Data and File Structures Laboratory

Heaps

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

September, 2018

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.

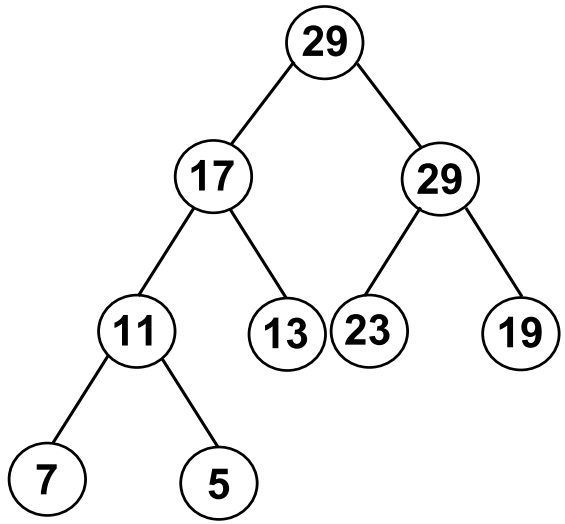
Definition (Min-heap property)

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.

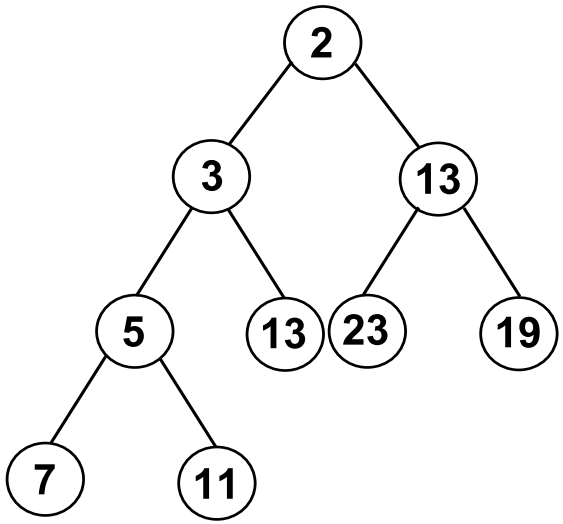


Note: The maximum data item is at the root node.

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.

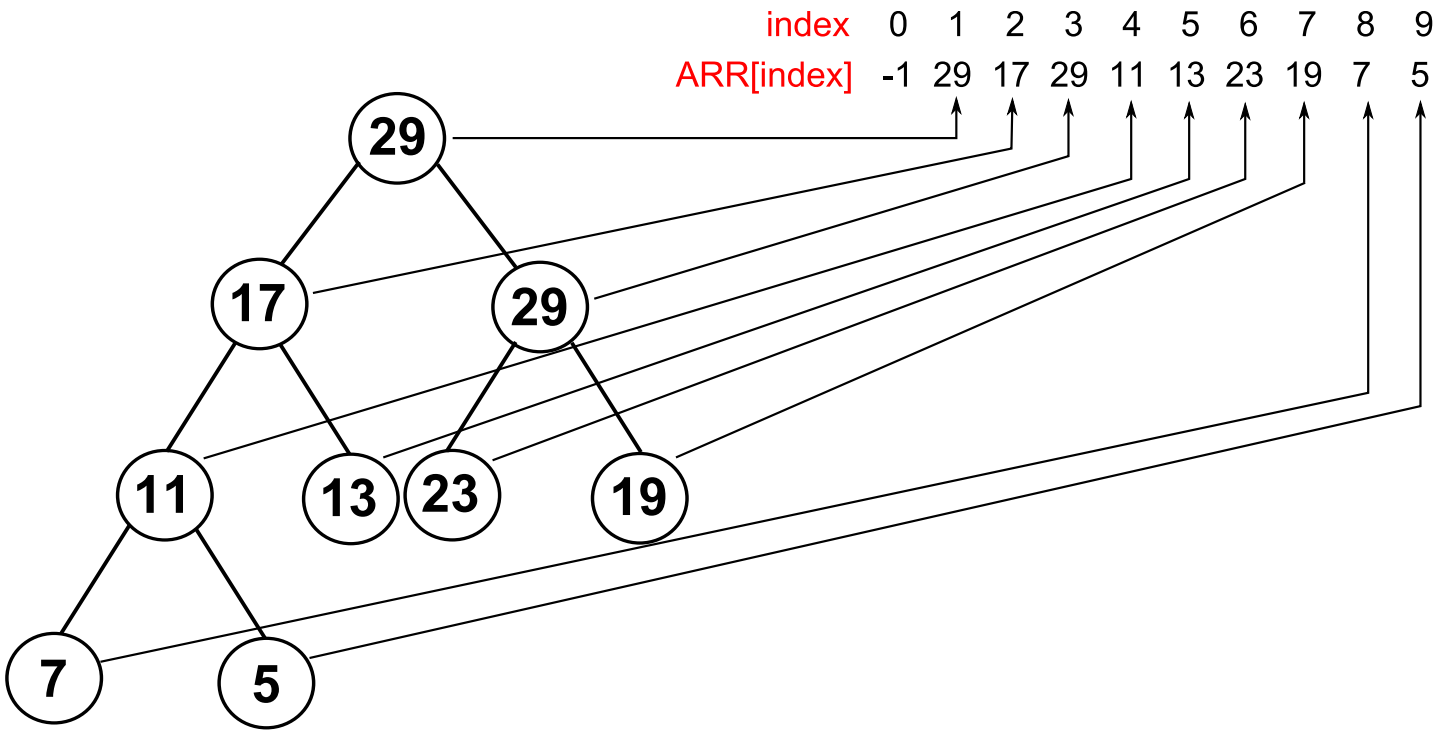
Operational efficiency on heaps

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

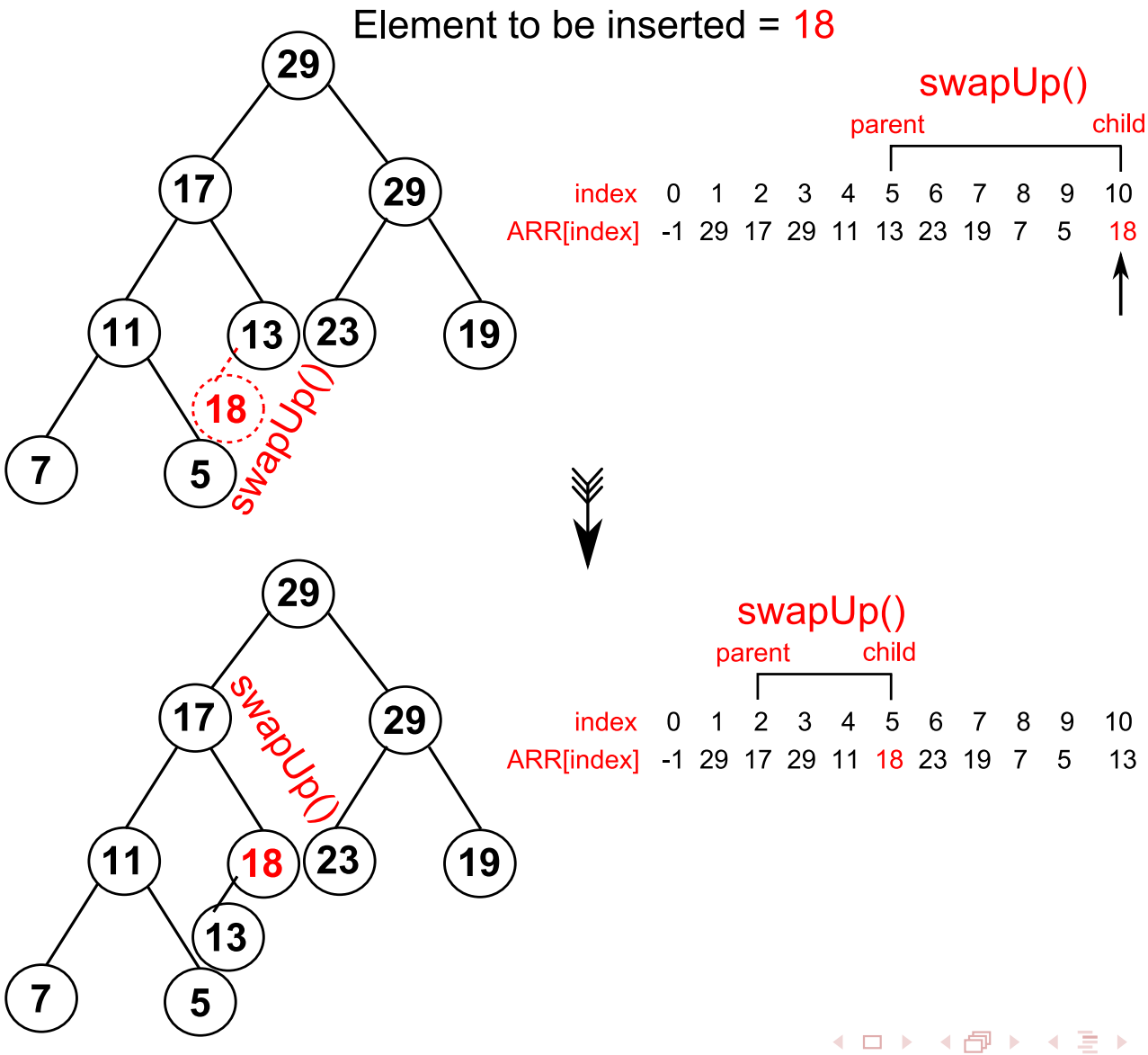
The data items traversed from a heap in level-order can be kept in an array.



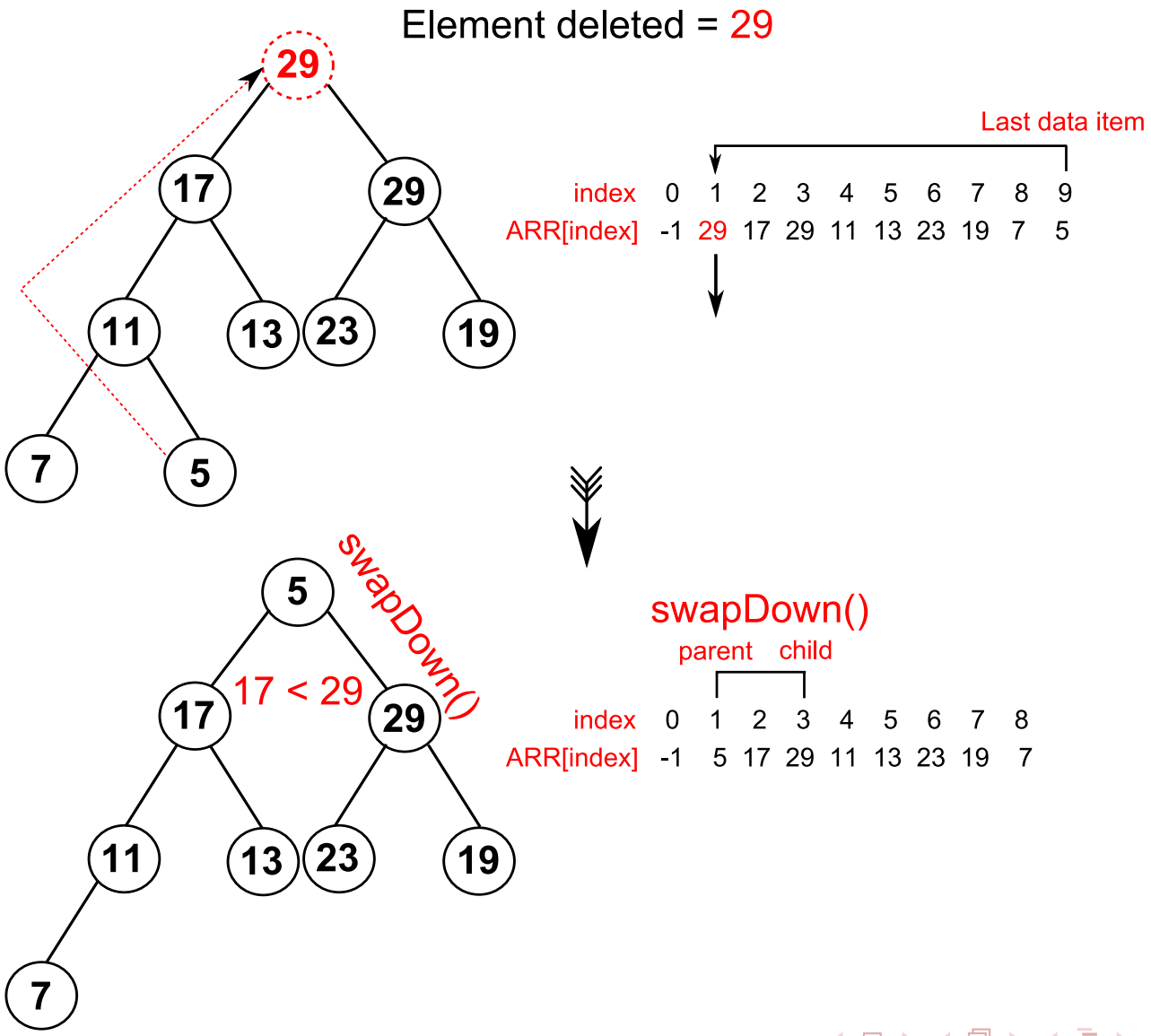
Heaps as arrays

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



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 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 15

1. 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

2. 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.
   ```