Computing Laboratory

Heaps

Malay Bhattacharyya

Associate Professor

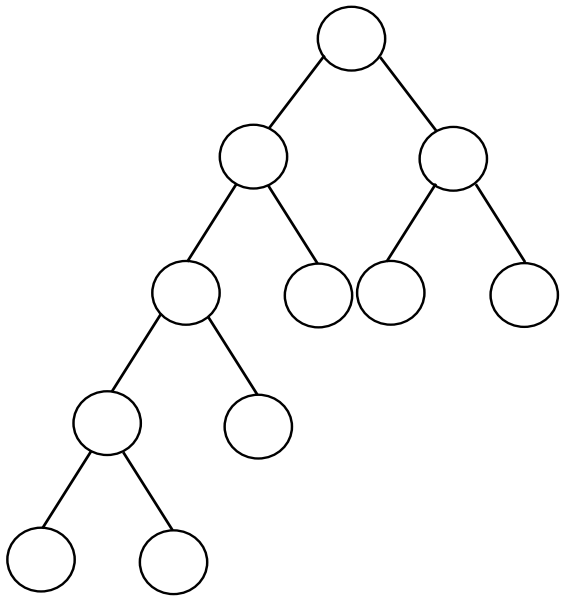
MIU, CAIML, TIH
Indian Statistical Institute, Kolkata

November, 2023

Full binary trees

Definition (Full Binary Tree)

A binary tree is said to be full binary tree (or 2-tree) if each node in the tree has either no child or exactly two children.



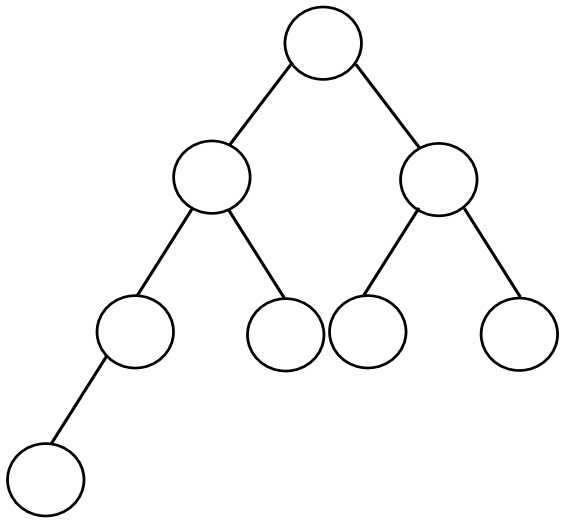
Note: An extended binary tree becomes a full binary tree.

Complete binary trees

Definition (Complete Binary Tree)

A binary tree is said to be complete binary tree if it satisfies both of the following:

- Every level, except possibly the last, is completely filled.
- All nodes appear as far left as possible.



Complete binary trees

Some properties:

1. A complete binary tree of height h comprises at least 2^h and at most $2^{h+1} - 1$ nodes.
2. The height of a complete binary tree having exactly n nodes is $\lceil \log_2(n+1) \rceil$.
3. The maximum possible number of parents with absent children in a complete binary tree having n nodes is $(n+1)$.
4. The number of internal nodes in a complete binary tree having n nodes is $\lfloor n/2 \rfloor$.

Binary heap

Definition (Binary Heap)

A complete binary tree is said to be binary heap (or simply a heap) if the data items it contains (in the domain) are arranged following a heap property.

Definition (Max-heap property)

The data item in each parent node is more than or equal to the data items in its children nodes.

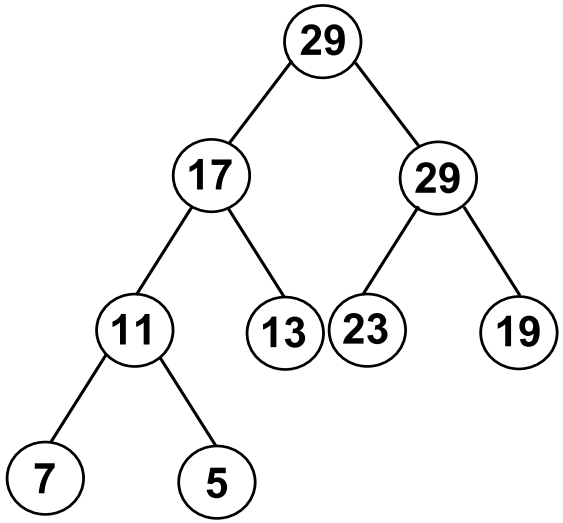
Definition (Min-heap property)

The data item in each parent node is less than or equal to the data items in its children nodes.

Max-heap

Definition (Max-heap)

A max-heap is a binary heap that satisfies the max-heap property.

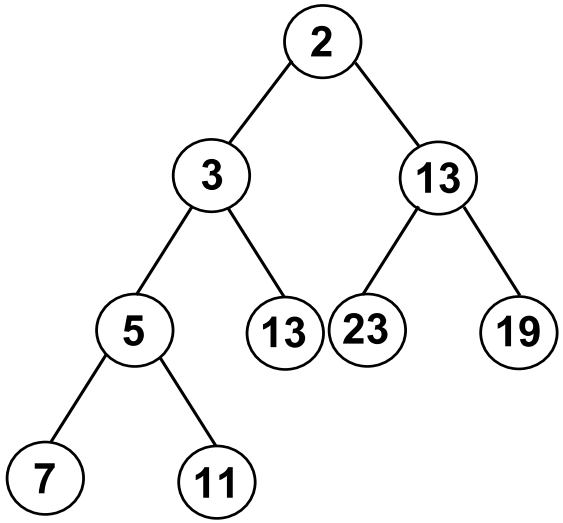


Note: The maximum data item is at the root node.

Min-heap

Definition (Min-heap)

A min-heap is a binary heap that satisfies the min-heap property.



Note: The minimum data item is at the root node.

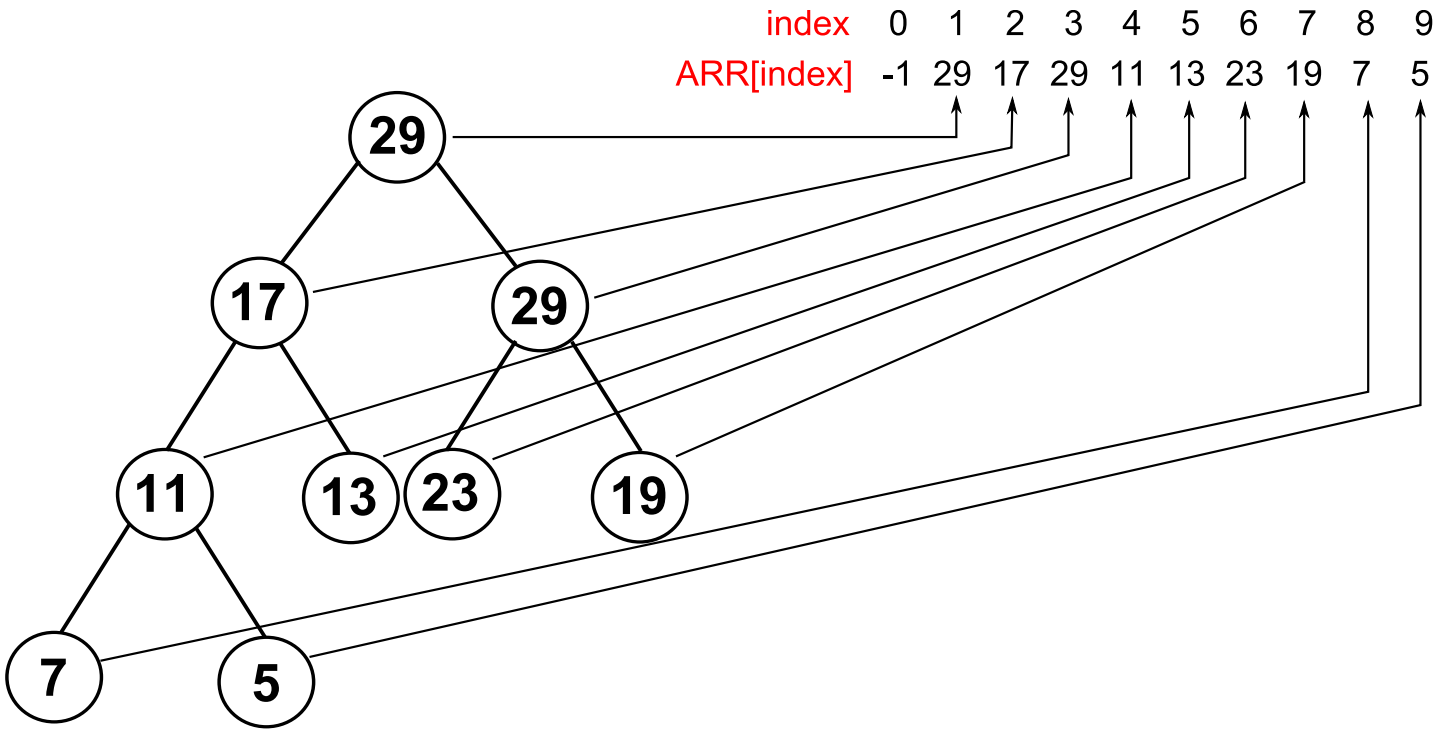
Operational efficiency on heaps

Order of growth of worst-case running time for the various implementations of priority queues is provided below.

Data Structure	**Insertion**	**Deletion of Maximum/Minimum**
Ordered Array	N	1
Unordered Array	1	N
Heap	$\log N$	$\log N$
Impossible	1	1

Heaps as arrays

The data items traversed from a heap in level-order can be kept in an array.



Heaps as arrays

Zero-based Indexing	**One-based Indexing**
1. Left child of the node at index i is at $2i + 1$.	1. Left child of the node at index i is at $2i$.
2. Right child of the node at index i is at $2(i + 1)$.	2. Right child of the node at index i is at $2i + 1$.
3. Parent of the node at index i is at $\lfloor (i - 1)/2 \rfloor$.	3. Parent of the node at index i is at $\lfloor i/2 \rfloor$.

Operations on heaps – Insertion

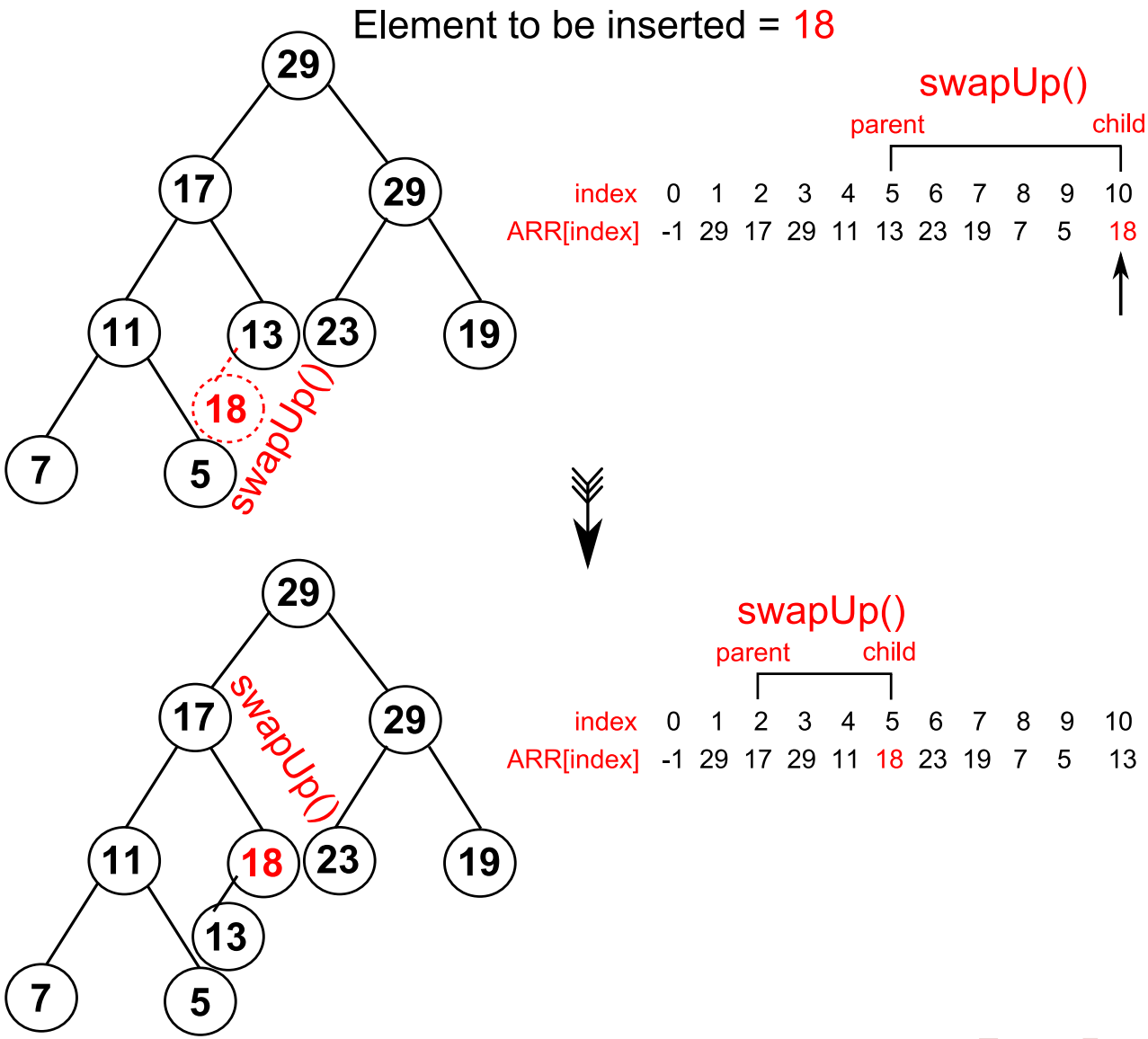
MAX-HEAP:

1. Add the data item (to be inserted) at the end of the heap as a new node.
2. If the new node is greater than its parent, then traverse up to fix the violated heap property.

MIN-HEAP:

1. Add the data item (to be inserted) at the end of the heap as a new node.
2. If the new node is less than its parent, then traverse up to fix the violated heap property.

Operations on heaps – Insertion



Operations on heaps – Deletion of maximum/minimum

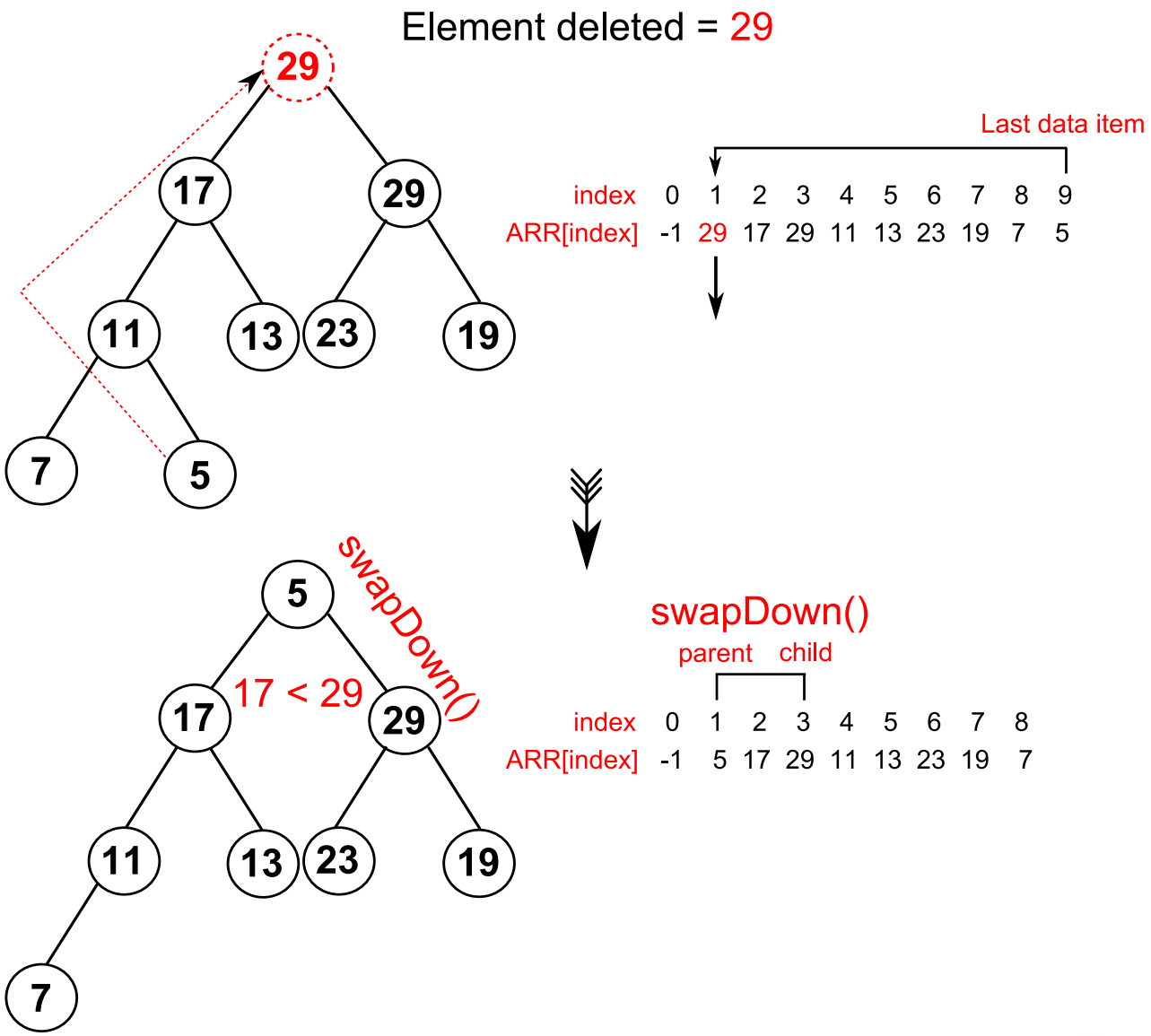
MAX-HEAP:

1. Delete the data item from the top of the heap.
2. Take the last element to the root of the heap.
3. Identify the greater data item among both the children of the new root node and swap it with the root.

MIN-HEAP:

1. Delete the data item from the top of the heap.
2. Take the last element to the root of the heap.
3. Identify the greater data item among both the children of the new root node and swap it with the root.

Operations on heaps – Deletion of maximum



Implementation of heaps

We can implement a max-heap in the form of a maximum priority queue with a generic array as follows.

```
typedef struct{
    size_t element_size; // Generic implementation
    unsigned int num_allocated, num_used; // Keep track of the size
    void *array; // Using one-based indexing
    int (*comparator)(void *, int, int); // Returns -ve, 0, or +ve
}HEAP;
```

Initialization

```
void initHeap(HEAP *h, size_t element_size,
                  int (*comparator)(void *, int, int)){ // Continued
    h->element_size = element_size;
    // Allocated size
    h->num_allocated = 10;
    // Current size
    h->num_used = 0;
    if(NULL == (h->array = Malloc(h->num_allocated, element_size))){
        ERR_MSG("initHeap: Out of memory");
        exit(-1);
    }
    h->comparator = comparator;
    return;
}
```

Insertion – Main routine

```
void insert(HEAP *h, void *x){
    // Make sure there is space for another element
    if(h->num_used + 1 == h->num_allocated){
        h->num_allocated *= 2;
        if(NULL == (h->array = realloc(h->array,
                h->num_allocated * h->element_size))){ // Continued
            ERR_MSG("insert: Out of memory");
            exit(-1);
        }
    }
    // Insert element at the end
    h->num_used++;
    memcpy((char *) h->array + h->num_used * h->element_size,
                x, h->element_size); // Continued
    // Restore the heap property (max-heap)
    swapUp(h, h->num_used);
    return;
}
```

Insertion – Auxiliary routine

```
static void swapUp(HEAP *h, int k){
    // Repeat until the parent is not the root
    while (k > 1 && (h->comparator(h->array, k/2, k) < 0)){
        // Swap child at position k with the parent
        swap(h, k, k/2);
        // Move up to the parent level
        k = k/2;
    }
    return;
}
```

Deletion of maximum – Main routine

```
void deleteMax(HEAP *h, void *max){
    // Max is at the root (index 1)
    memcpy(max, h->array + h->element_size, h->element_size);
    // Copy the last element to root
    memcpy(h->array + h->element_size, h->array + h->num_used *
                h->element_size, h->element_size); // Continued
    h->num_used--;
    // Restore the heap property (max-heap)
    swapDown(h, 1);
    return;
}
```

Note: deleteMin(HEAP *, void *) can be easily implemented following this.

Deletion of maximum – Auxiliary routine

```
static void swapDown(HEAP *h, int k){
    // Repeat until the left child (2k) is within the boundary
    while (2*k <= h->num_used){
        int j = 2*k; // Left child (2k)
        // Choose the child with larger key
        if(j < h->num_used && (h->comparator(h->array, j, j+1) < 0))
            j++; // Right child (larger key is at 2k+1)
        if(h->comparator(h->array, k, j) >= 0) // No swapping needed
            break;
        // Swap parent at position k with the largest child
        swap(h, k, j);
        k = j;
    }
    return;
}
```

Other functions – Swapping

```
static void swap(HEAP *h, int i, int j){
    // h->array[0] used as swapping variable in one-based indexing
    char *ip = (char *) h->array + i * h->element_size;
    char *jp = (char *) h->array + j * h->element_size;
    char *tp = (char *) h->array;
    // Memory to memory copy of the elements for swapping
    memcpy((void *) tp, (void *) ip, h->element_size);
    memcpy((void *) ip, (void *) jp, h->element_size);
    memcpy((void *) jp, (void *) tp, h->element_size);
    return;
}
```

Other functions – Comparison

```
static int compare_int(void *array, int i1, int i2){
    // Pick up the element at index i1
    int n1 = *((int *) array + i1);
    // Pick up the element at index i2
    int n2 = *((int *) array + i2);
    return (n1 - n2);
}
```

Problems – Day 12

1. Every research publication has a citation count that represents the number of times it has been referred in other research publications, thereby indicating its credit. The h-index is one of the popular citation metrics that is defined as the maximum value of h such that the given scientists has published h papers that have each been cited at least h number of times. Suppose you are given the citation counts of the research publications of a scientist in chronological order (i.e., following the order in which they were published). Write a program to compute the h-index of the scientist.

   Sample Input: 10 2 1 5 11 4 3
   Sample Output: 4

Problems – Day 12

1. Write a program that receives a stream of names of independent events and the probabilities of their occurrences in some context and return, as and when asked for, the names of those three events that have the maximum chance to co-occur. In case there is a tie, return all the possible event triplets.

   Input file format:

```
u 0.5 # Event name and probability of its occurrence
v 0.3 # As above
w 0.3 # As above
x 0.1 # As above
y 0.8 # As above
z 0.2 # As above
```

Problems – Day 12

3. Write a program that takes k sorted lists of integers or floating point numbers or strings, and merges them into a single sorted list.

   Input file format:

```
2 # Number of test cases
3 # Test case 1: Number of sorted lists
2 20 40 # List 1: Count, followed by ordered elements
6 2 4 6 8 10 12  # List 2: As above
5 5 15 25 30 35 # List 3: As above
2 # Test case 2: Number of sorted lists
2 3 10 # List 1: Count, followed by ordered elements
4 1 5 8 11  # List 2: As above
```

Problems – Day 12

4. You are given n ropes of lengths $l_1, l_2, \ldots, l_n$ respectively. The ropes need to be tied together to form one long rope. At a time, you can only tie two ropes together. Let the cost of tying two ropes together is equal to the sum of their lengths. Write a program that takes $l_1, l_2, \ldots, l_n$ as command line arguments and prints the minimum cost of joining the ropes together into a single one.

   Example:

   Let us assume $l_1 = 3, l_2 = 5, l_n = 2$. Then we have

   ```
   Cost((3+5)+2) = 8 + 10 = 18,
   Cost((3+2)+5) = 5 + 10 = 15,
   Cost((5+2)+3) = 7 + 10 = 17.
   ```

Problems – Day 12

5. Write a program with the following two functions.

   `MaxMin() - Takes a heap, returns 'MAX' if it is a max-heap and converts it into a mean-heap in linear (to the number of elements) time.`
   `MinMax() - Takes a heap, returns 'MIN' if it is a min-heap and converts it into a max-heap in linear (to the number of elements) time.`

6. Write a program that can find out the top k (provided as user input) frequently occurring elements in a given list. The list should be of generic type.