Binomial Heap

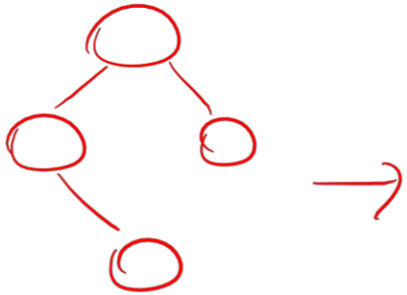
B_3

$k = 3, \; k-1 = 2$

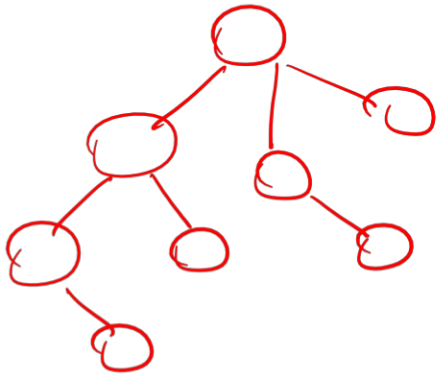
B_3 is formed using 2 B_2 trees



B_2



B_2



3 children for the root

Binomial Heap

Properties of Binomial tree

1) There are 2^k nodes in a Binomial tree B_k

2) The height of the tree is k.

Binomial Heap

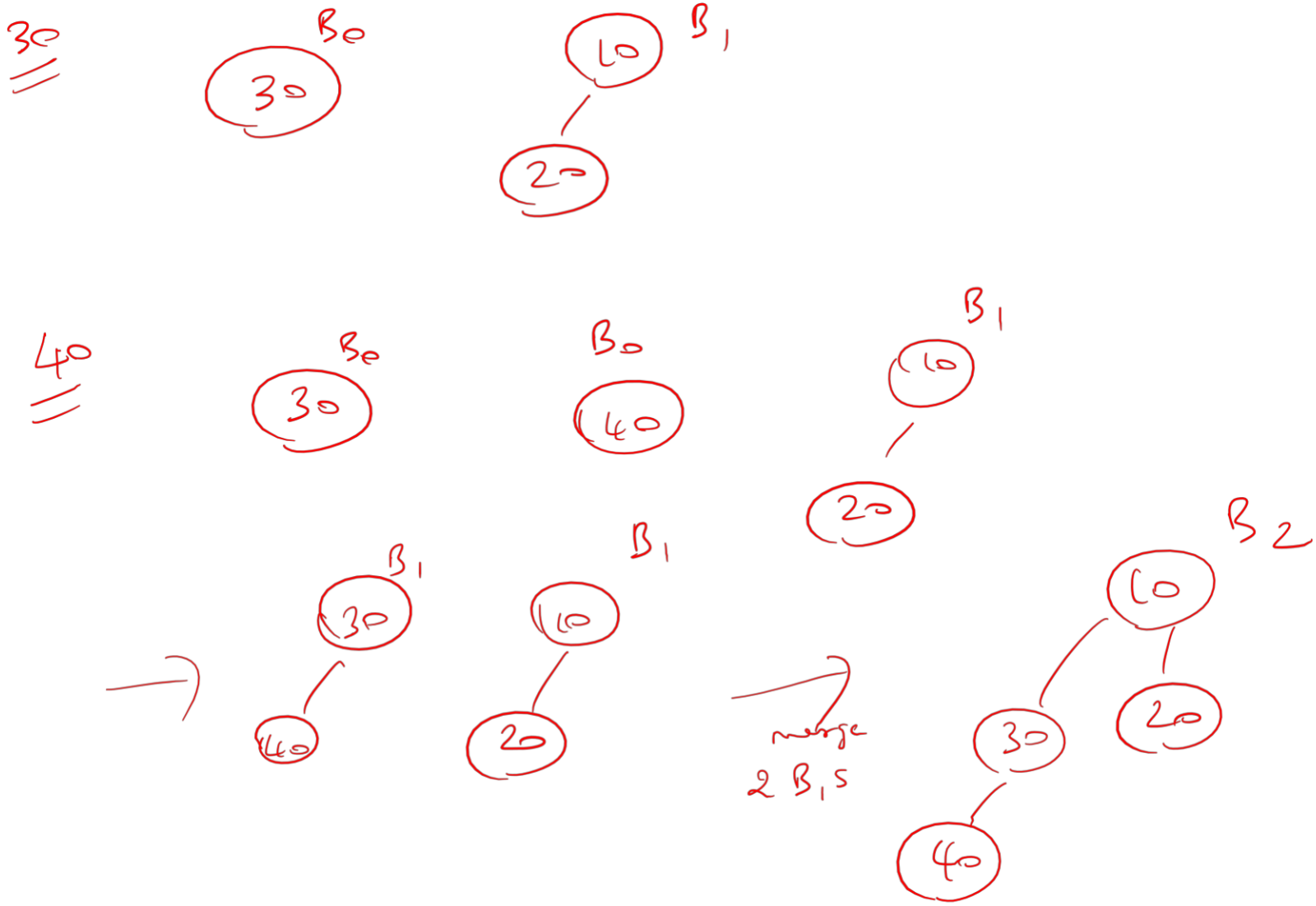
Construct a binomial heap with the following data : 10, 20, 30, 40, 50, 60, 70, 80

10

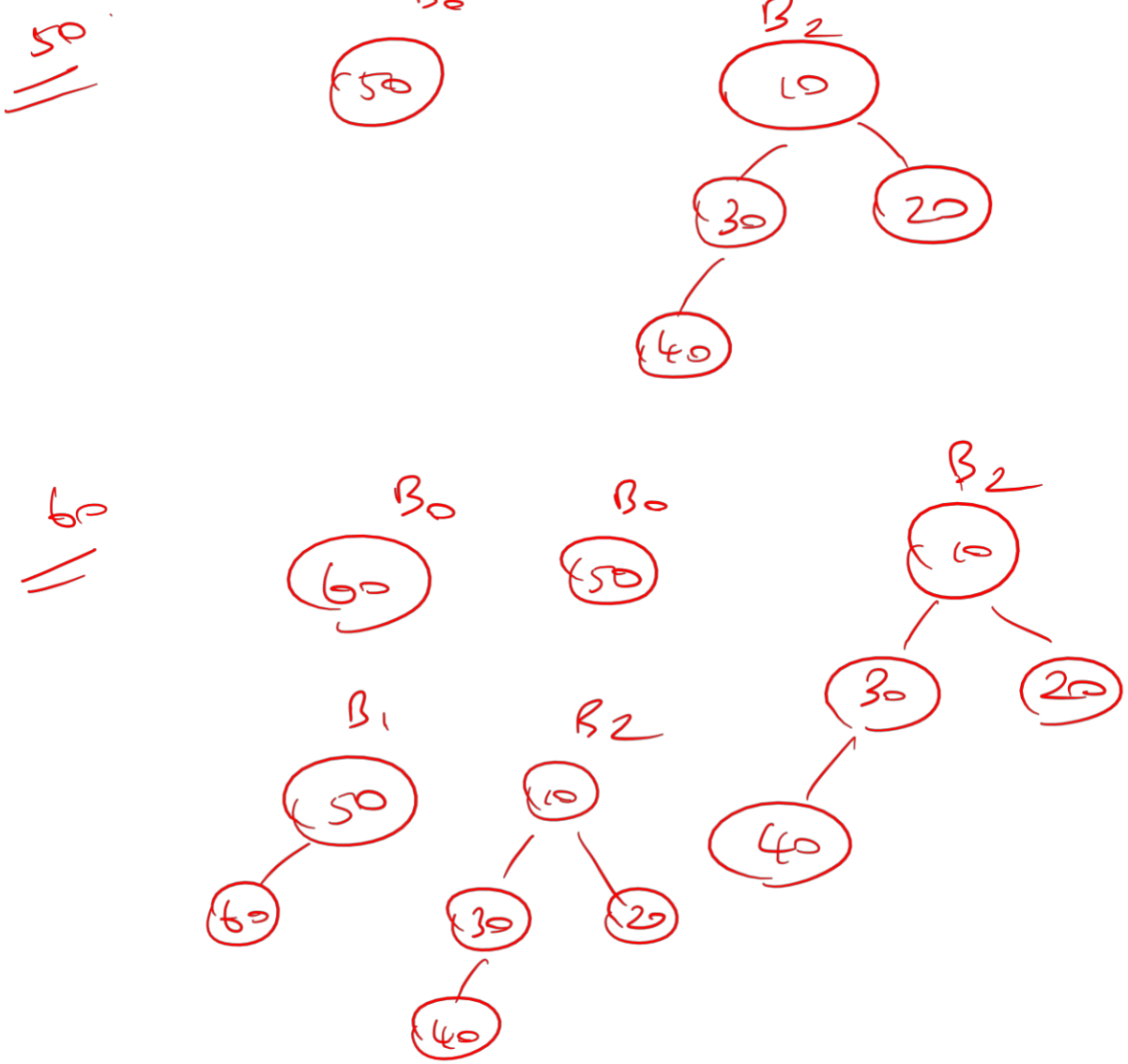
20

Two B_0 s → merge (smaller one is the root)

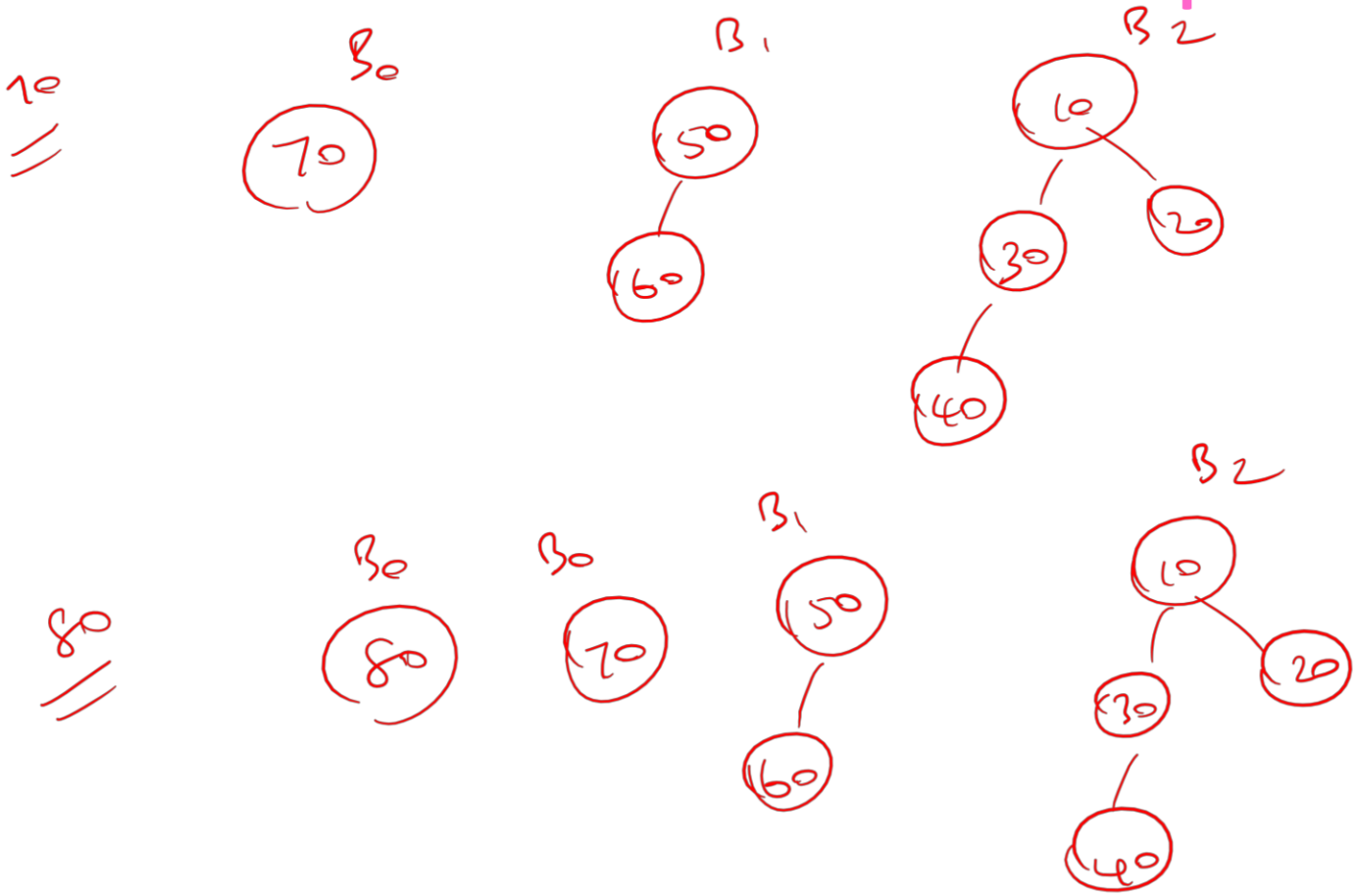
Binomial Heap



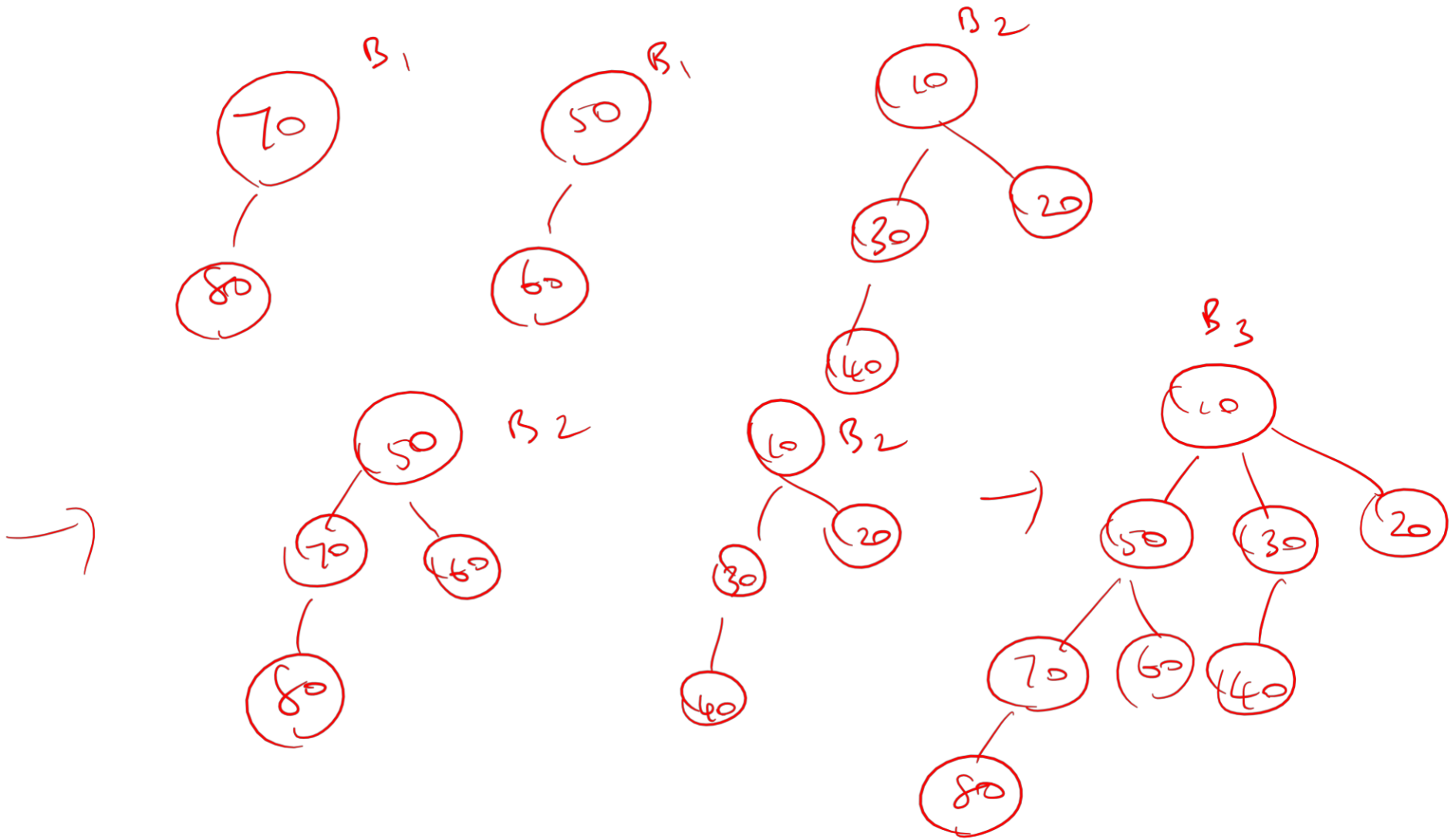
Binomial Heap



Binomial Heap



Binomial Heap

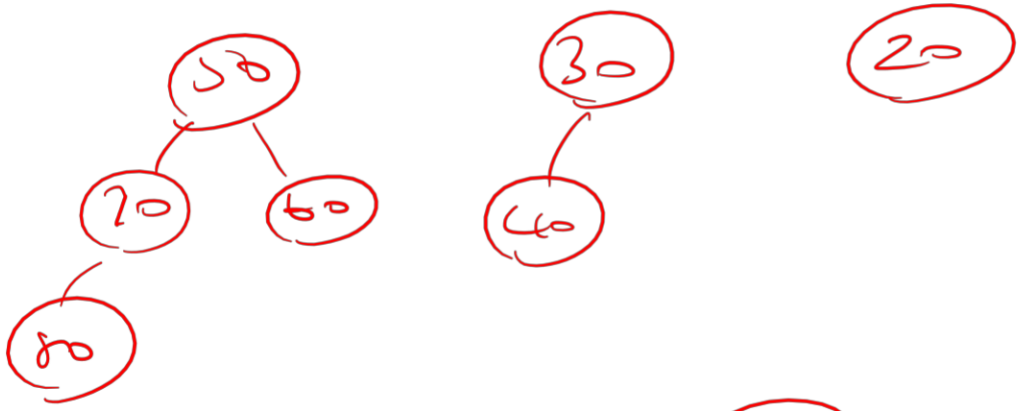


Binomial Heap

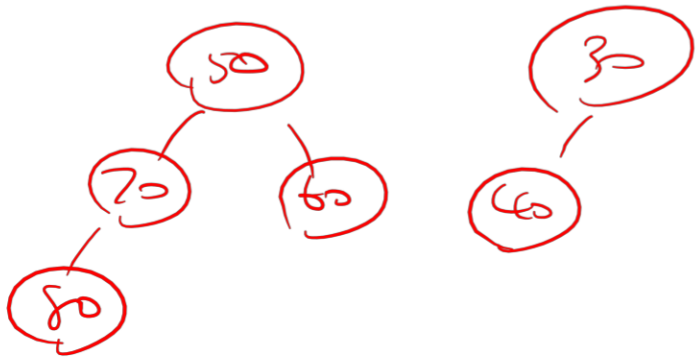
To delete (DeleteMin)

10

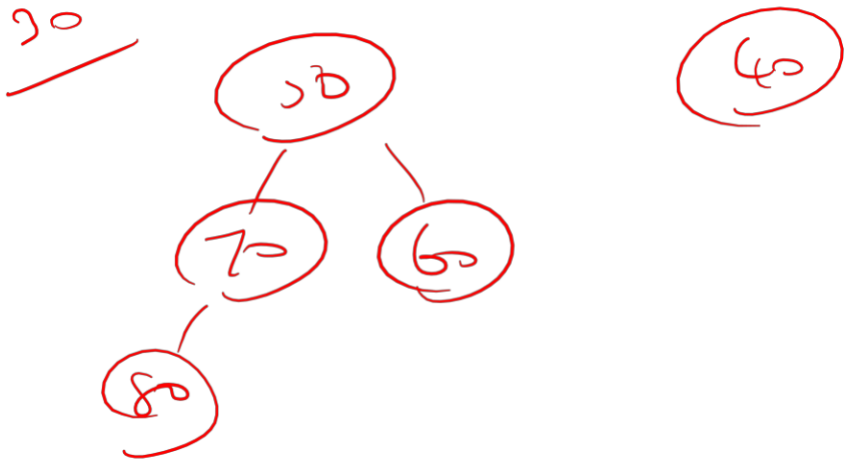
After removing 10, Binomial heap becomes



20



Binomial Heap



Binomial Heap

Exercise:

Construct a binomial heap where we insert the following numbers: 12, 7, 25, 15, 28, 33, 41.

Binomial Heap

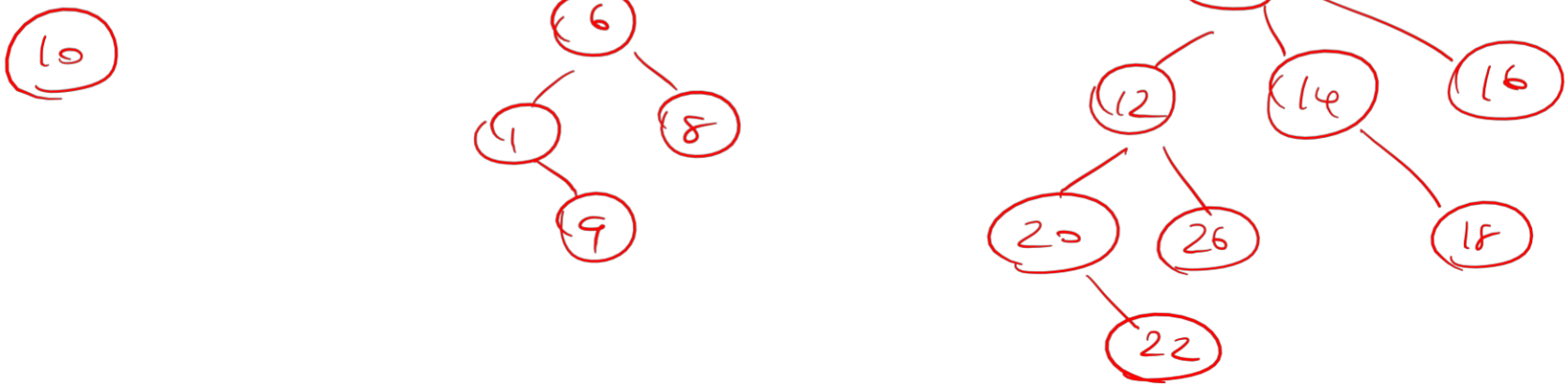
Binomial Heap

* Collection of many Binomial trees

B_0 (no children)

B_2 (2 children)

B_3 (3 children)



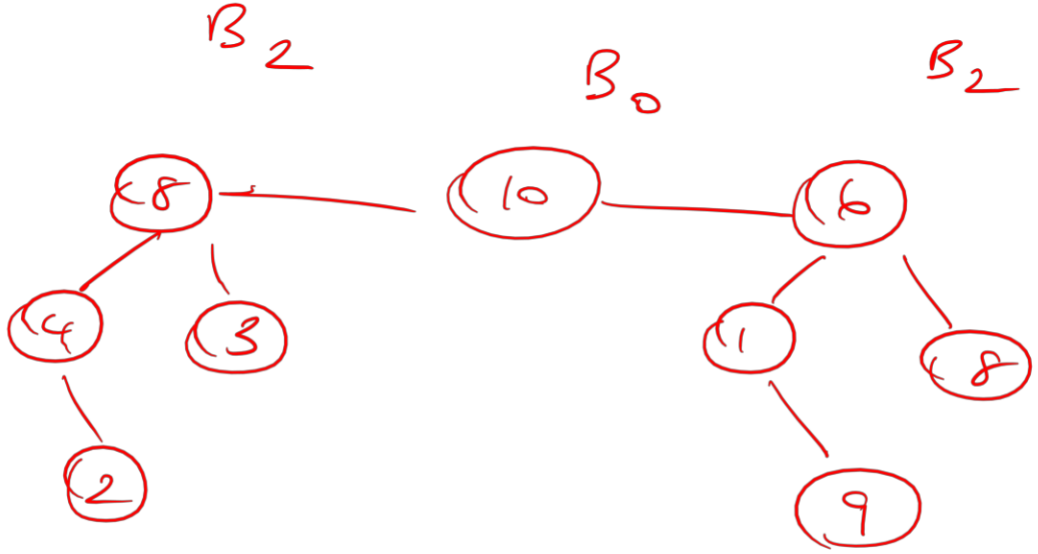
Binomial Heap

Condition for a binomial tree to become binomial heap

* Every node should be smaller than all its descendants in each binomial tree (min heap order property)
* There should not be more than one binomial tree with same order

Binomial Heap

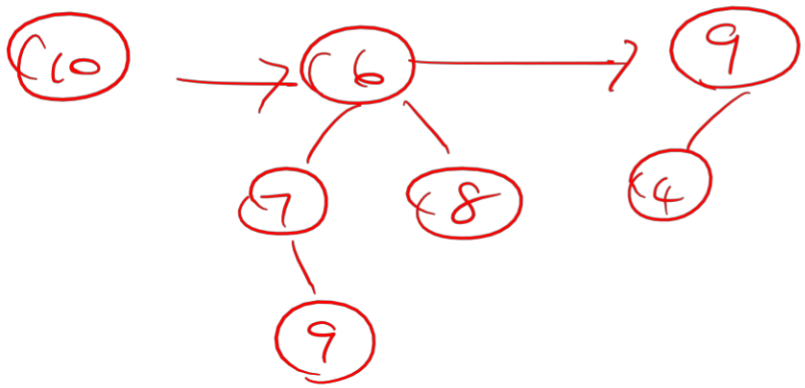
eg



It's not a binomial heap as two trees are with same order

Binomial Heap

* Binomial heap should be arranged in ascending order



(not in ascending order, here not a binomial heap)

Ascending order of the tree order (here it is in the order, B0, B2, B1)

Binomial Heap

Suppose $n = 13$ nodes, we need to make binomial heap, but you are not sure if it has B_0, B_2, B_4 binomial tree

So, represent 13 in binary format

13 → 1 1 0 1

3 2 1 0

will contain B_0, B_2, B_3 as it has 1 in binary representation

Binomial Heap

B_3 B_2 B_0

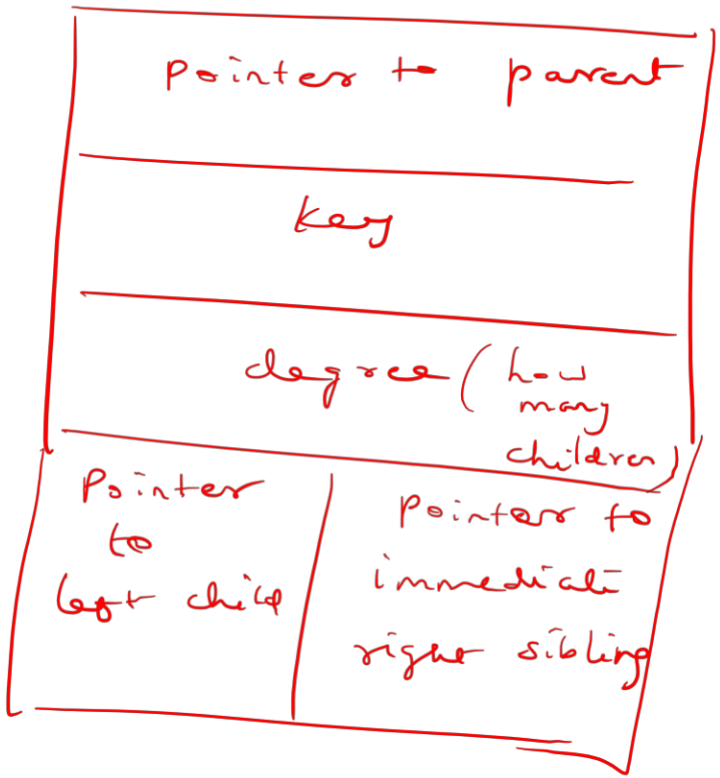
$2^3 + 2^2 + 2^0 = 13$ nodes

Binomial Heap

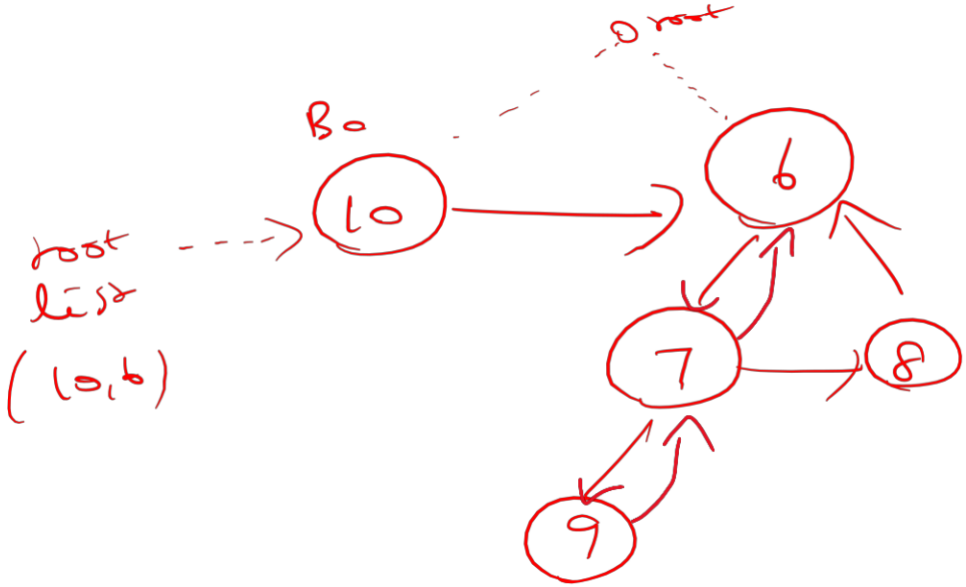
Structure of a node in binomial heap

Binary heap

Binomial heap



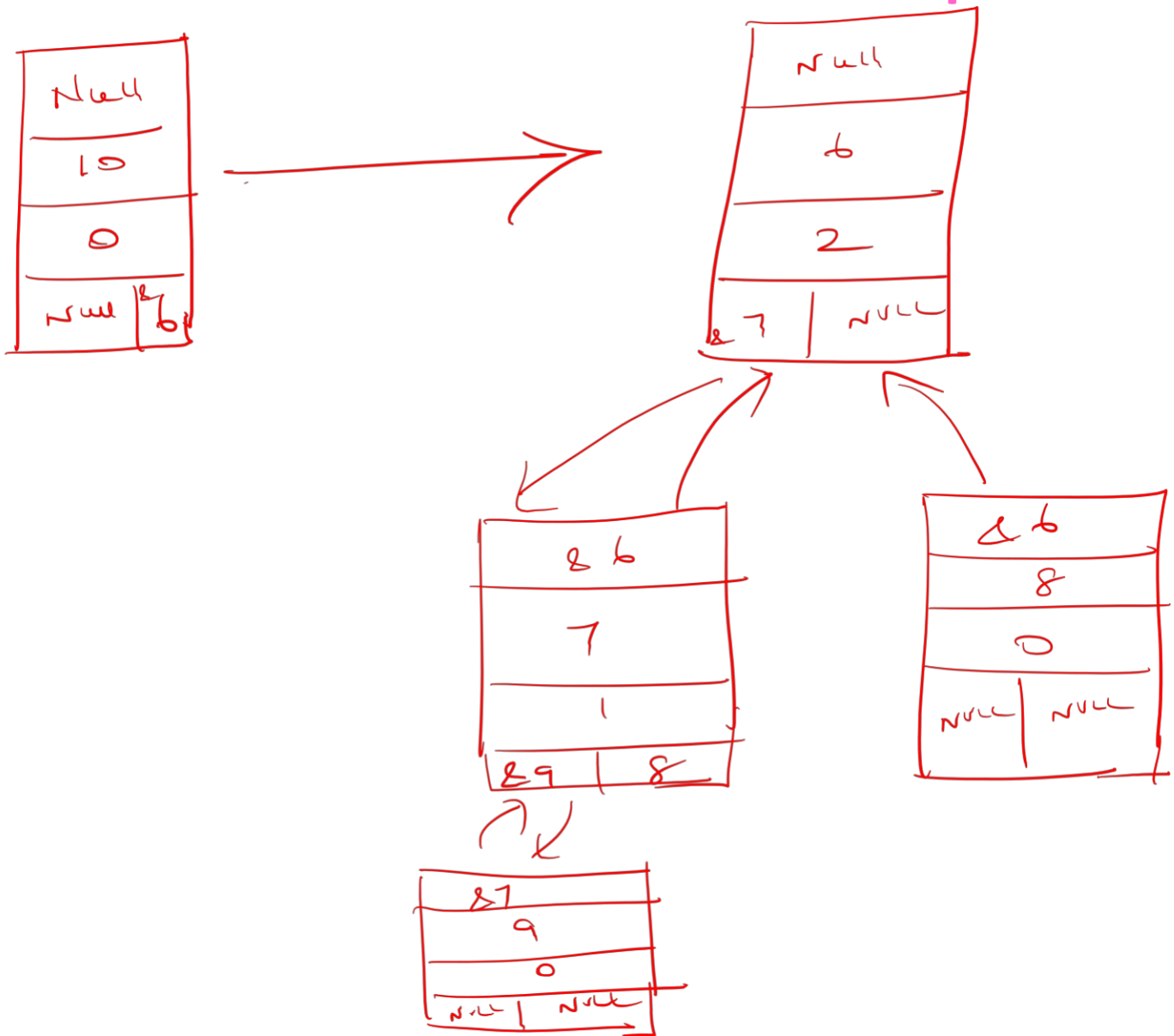
Binomial Heap



Note:

NO CONCEPT RIGHT CHILD

Binomial Heap



Binomial Heap

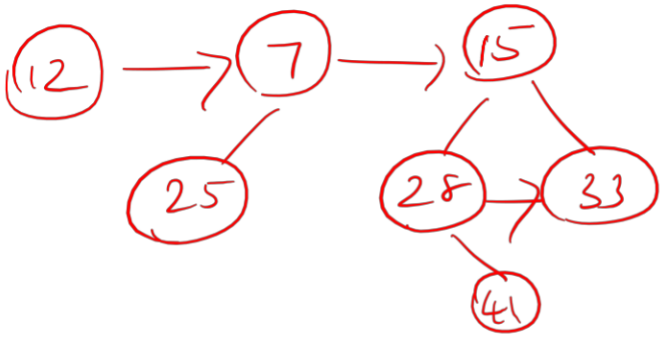
why we need binomial heap ?

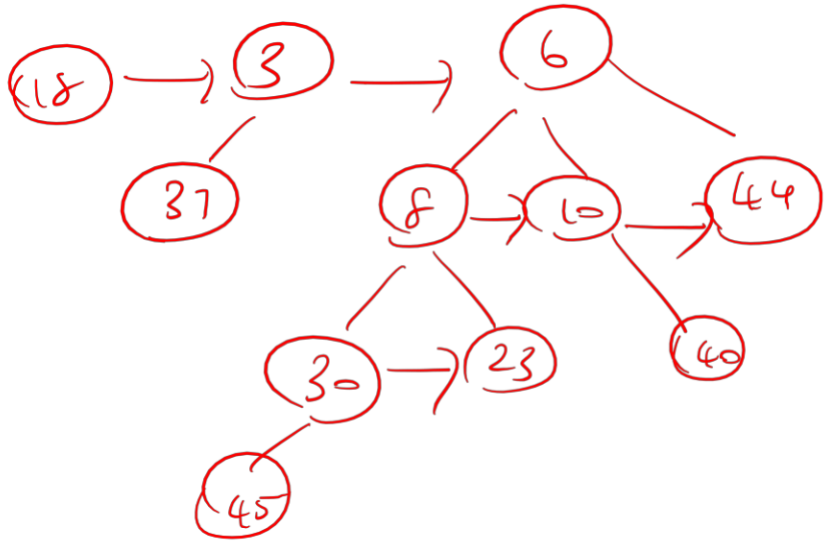
- Binary heap → merge 2 heap
  → $O(n \log n)$
- Binomial heap → reduced from $O(n \log n)$ to $O(\log n)$

Binomial Heap

Merge in binomial heap / union operation

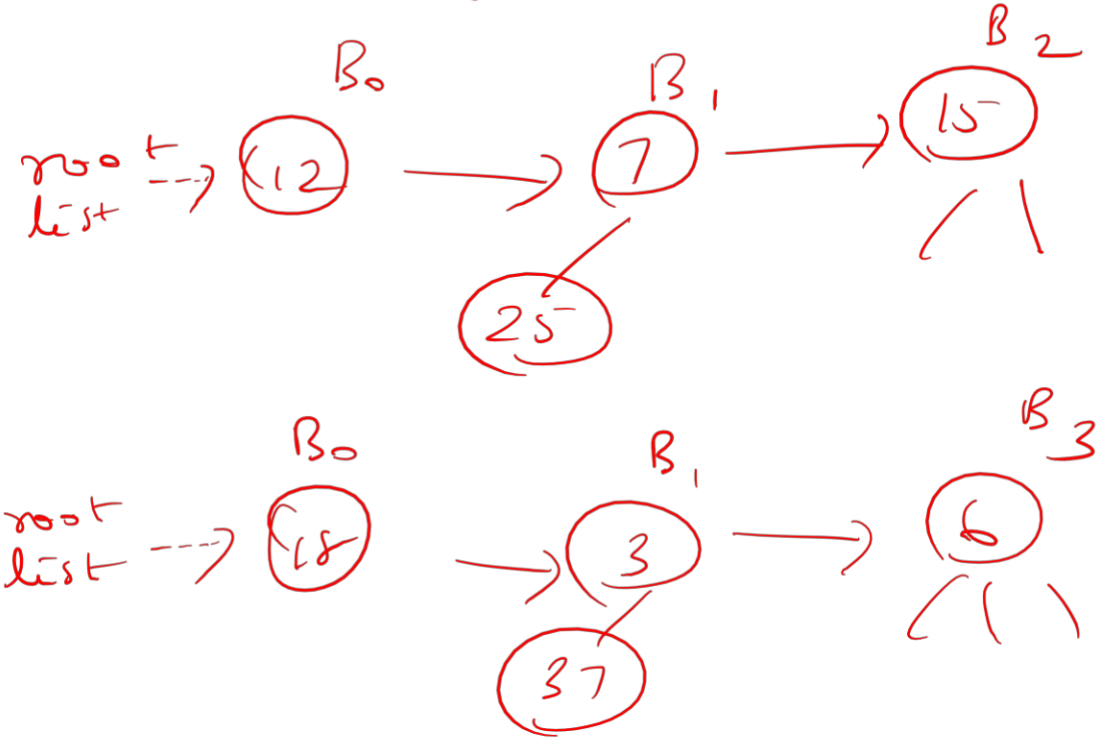
Other operations (insert / delete) will be easy to understand if we understand merge operation





Binomial Heap

Let's merge previous two binomial heaps

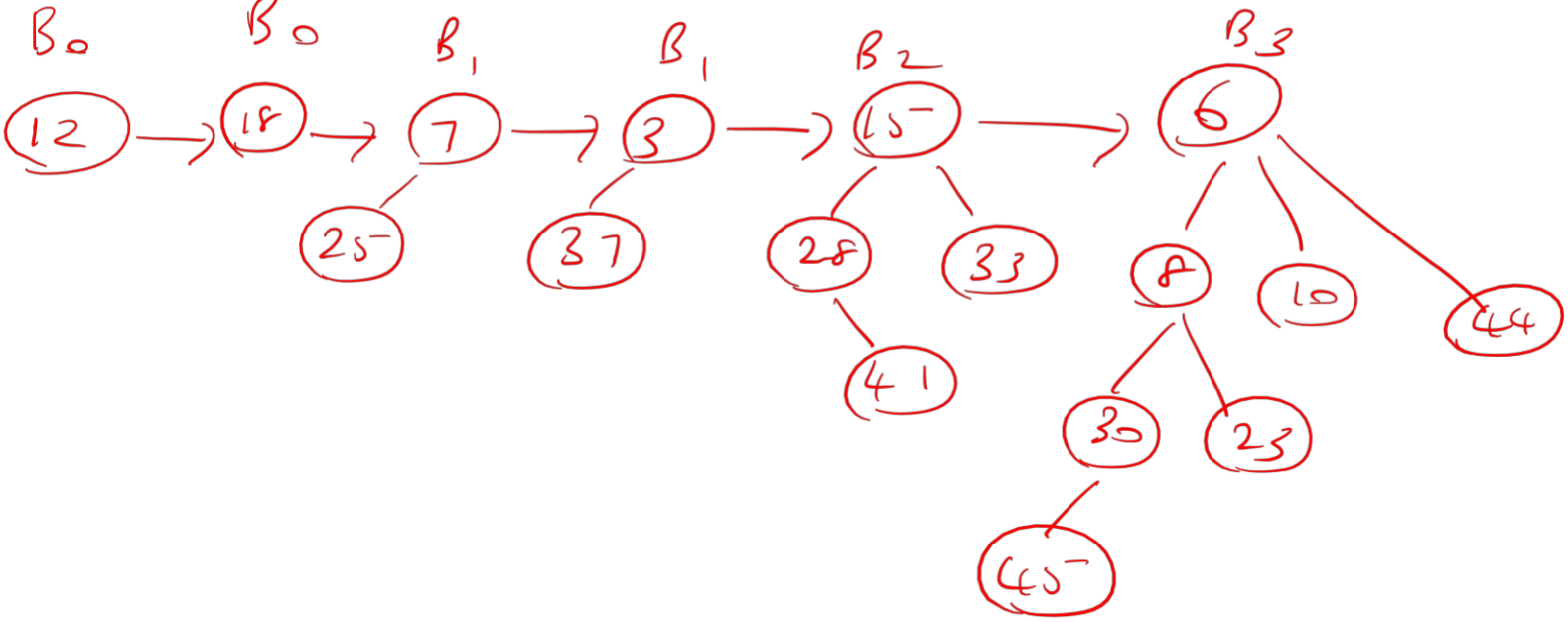


Binomial Heap

I. Merge both root list

Binomial Heap

Ascending order of order of tree



Binomial heap cannot have binomial trees of same order.

(2 B_0 's)

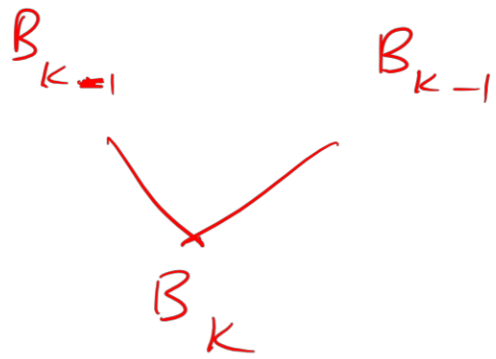
(2 B_1 's)

So, merge operation is incomplete

Binomial Heap

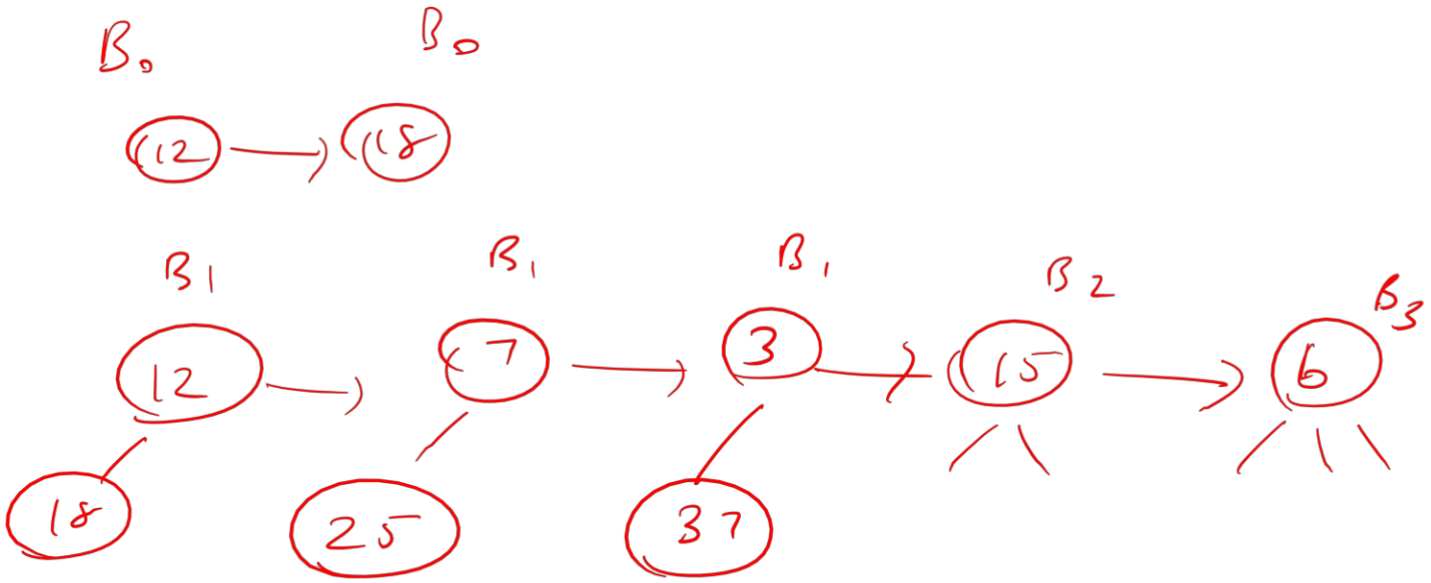
II.

If we encounter 2 binomial trees with same order, merge them



Still maintain heap order property (minimum element of binomial tree is at the root)

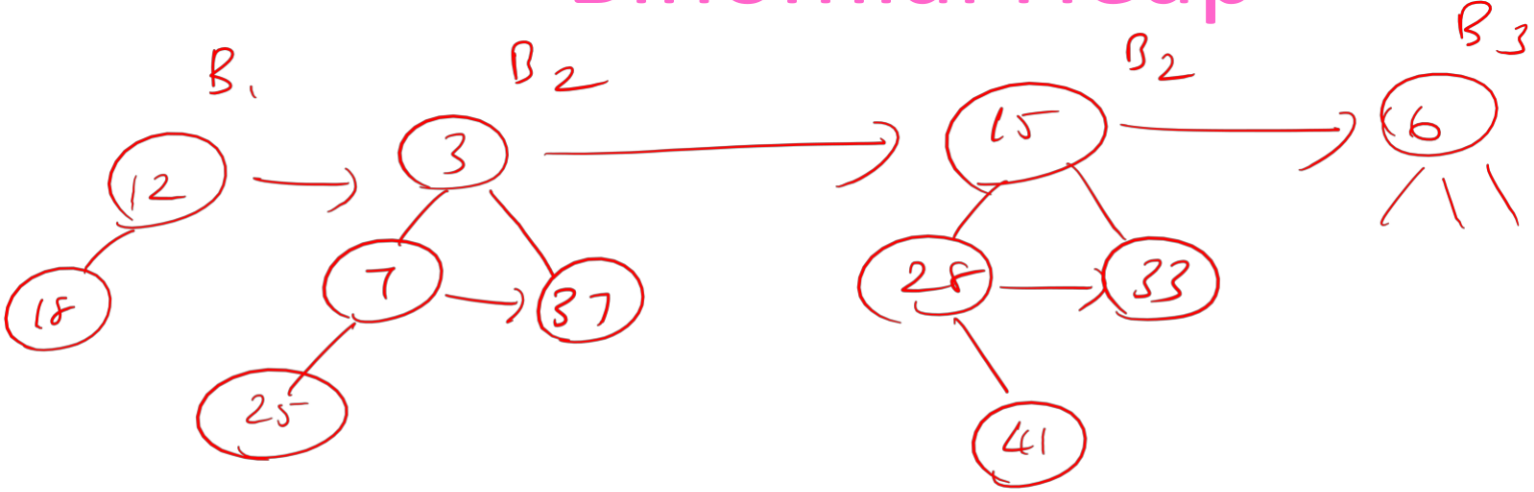
Binomial Heap



Again, we have 3 trees with same order

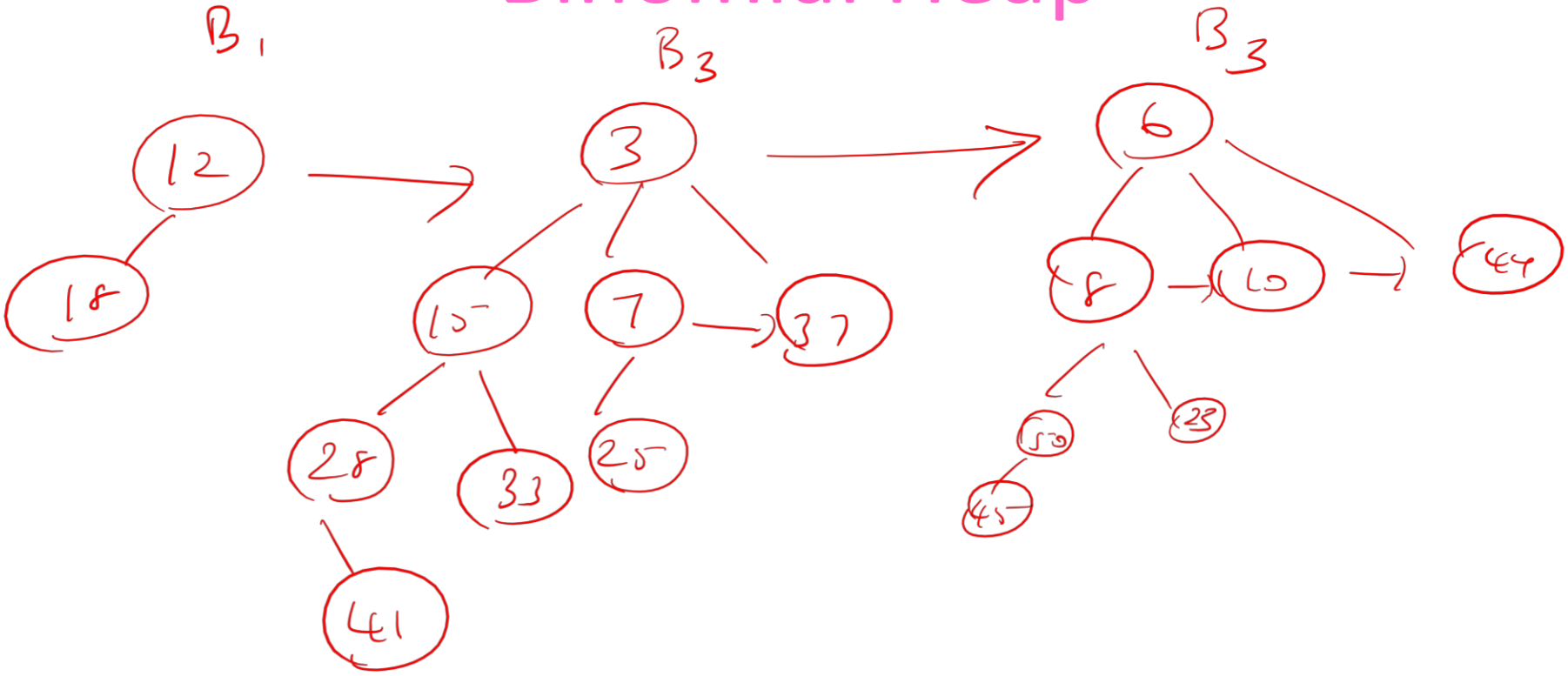
But leave first B_1 & merge next two B_1 trees according to heap order

Binomial Heap



Two B_2 s

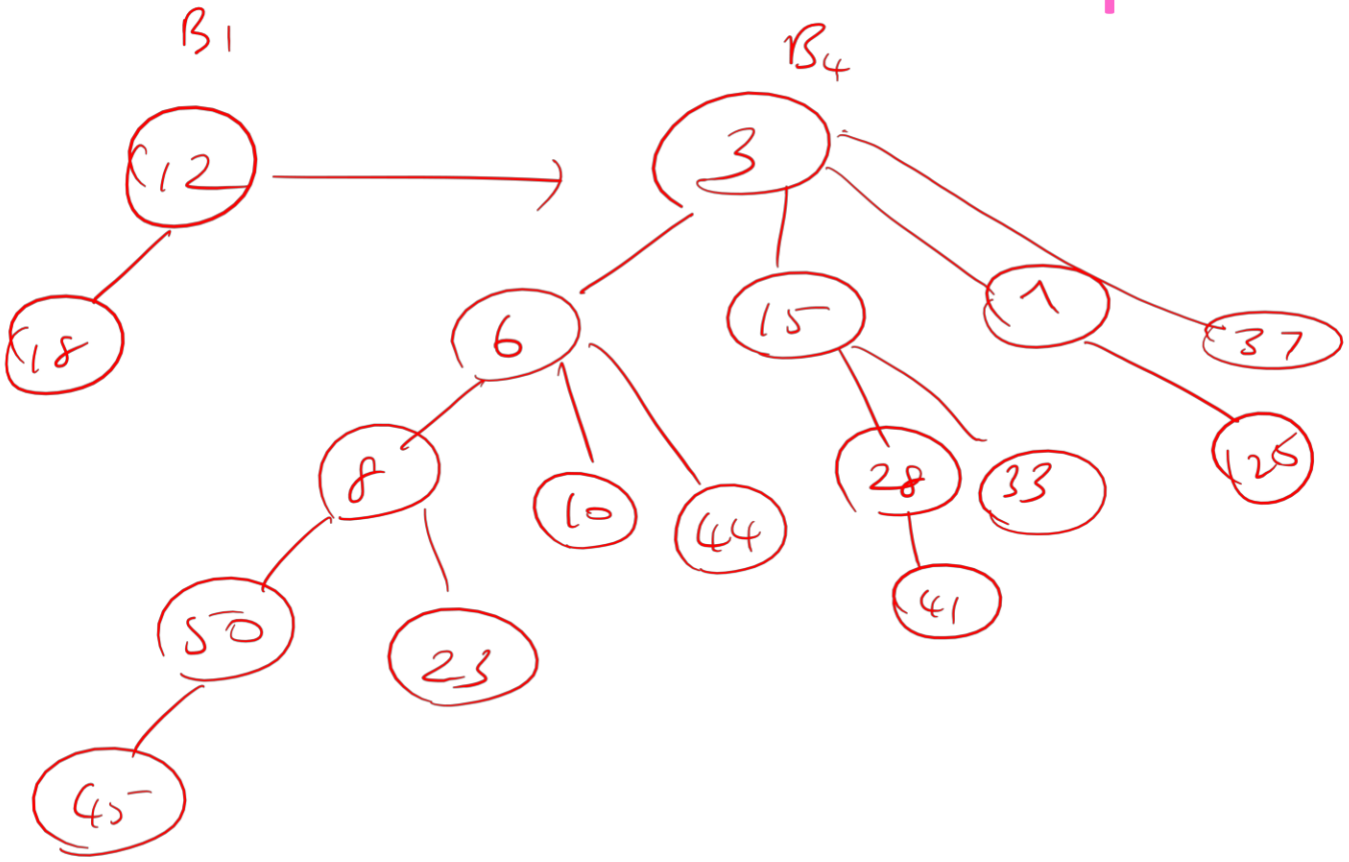
Binomial Heap



Two B_3s

merge → always left most child of smaller root

Binomial Heap



Two trees of different order,
So merging steps

Binomial Heap

Find Min

Traverse only the root list

6 is the minimum element

Summary-Binomial Heap

1. Binomial heap is not a binary tree
2. It is not a balanced tree
3. Has an order value k
4. B_k is formed by combining two B_{k-1} and while combining consider Min Heap property and attach the other tree to the left of another tree (there is no concept of right child)
5. Has min heap property
6. Collection of binomial trees form binomial heap where no two trees are with the same order and also collection is arranged according to the tree order

Fibonacci Heap

Fibonacci Heap

It implements a mergeable heap

Supports 5 operations:

- Make-heap – creates an empty heap
- Insert - inserts a node
- Minimum - returns the minimum node
- Extract-min – deletes the minimum and returns it
- Union –merges two heaps

Fibonacci Heap

Properties:

1. Collection of Min-heap trees (Not a collection of binomial trees)
2. Trees may be in any order in the root list (No ascending order in terms of degrees)

(For eg: In binomial tree, the order may be B0, B2, B3, B4. In case of fib heap, it is not there)

4. Can have two trees of same order/degree (In binomial tree, two trees of same order is not allowed, but in Fib heap, it is allowed :

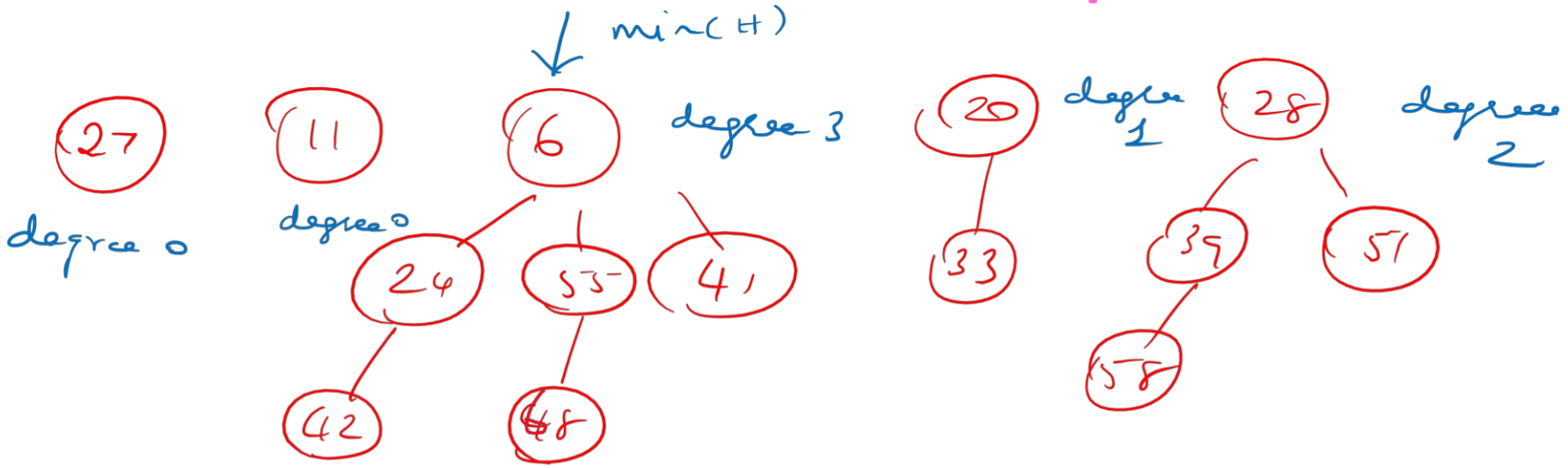
For eg: Two degree 0s) , so fib heap is a relaxed structure

5. Always there is a pointer pointing to the minimum element in the root list

Fibonacci Heap

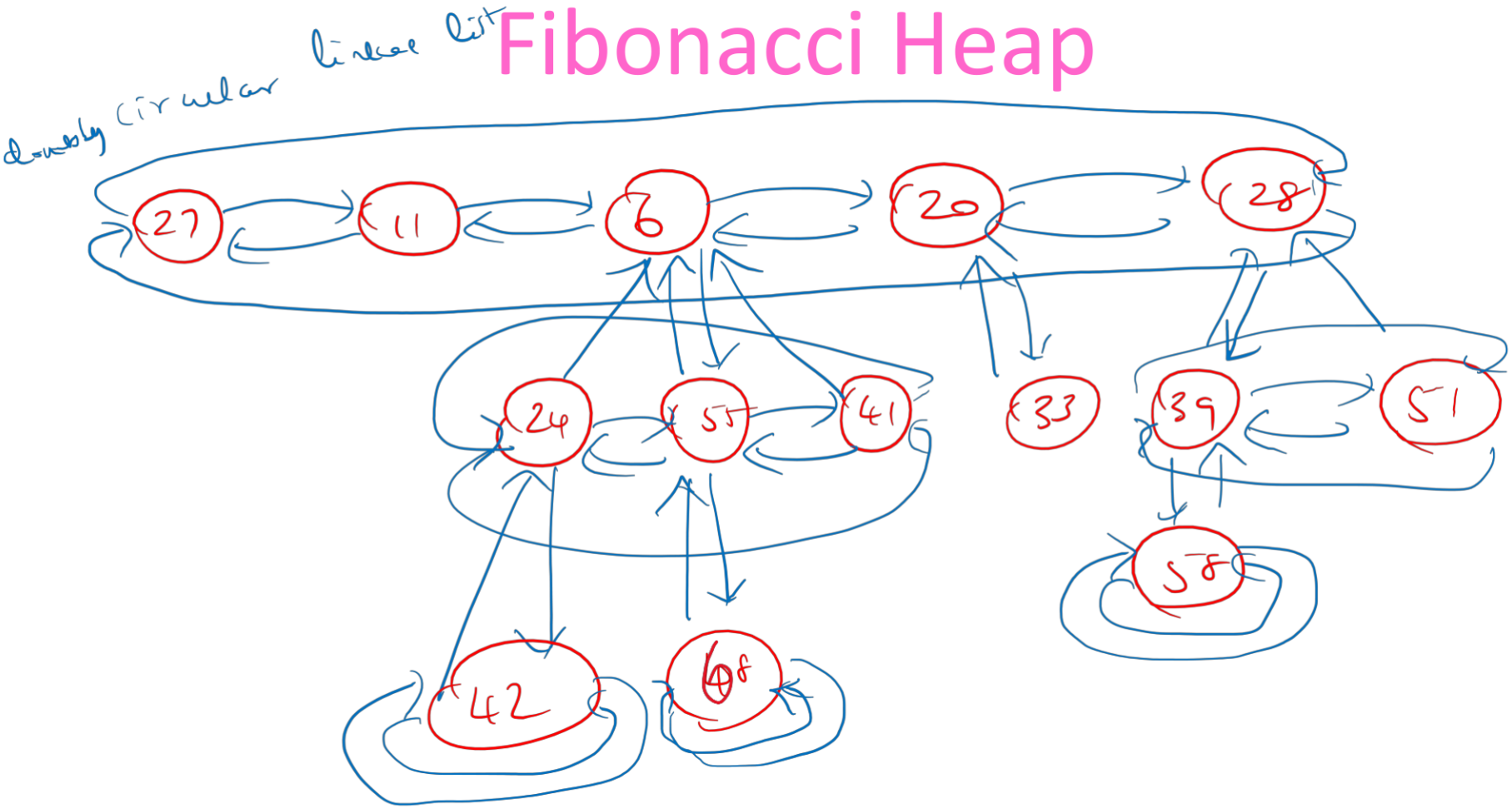
6. Each child points to its parent
7. Each parent points to any one of the child
   (For eg: node 6 points to node 55 in the next to next slide)
8. degree(x) : No of children of node of a tree
9. mark(x) : 1 : lost one of its child
   0 : lost no child
10. Each siblings are connected through a circular doubly linked list
11. Min(H) is a pointer points to minimum node in the root list

Fibonacci Heap



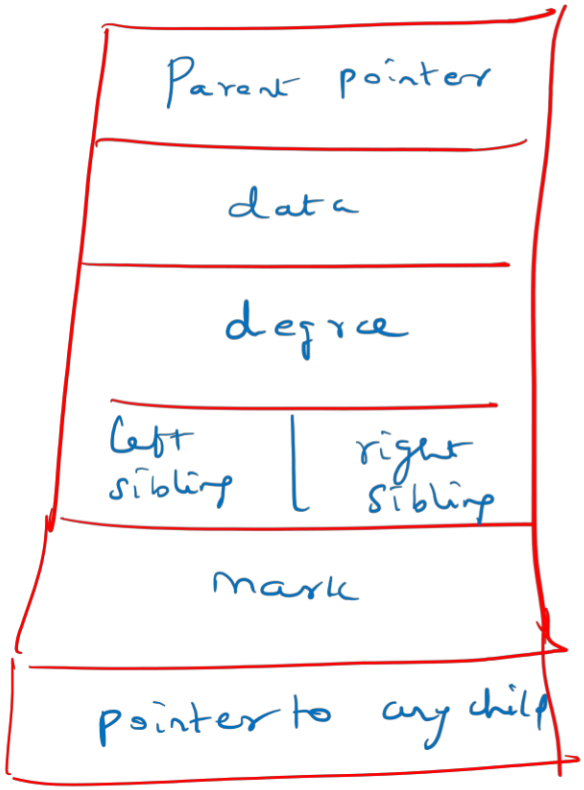
* No ascending order in terms of degree
* Two trees of same degree
* There is a pointer pointing to the minimum element (6)

Fibonacci Heap



Fibonacci Heap

Structure of a node



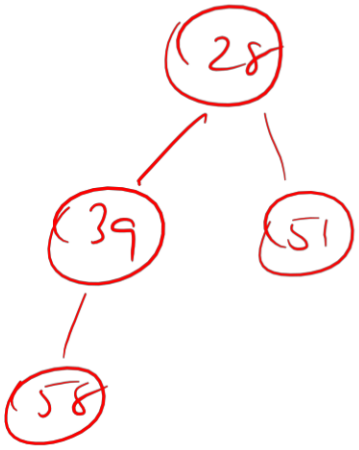
if mark(x) = 1, it means x has lost one of its child

By default, mark of all node is 0

Fibonacci Heap

In Fib heap, each tree of order ‘n’ has atleast F_{n+2} nodes in it

Eg:



Order $n = 2$ (degree)

$F_{2+2} = F_4$ nodes

Fibonacci Heap

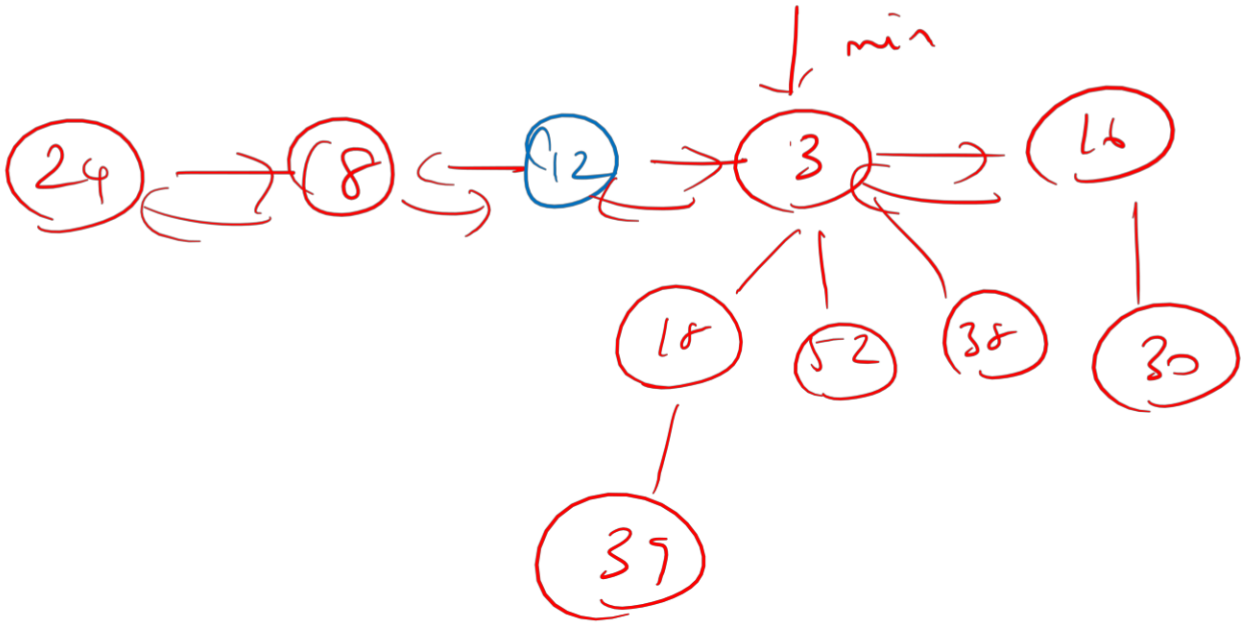
Operations:

1. Insert
2. Create
3. Find_Min
4. Union

Fibonacci Heap

(I) Insert 12 in the previous heap

* Always add to the left of min pointer
* update min (if required)



Fibonacci Heap

(2) Find_Min

Always a pointer points to the minimum element.

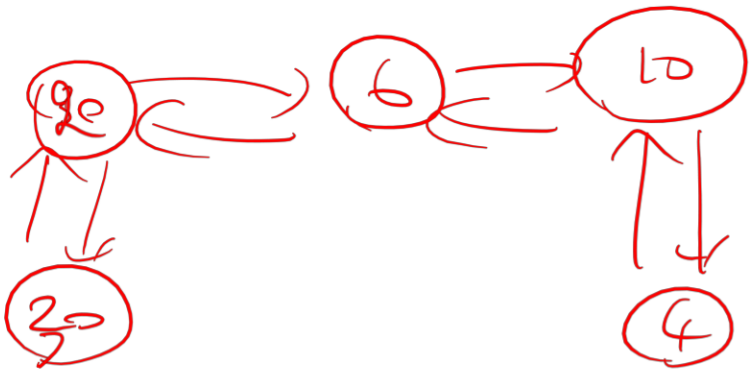
Here it is (3).

Time complexity O(1)

Fibonacci Heap

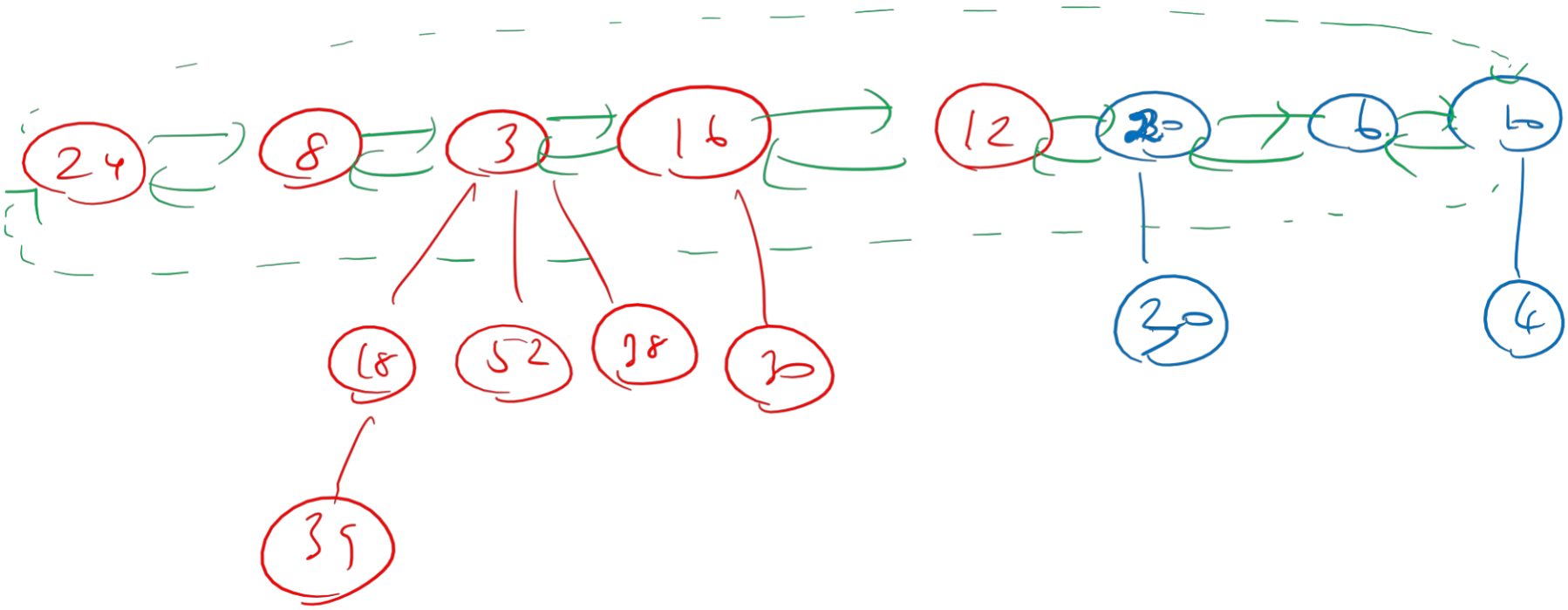
(3) union

Add the following in the previous heap



just concatenate & rearrange the pointers.

Fibonacci Heap



Fibonacci Heap

Operations	Binomial Heap	Fibonacci Heap
Find-Min	O(1)	O(1)
Insert	O(1)	O(1)
Merge	O(log n)	O(1)

Fib heap is the least complex advanced data structure

Heap in Huffman coding

Heap in Huffman coding

- Huffman coding is a lossless data compression algorithm.
- The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

Heap in Huffman coding

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Heap in Huffman coding

This builds a tree in bottom up manner

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

Steps to build Huffman Tree:

1. Create a leaf node for each unique character and build a min heap of all leaf nodes
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps 2 and 3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Heap in Huffman coding

Applications of Huffman Coding:

- They are used for transmitting text.
- They are used by conventional compression formats like PKZIP, GZIP, etc.
- Multimedia codecs like JPEG, PNG, and MP3 use Huffman encoding

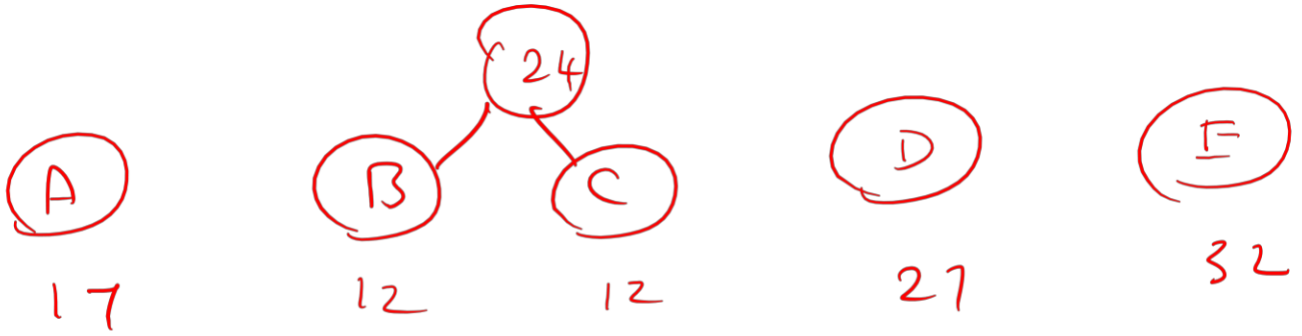
Heap in Huffman coding

Characters :	A	B	C	D	E
Frequency :	17	12	12	27	32

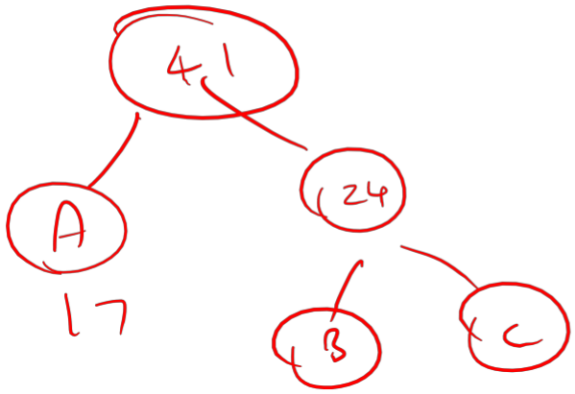
Use min-heap:

Lowest 2 keys B & C. Make it children & parent is sum of these values

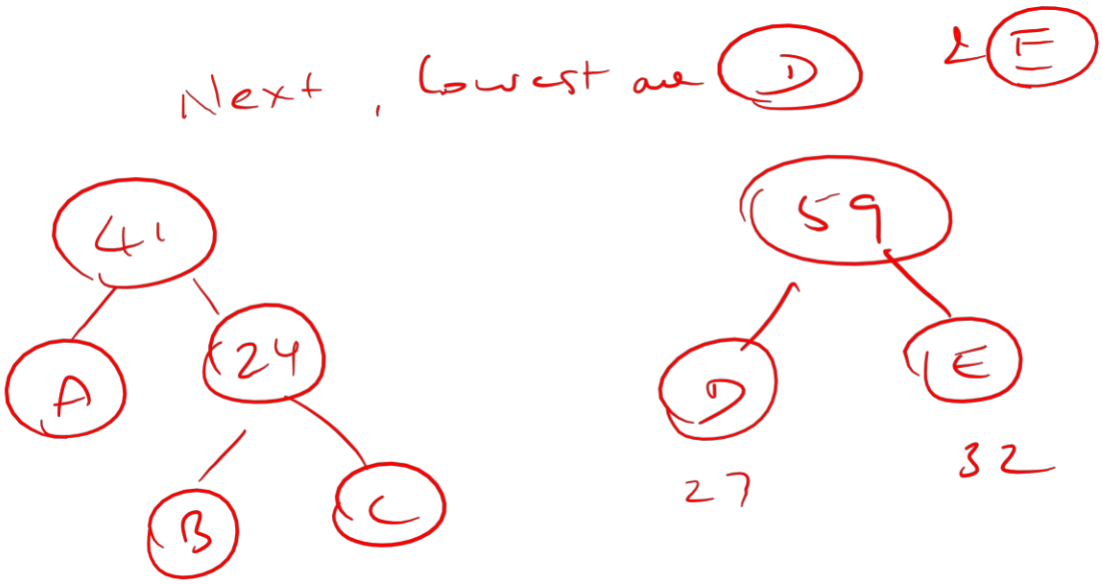
Heap in Huffman coding

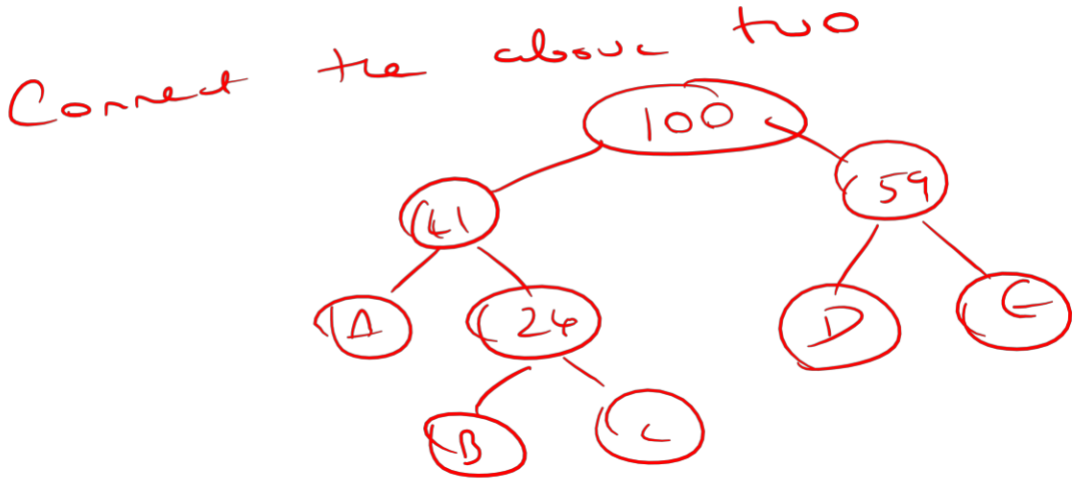


Next lowest value is A & 24



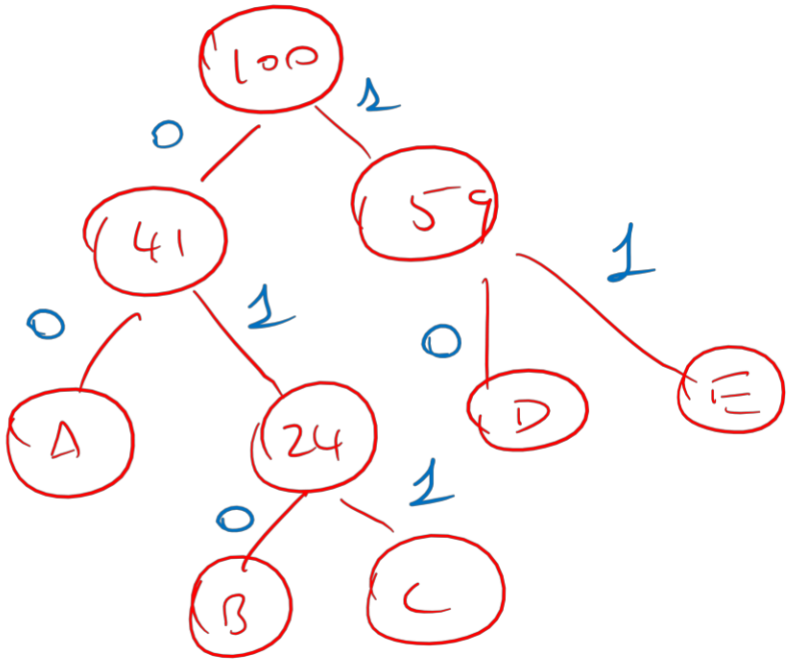
Heap in Huffman coding





Heap in Huffman coding

Assign 0 to left & 1 to right



Code for each letter (from root to that node)

A : 00

B : 010

C : 011

D : 10

E : 11

Heap in Huffman coding

Encoding (compression)

Text to compress : E A E B A E C D E A

Encoder :

A → 00
B → 010
C → 011
D → 10
E → 11

Huffman code : 11 00 11 010 00 11

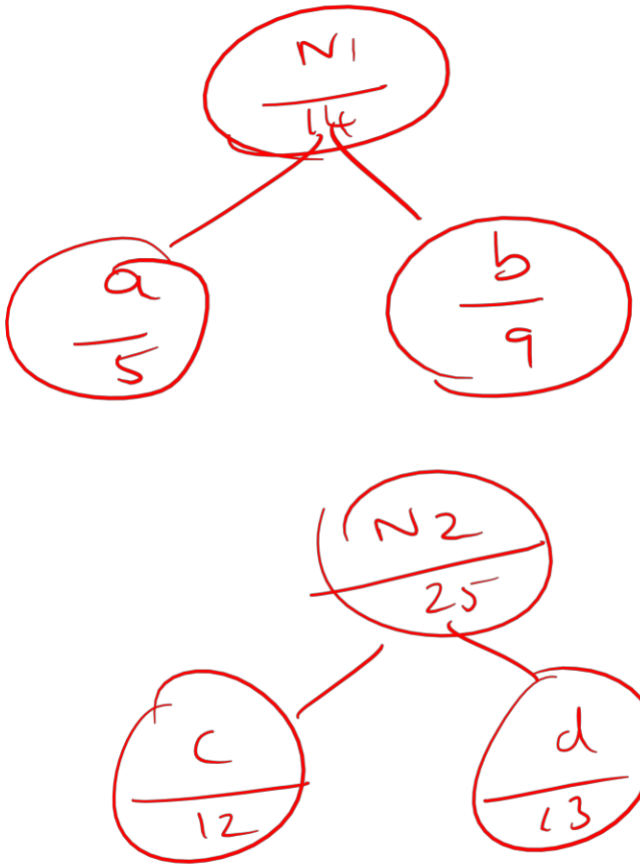
(compressed file) 011 10 11 00

Heap in Huffman coding

Exercise :-

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

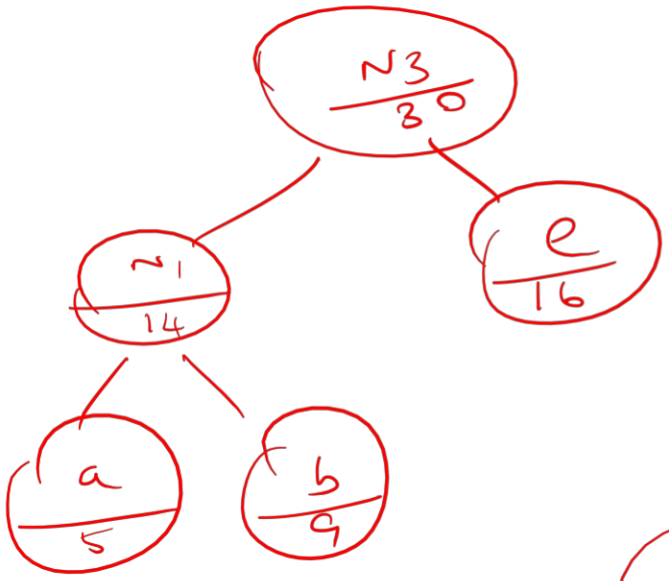
Heap in Huffman coding



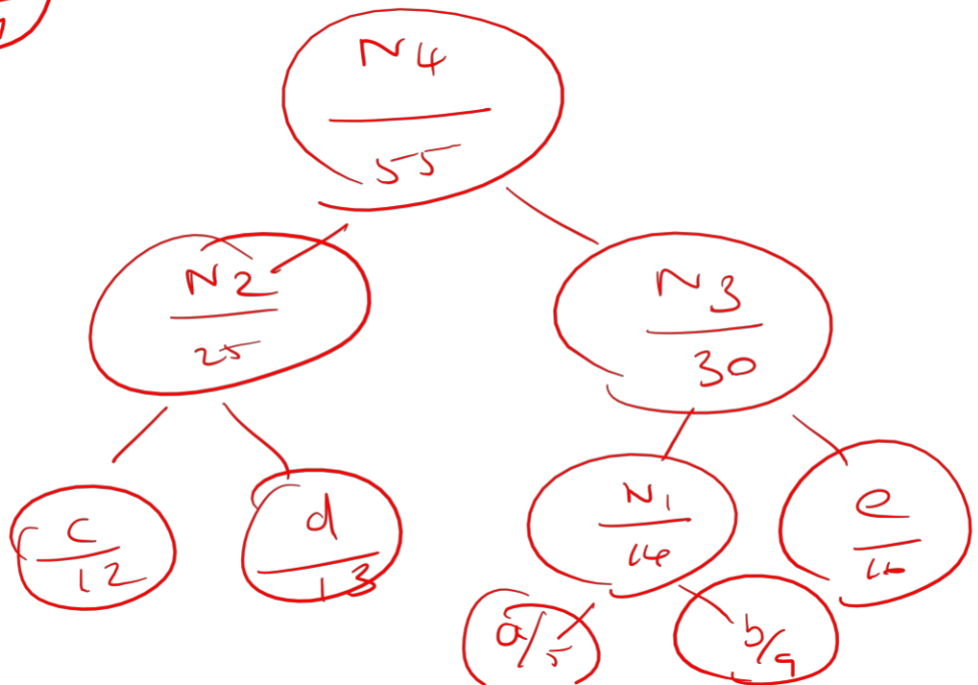
char	freq
N_1	14
c	12 ✓
d	13 ✓
e	16
f	45

char	freq
N_1	14 ✓
N_2	25
e	16 ✓
f	45

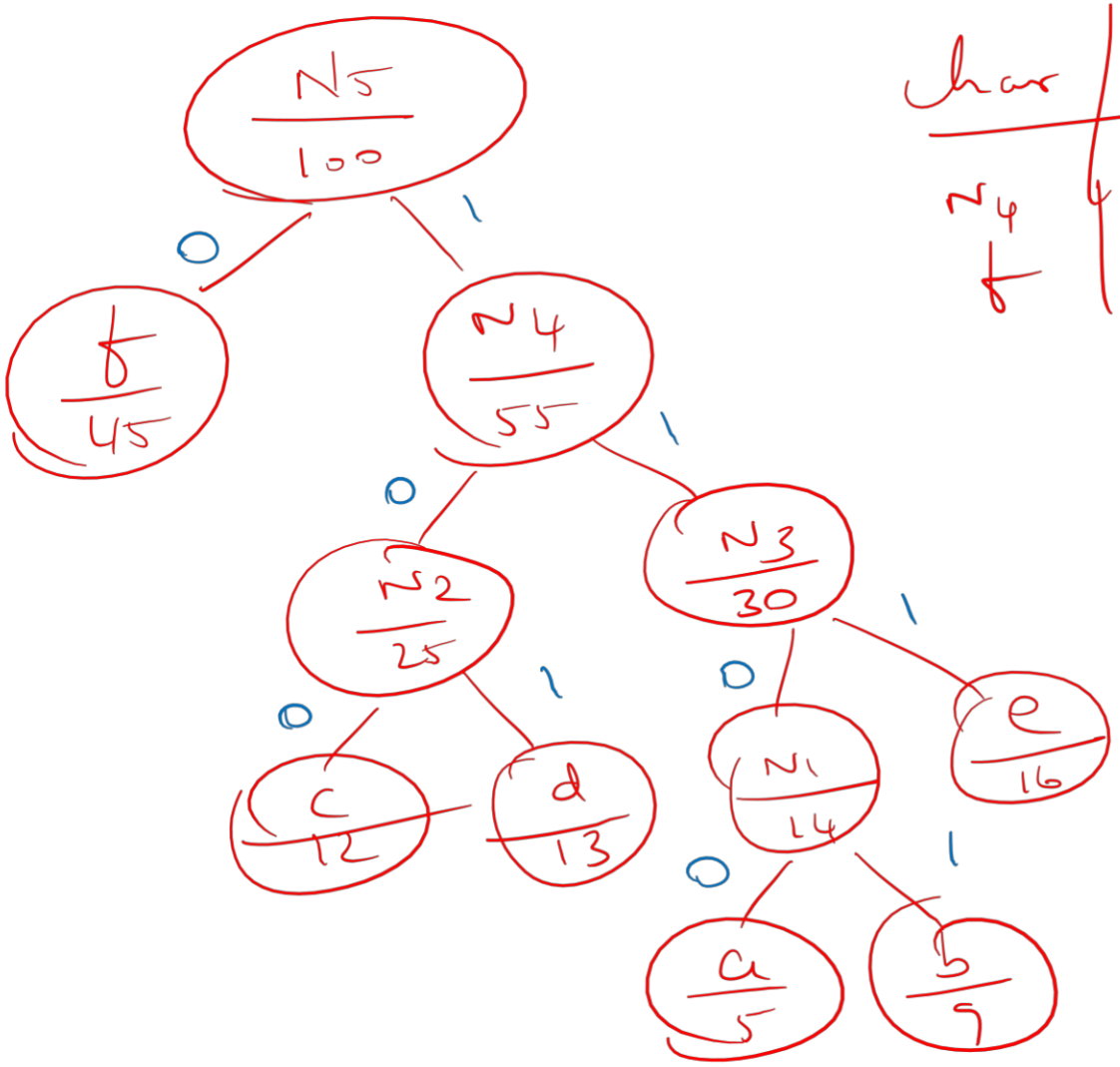
Heap in Huffman coding



char	freq
N_2	25
N_3	30
t	45



Heap in Huffman coding



char	freq
N5	55
N4	45
f	

character	code
a	1100
b	1101
c	100
d	101
e	111
f	0

Hashing

Searching Techniques

Searching Techniques

- In data structures,
  - There are several searching techniques like linear search, binary search, search trees etc.
  - In these techniques, time taken to search any particular element depends on the total number of elements.

Searching Techniques

Example

- **Linear Search** takes O(n) time to perform the search in unsorted arrays consisting of n elements.
- **Binary Search** takes O(logn) time to perform the search in sorted arrays consisting of n elements.
- It takes O(logn) time to perform the search in **Binary Search Tree** consisting of n elements.

Drawback

- The main drawback of these techniques is-
  - As the number of elements increases, time taken to perform the search also increases.
  - This becomes problematic when total number of elements become too large.

Hashing in Data Structure

- Hashing is a well-known technique to search any particular element among several elements.
- It minimizes the number of comparisons while performing the search.

Advantage

Unlike other searching techniques,

- Hashing is extremely efficient.
- The time taken by it to perform the search does not depend upon the total number of elements.
- It completes the search with constant time complexity O(1).

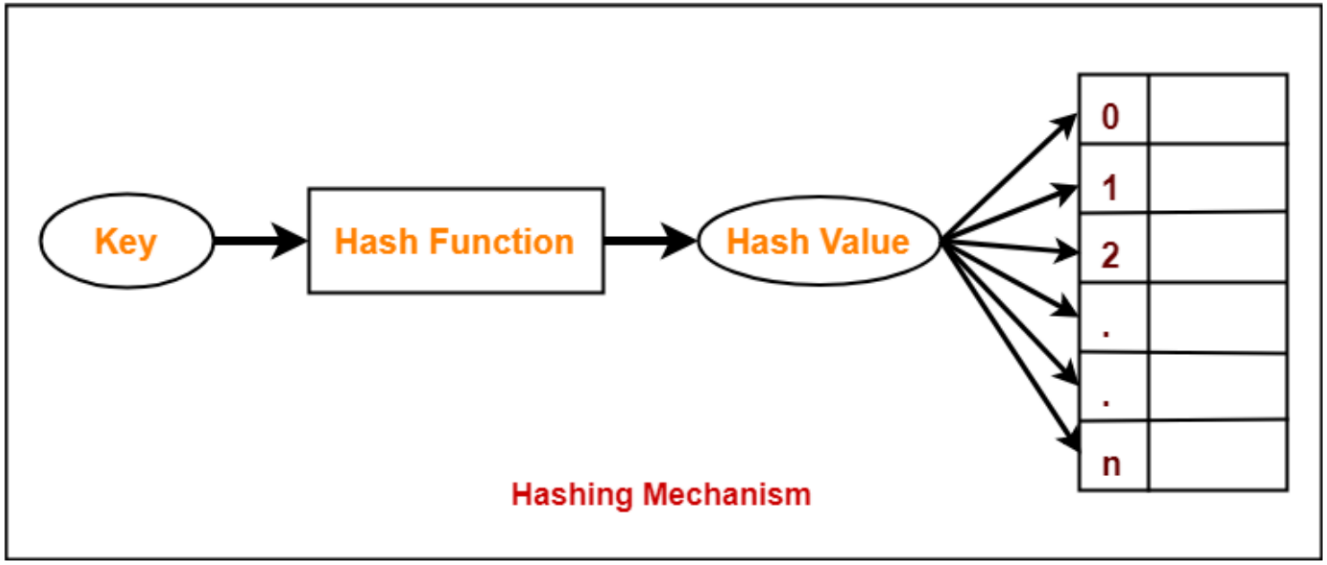
Hashing Mechanism

- An array data structure called as **Hash table** is used to store the data items.
- Based on the hash key value, data items are inserted into the hash table.

Hash Key Value

- Hash key value is a special value that serves as an index for a data item.
- It indicates where the data item should be stored in the hash table.
- Hash key value is generated using a hash function.

Hash Key Value



Hash Function

- Hash function is a function that maps any big number or string to a small integer value.
- Hash function takes the data item as an input and returns a small integer value as an output.
- The small integer value is called as a hash value.
- Hash value of the data item is then used as an index for storing it into the hash table.

Types of Hash Functions

- There are various types of hash functions available such as-
  - Mid Square Hash Function
  - Division Hash Function
  - Folding Hash Function etc
- It depends on the user which hash function he wants to use.

Hash Function- Division Method

- h(k)=k mod tablesize or arraysize

Extendible Hashing

- Extendible hashing is a dynamic hashing technique used in database systems to handle hash collisions and efficiently manage growing datasets.
- Unlike static hashing, where the size of the hash table is fixed, extendible hashing allows the table to grow as more data is added, ensuring efficient access and insertion times.

Extendible Hashing

- **Key Concepts:**

1. **Hashing Function**: A hash function maps keys to a hash table using binary values. In extendible hashing, the hash function generates binary numbers, and the directory uses these bits to store and retrieve data.

Extendible Hashing

- **Key Concepts:**

2. Global Depth and Local Depth:

1. **Global Depth**: Refers to the number of bits used from the hash to index into the directory. It controls the overall structure.
2. **Local Depth**: Refers to the number of bits used to differentiate between records within a particular bucket.

Extendible Hashing

•**Directory**: A table that holds pointers to buckets. The size of the directory is *2^Global Depth*, meaning it can point to that many buckets. The directory grows by doubling its size when necessary.

•**Buckets**: Each bucket holds a fixed number of records (data entries). When a bucket overflows, it is split, and the local depth of the bucket increases. If the local depth equals the global depth, the directory size must be doubled.

•**Doubling the Directory**: When a bucket is split and the local depth exceeds the global depth, the directory must expand. The global depth increases by 1, and the directory is doubled. The new directory entries point to the existing buckets until further splitting occurs.

Extendible Hashing

Operations in Extendible Hashing:

1. Insertion:

- The hash of the key is computed.
- The key is placed into the corresponding bucket based on the hash's first *Global Depth* bits.
- If the bucket is full, the bucket is split, and the directory may be doubled if necessary.

Extendible Hashing

2. Splitting a Bucket:

- When a bucket overflows, a new bucket is created, and the keys in the original bucket are rehashed.
- If the local depth of the bucket after splitting exceeds the global depth, the directory size is doubled.

Extendible Hashing

3. Search:

- The hash function is applied to the key to obtain its hash.
- The directory is used to locate the appropriate bucket by using the *Global Depth* bits from the hash.
- The search proceeds in the bucket to find the desired key.

Extendible Hashing

Terminologies:

Directories:

The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.

The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.

Buckets:

They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.

Extendible Hashing

Global Depth:

It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.

Local Depth:

It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.

Extendible Hashing

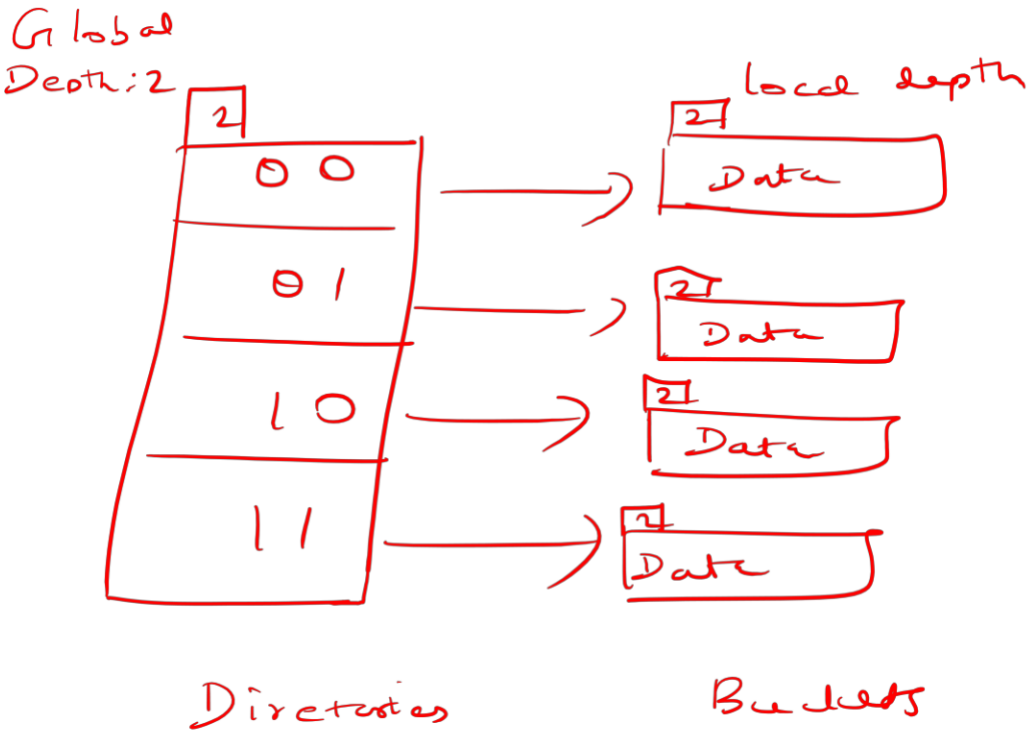
Bucket Splitting:

When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.

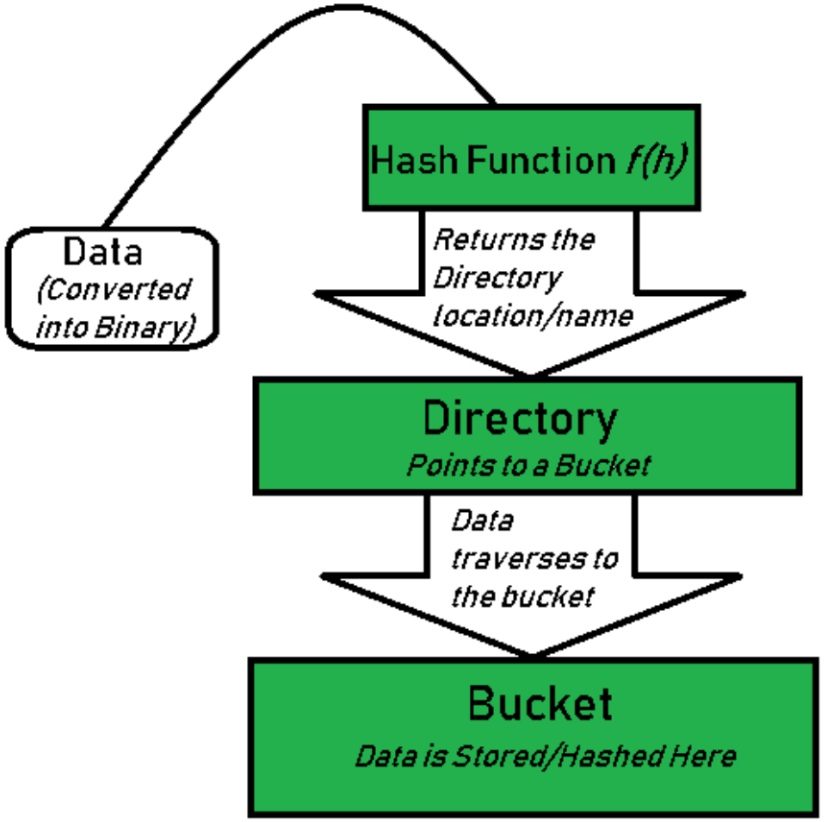
Directory Expansion:

Directory Expansion takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

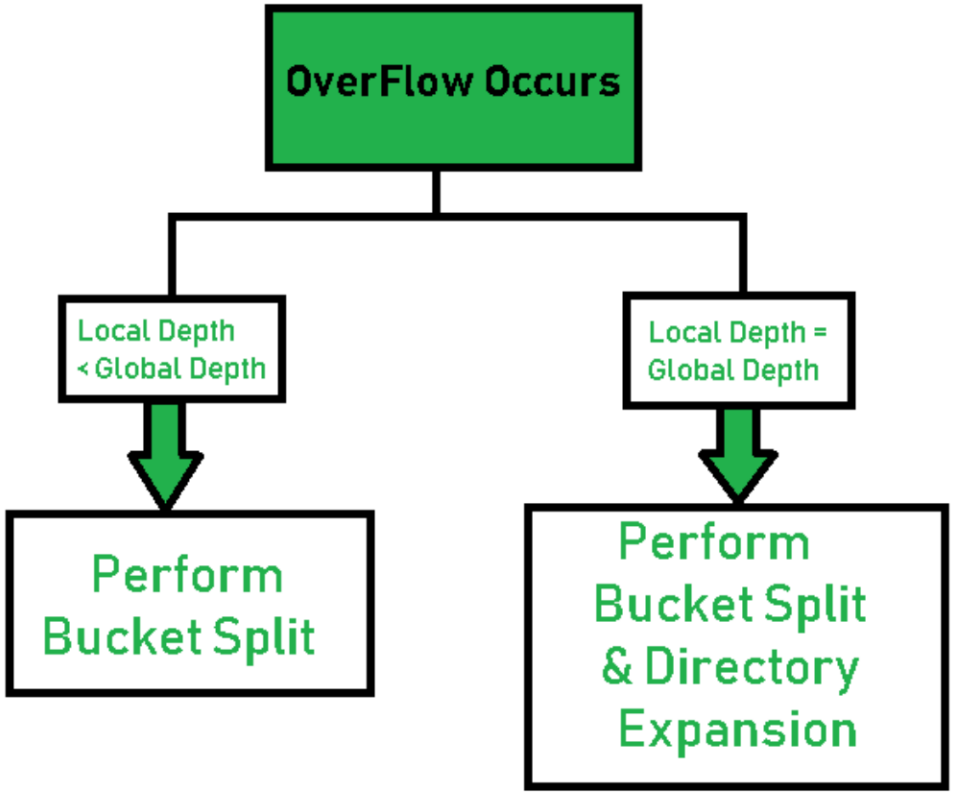
Extendible Hashing



Extendible Hashing



Extendible Hashing



Extendible Hashing

Hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**

Bucket Size: 3 (Assume)

Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

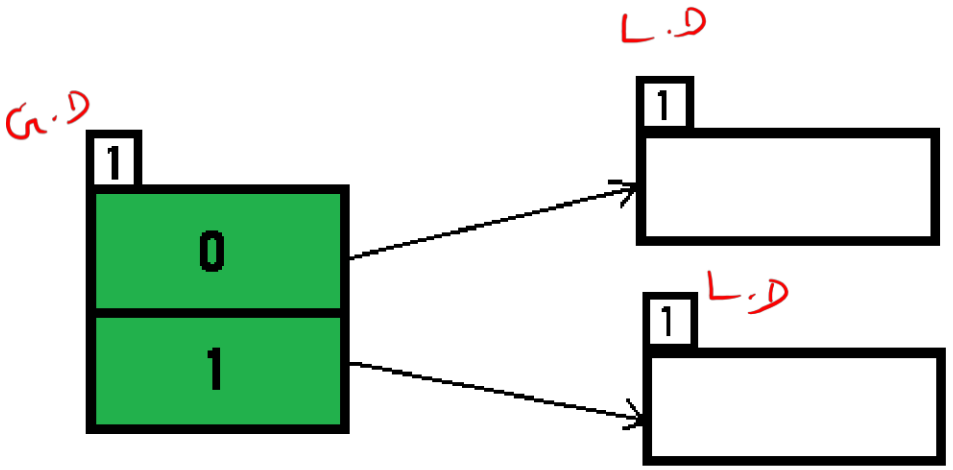
Extendible Hashing

Solution: First, calculate the binary forms of each of the given numbers.

16- 10000
4- 00100
6- 00110
22- 10110
24- 11000
10- 01010
31- 11111
7- 00111
9- 01001
20- 10100
26- 11010

Extendible Hashing

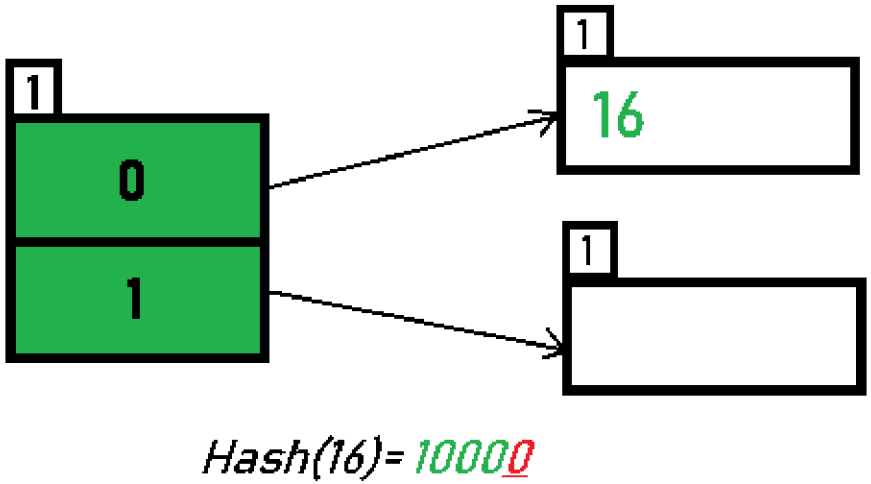
Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



Extendible Hashing

Inserting 16:

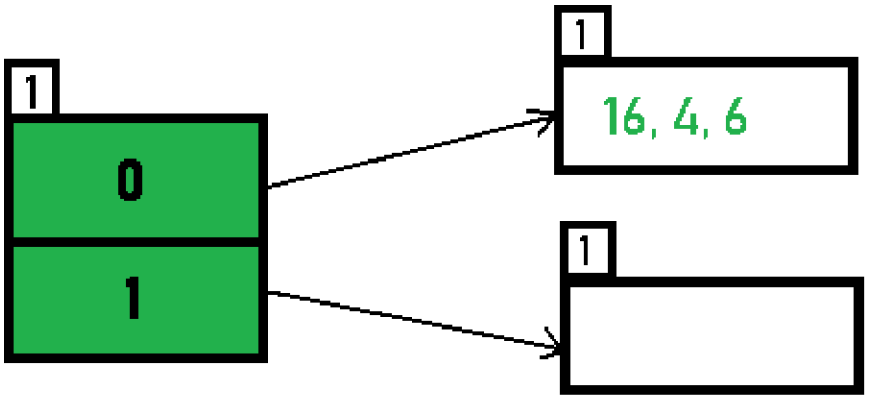
The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 10000 which is 0. Hence, 16 is mapped to the directory with id=0.



Extendible Hashing

Inserting 4 and 6:

Both 4(100) and 6(110)have 0 in their LSB. Hence, they are hashed as follows:

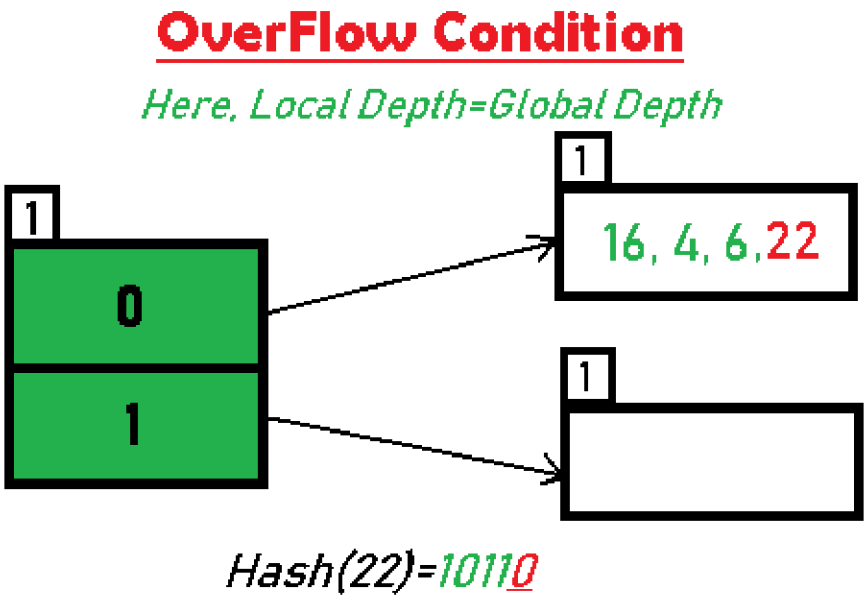


Hash(4)=100
Hash(6)=110

Extendible Hashing

Inserting 22:

The binary form of 22 is 10110. Its LSB is 0. The bucket pointed by directory 0 is already full (as the bucket size is 3). Hence, Overflow occurs.

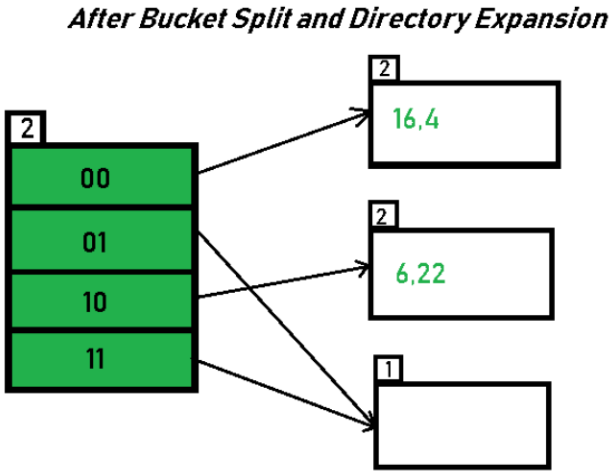


Extendible Hashing

Since Local Depth = Global Depth, the bucket splits and directory expansion takes place.

Also, rehashing of numbers present in the overflowing bucket takes place after the split.

And, since the global depth is incremented by 1, now, the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(10000),4(100),6(110),22(10110)] and the corresponding buckets local depth is also increases by 1



Extendible Hashing

*Notice that the bucket which was underflow has remained untouched.

But, since the number of directories has doubled, we now have 2 directories 01 and 11 pointing to the same bucket.

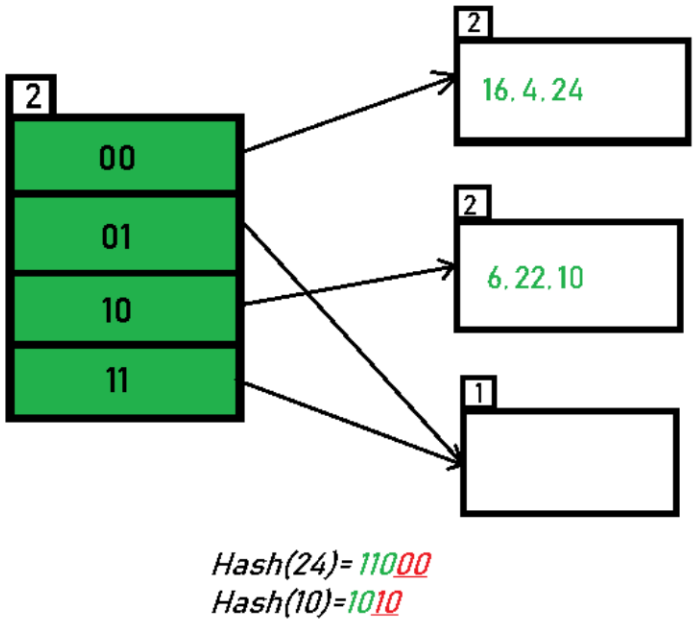
This is because the local-depth of the bucket has remained 1.

And, any bucket having a local depth less than the global depth is pointed-to by more than one directories.

Extendible Hashing

Inserting 24 and 10:

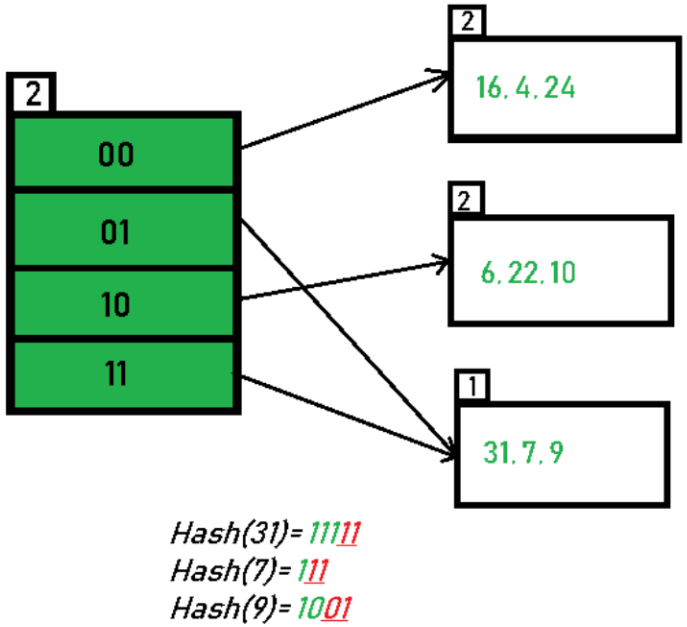
24(11000) and 10 (1010) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



Extendible Hashing

Inserting 31,7,9:

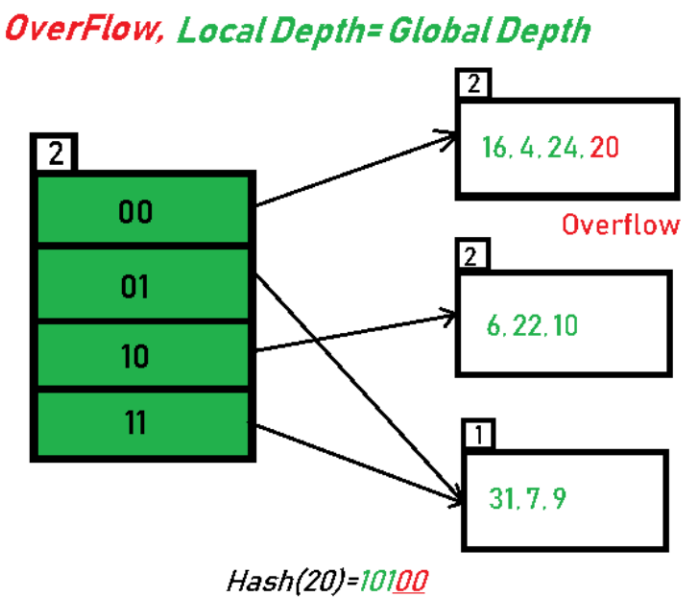
All of these elements[31(11111), 7(111), 9(1001)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



Extendible Hashing

Inserting 20:

Insertion of data element 20 (10100) will again cause the overflow problem.



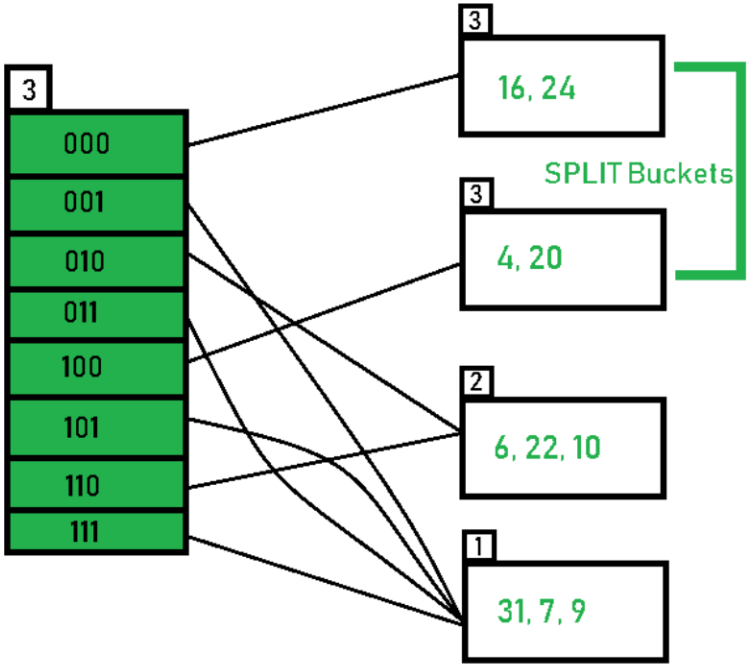
Extendible Hashing

20 is inserted in bucket pointed out by 00.

Since the local depth of the bucket = global-depth, directory expansion (doubling) takes place along with bucket splitting. $(2^3 = 8)$

Elements present in overflowing bucket are rehashed with the new global depth (3) and he corresponding Local depth is also increased by 1

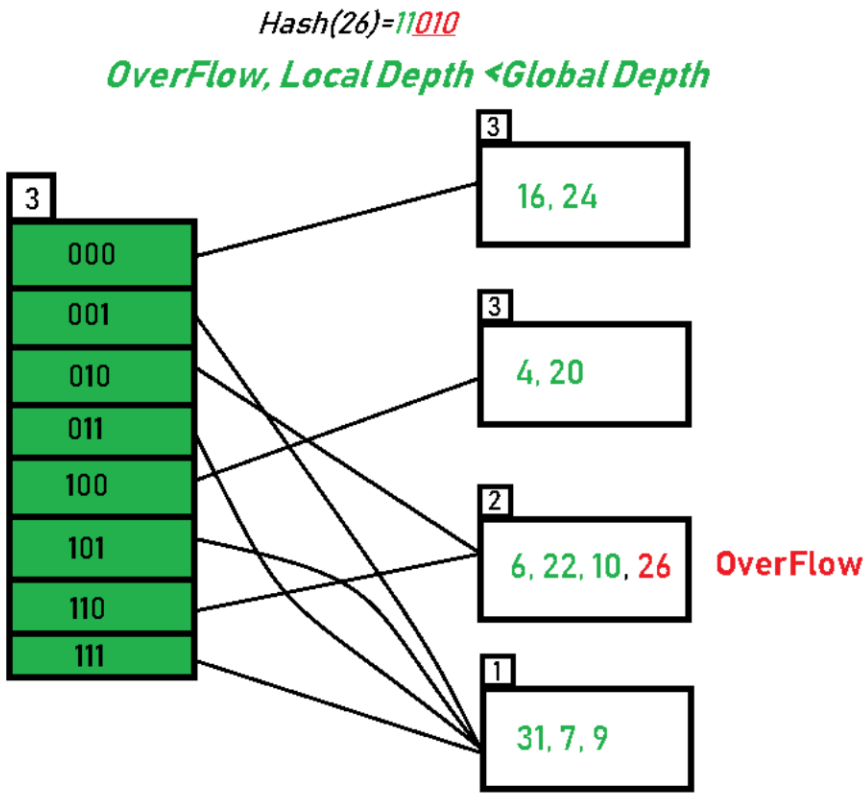
Now, the new Hash table looks like this:



Extendible Hashing

Inserting 26:

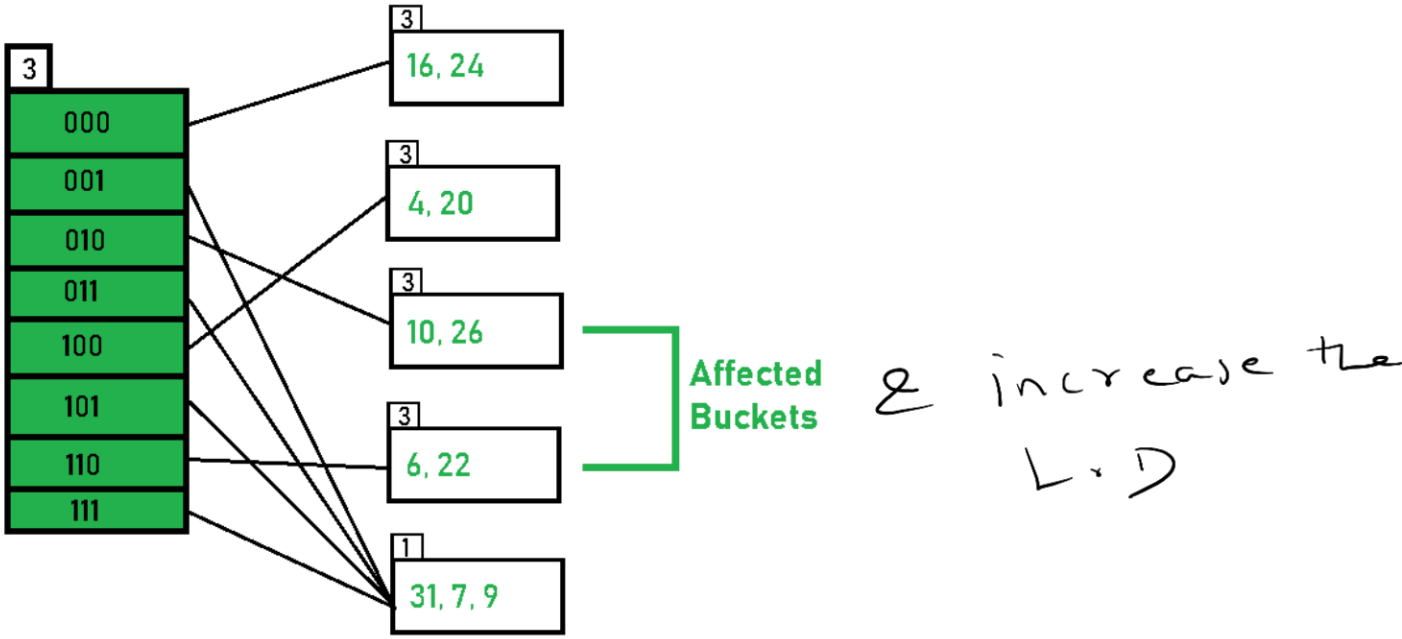
Global depth is 3. Hence, 3 LSBs of 26(11010) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



Extendible Hashing

The bucket overflows, since the local depth of bucket < Global depth (2<3), directories are not doubled but, only the bucket is split and elements are rehashed.

Finally, the output of hashing the given list of numbers is obtained.



Extendible Hashing

Example 2 :-

Assume

Each bucket holds 2 keys

Hash function generates binary representation of keys

Directory starts with a Global Depth

G.D = 1

Extendible Hashing

Keys to be inserted:

5, 12, 7, 15, 9, 4, 6, 10

1. Start with an empty directory of $2^{G.D}$ where $G.D = 1$

The directory has 2 entries, each pointing to a bucket

Extendible Hashing

Binary Representation

5 - 101

12 - 1100

7 - 111

15 - 1111

9 - 1001

4 - 100

6 - 110

10 - 1010