Data and File Structures Laboratory

Heaps

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

February, 2021

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.

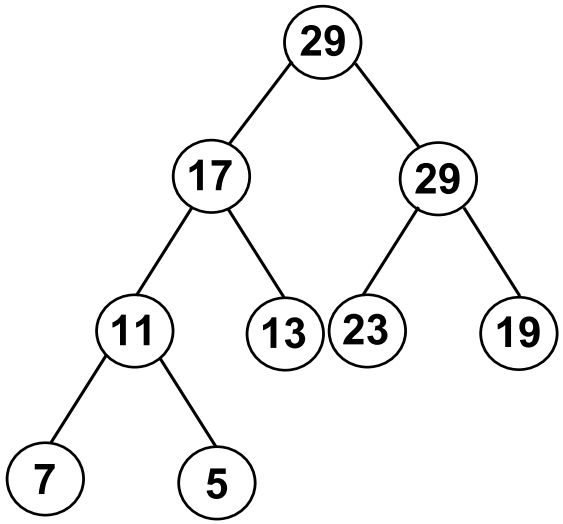
Definition (Min-heap property)

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.

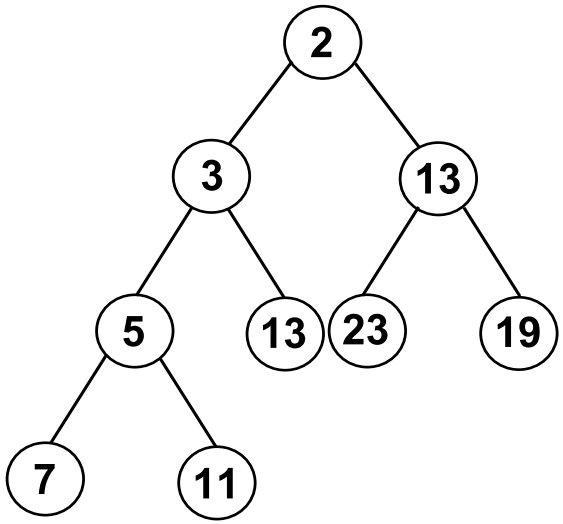


Note: The maximum data item is at the root node.

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.

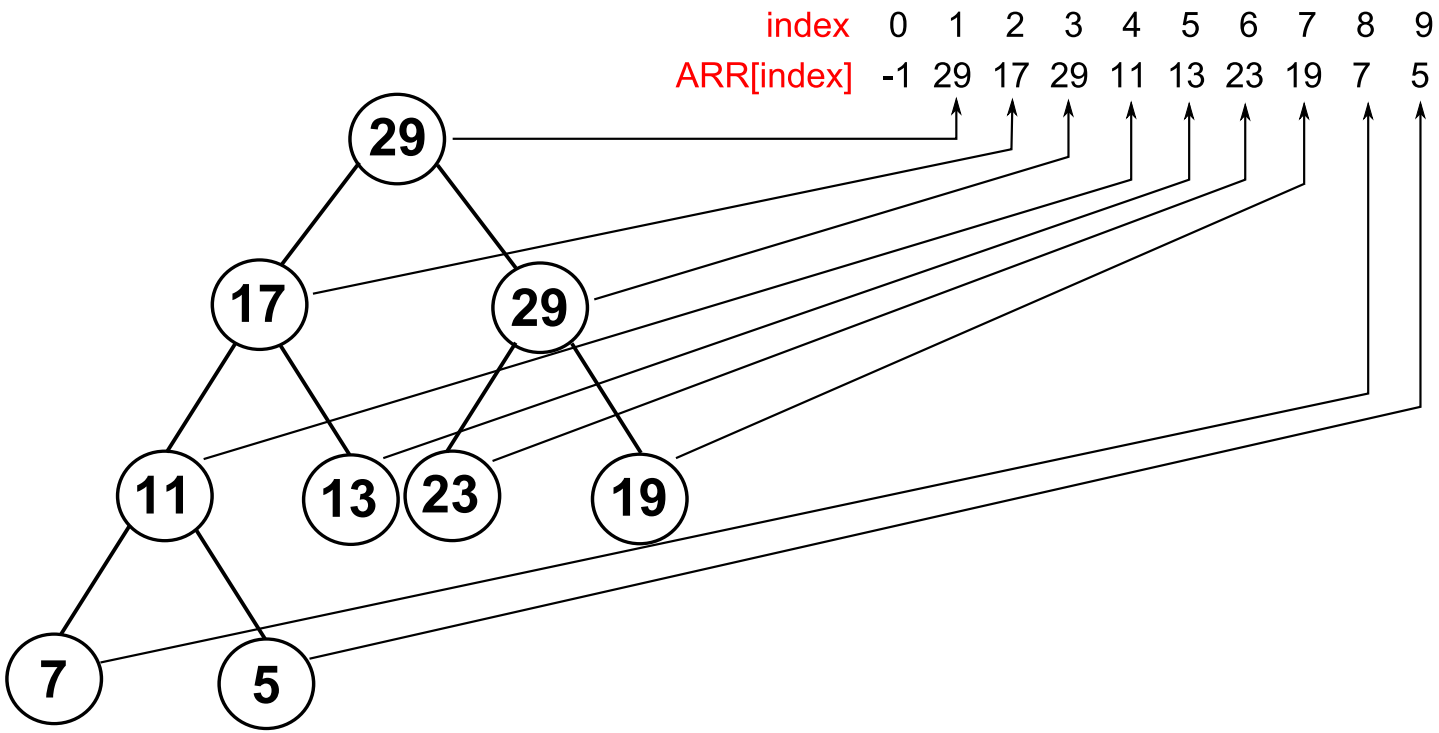
Operational efficiency on heaps

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

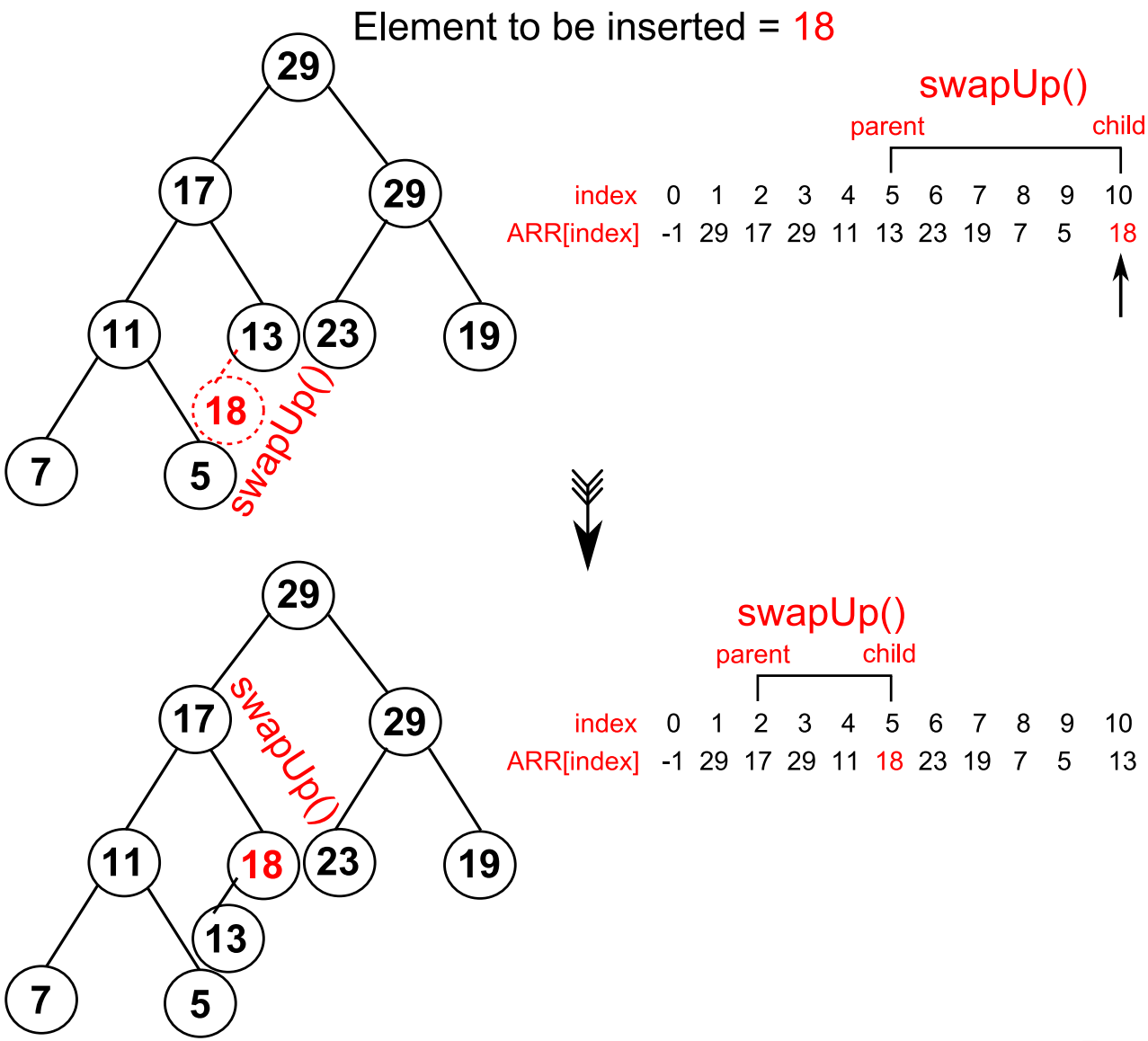
MAX-HEAP:

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

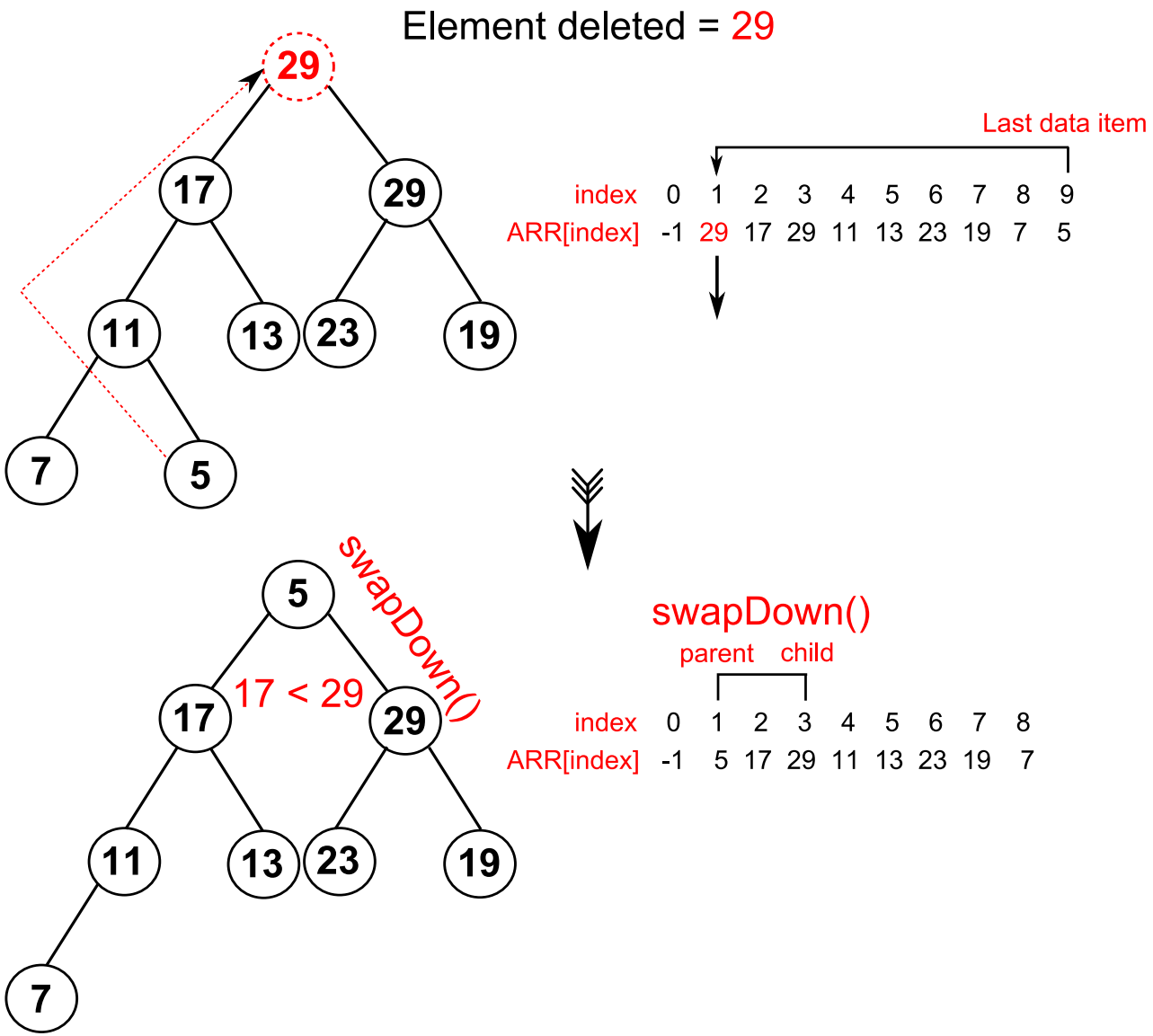
MAX-HEAP:

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 the help of a class as follows.

```
import sys

class MaxHeap:
    def __init__(self):
        ...
    def swapUp(self, k):
        ...
    def insert(self, x):
        ...
    def swapDown(self, k):
        ...
    def deleteMax(self):
        ...
    def display(self):
        ...
```

Initialization

```
def __init__(self):
    self.size = 0
    self.HEAP = [-1]
```

Insertion – Main routine

```
def insert(self, x):
    # Insert element at the end
    self.size += 1
    self.HEAP.append(x)
    # Restore the heap property (max-heap)
    swapUp(self.size)
```

Insertion – Auxiliary routine

```
def swapUp(self, k):
    # Repeat until the parent is not the root
    while(k > 1 and (self.HEAP[k//2] < self.HEAP[k])):
        # Swap child at position k with the parent
        self.HEAP[k//2], self.HEAP[k] = self.HEAP[k], self.HEAP[k//2]
        # Move up to the parent level
        k = k//2
```

Deletion of maximum – Main routine

```
def deleteMax(self):
    # Max is at the root (index 1)
    oldmax = self.HEAP[1]
    # Copy the last element to root
    self.HEAP[1] = self.HEAP[self.size]
    self.size -= 1
    # Restore the heap property (max-heap)
    self.swapDown(1)
    return oldmax
```

Note: deleteMin(self) can be easily implemented following this.

Deletion of maximum – Auxiliary routine

```
def swapDown(self, k):
    # Repeat until the left child (2k) is within the boundary
    while(2*k <= self.size):
        j = 2*k # Left child (2k)
        # Choose the child with larger key
        if(j < self.size and (self.HEAP[j] < self.HEAP[j+1])):
            j += 1 # Right child (larger key is at 2k+1)
        if(self.HEAP[k] >= self.HEAP[j]): # No swapping needed
            break
        # Swap parent at position k with the largest child
        self.HEAP[k], self.HEAP[j] = self.HEAP[j], self.HEAP[k]
        k = j
```

Printing the elements

We can print the heap as a list as follows:

```
def display_flat(self):
    print(self.HEAP[1:])
```

Alternatively, we can print the heap as a tree as follows:

```
def display(self):
    for k in range(1, (self.size//2)+1):
        print("PARENT: " + str(self.HEAP[k]), end = ", ")
        print("LEFT: " + str(self.HEAP[2*k]), end = ", ")
        if(2*k+1 < self.size): # Whether odd
            print("RIGHT: " + str(self.HEAP[2*k+1]))
        else:
            print("RIGHT: ")
```

Problems – Day 15

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 15

2. 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 15

3. 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 15

4. 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.`

5. 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.
6. Suppose the counts of COVID-19 patients across all the states in India until now are eventually distinct. Given a pair of numbers $n_1, n_2 (n_1 < n_2)$, find the total number of patients in the sates that have between n_1^{th} and n_2^{th} (inclusive range) largest counts.

Implementation of heaps in C

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

```
#define INITIAL_HEAP_SIZE 100

typedef struct{
    unsigned int num_allocated, num_used; // Keep track of the size
    int *array; // Using one-based indexing
}INT_HEAP;
```

Initialization in C

```
void initHeap(INT_HEAP *h){
    h->num_allocated = INITIAL_HEAP_SIZE; // Current size
    h->num_used = 0;
    if(NULL == (h->array = Malloc(h->num_allocated, sizeof(int))){
        ERR_MSG("initHeap: Out of memory");
        exit(-1);
    }
    return;
}
```

Insertion – Main routine in C

```
void insert(INT_HEAP *h, int 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 * sizeof(int)))){ // Continued
            ERR_MSG("insert: Out of memory");
            exit(-1);
        }
    }
    // Insert element at the end
    h->num_used++;
    h->array[h->num_used] = x;
    // Restore the heap property (max-heap)
    swapUp(h, h->num_used);
    return;
}
```

Insertion – Auxiliary routine in C

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

Note: The 0^{th} element is used as a temporary variable.

Deletion of maximum – Main routine in C

```
void deleteMax(INT_HEAP *h){
    int max;
    // Max is at the root (index 1)
    max = h->array[1];
    // Copy the last element to root
    h->array[1] = h->array[h->num_used];
    h->num_used--;
    // Restore the heap property (max-heap)
    swapDown(h, 1);
    return;
}
```

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

Deletion of maximum – Auxiliary routine in C

```
static void swapDown(INT_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->array[j] < h->array[j+1]))
            j++; // Right child (larger key is at 2k+1)
        if(h->array[k] >= h->array[j]) // No swapping needed
            break;
        // Swap parent at position k with the largest child
        h->array[0] = h->array[k];
        h->array[k] = h->array[j];
        h->array[j] = h->array[0];
        k = j;
    }
    return;
}
```

Note: The 0^{th} element is used as a temporary variable.