NON LINEAR DATA STRUCTURES : HEAP AND PRIORITY QUEUE

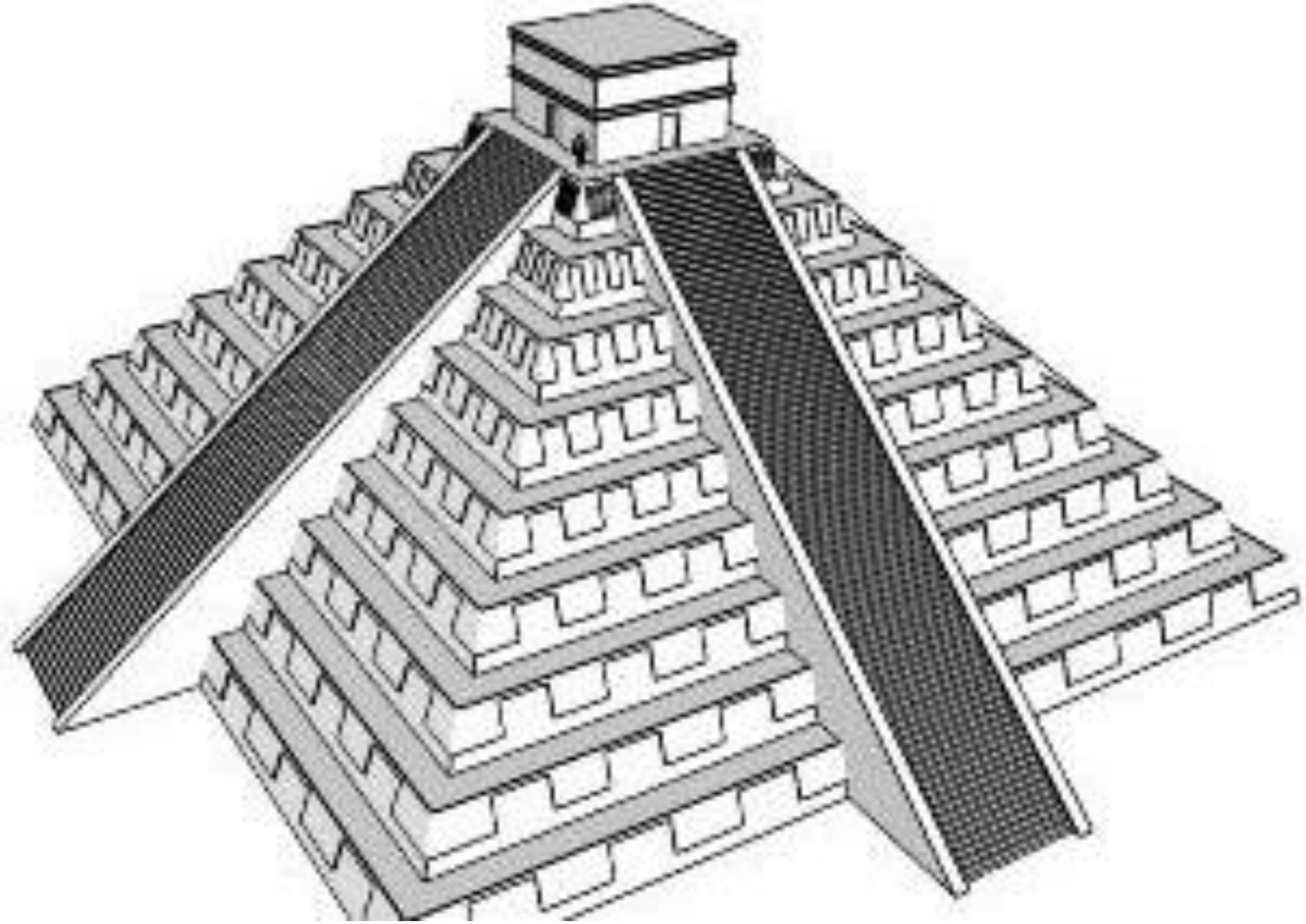
DATA STRUCTURES AND ALGORITHMS

HEAP DATA STRUCTURE

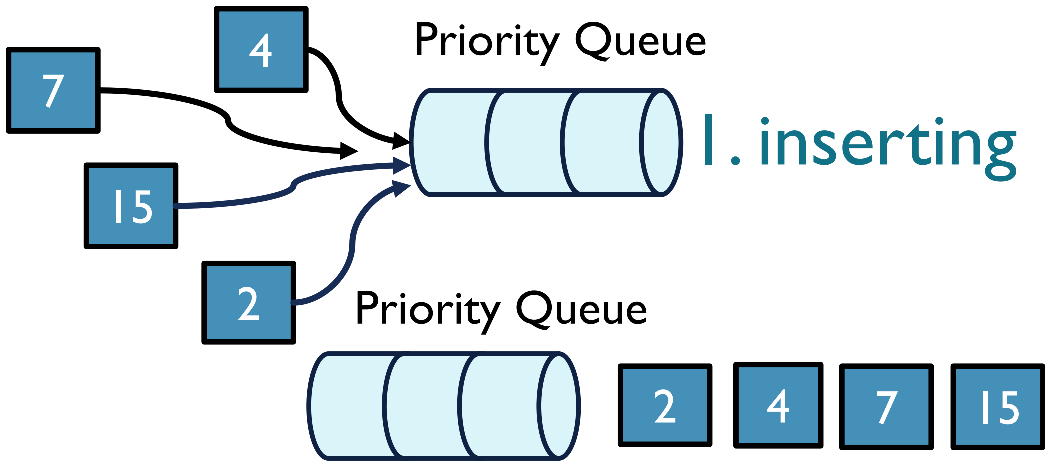
Heap

- Heap (also known as the pyramid)
  - Overview
  - Max and Min heap
  - Structure
  - Implementation

- Heap Application
  - Heap Sort
  - Job schedule problem



PRIORITY QUEUE



The Idea of Priority

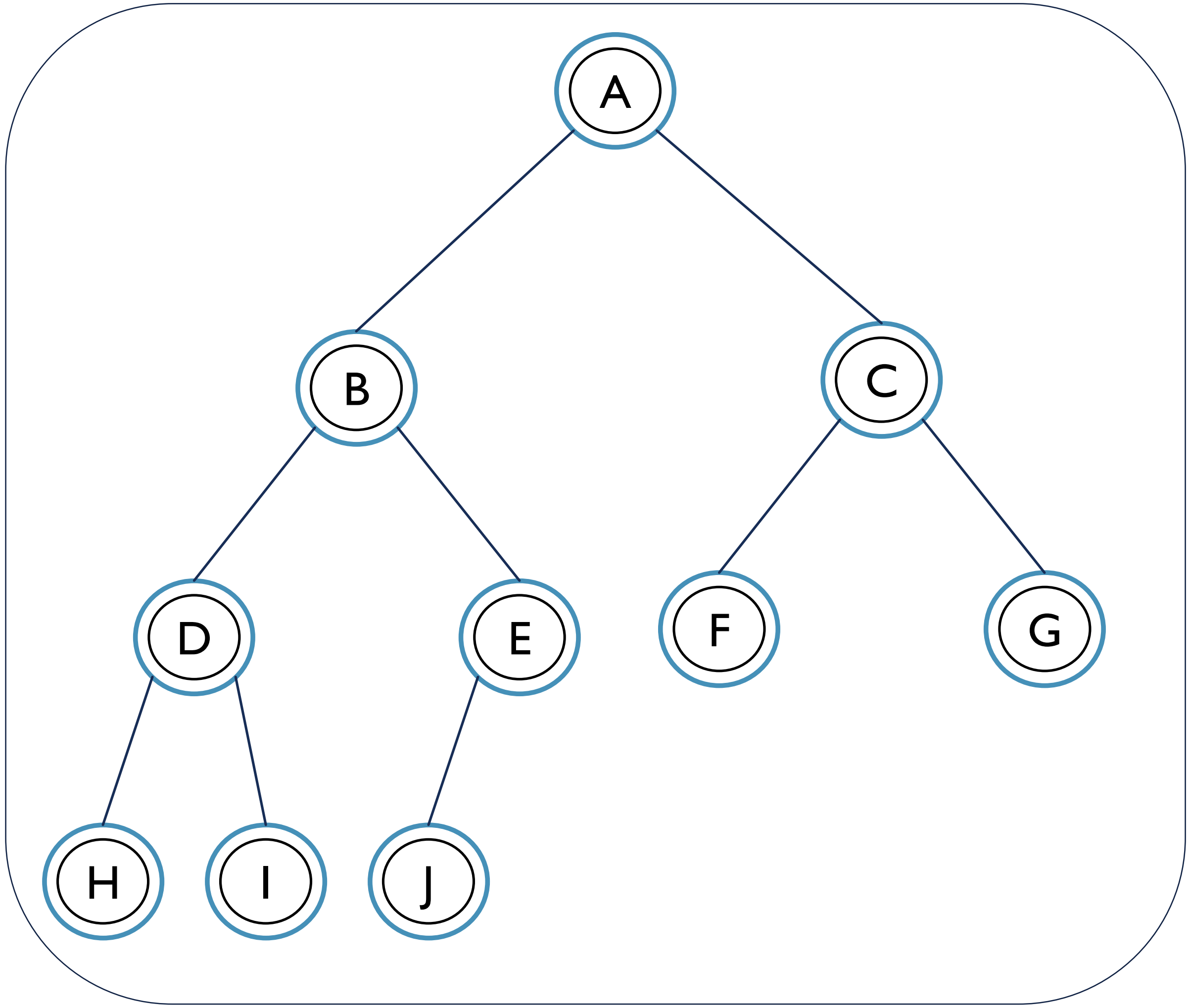
- Problem Description
  - Build and maintain a data structure for providing quick access to the smallest or largest key in the set.
- Discussion
  - Priority queues are useful data structures in simulations, particularly for maintaining a set of future events ordered by time. They are called "priority" queues because they enable you to retrieve items not by the insertion time (as in a stack or queue), nor by a key match (as in a dictionary), but by which item has the highest priority of retrieval.
- Implementation
  - One of Implementations achieved by the heap structure

STRUCTURE PROPERTY OF HEAPS

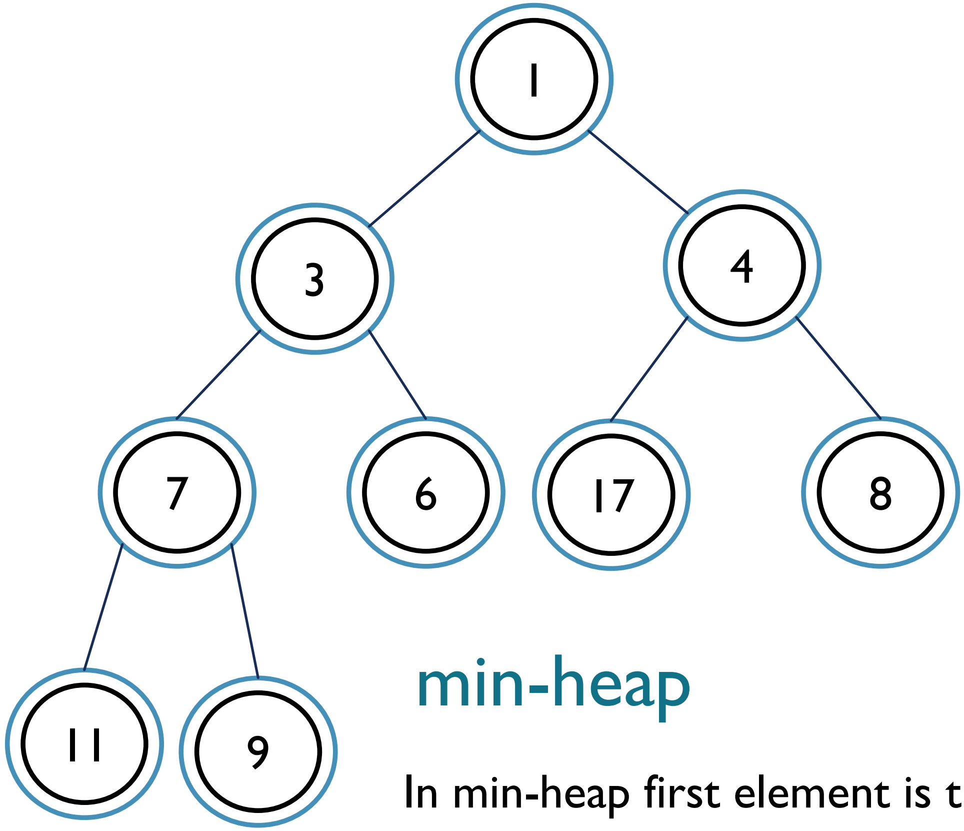
Binary Heap

- Binary heap is complete binary tree
  - Completely filled except the bottom level which is filled from left to right
- If heap has height h then number of nodes is between 2^h and $2^{h+1} - 1$
- Height of heap with N nodes is $O(logN)$

Complete binary tree it is a regular structure so can be represented in an array

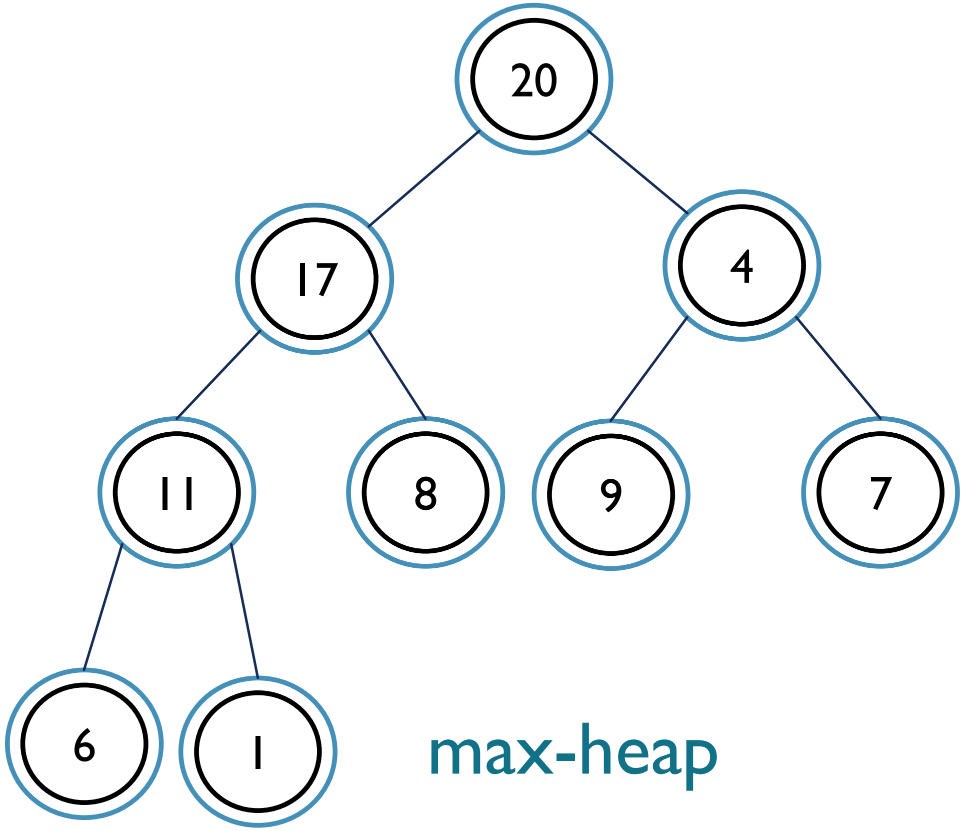


MAX VS MIN HEAP



min-heap

In min-heap first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



max-heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

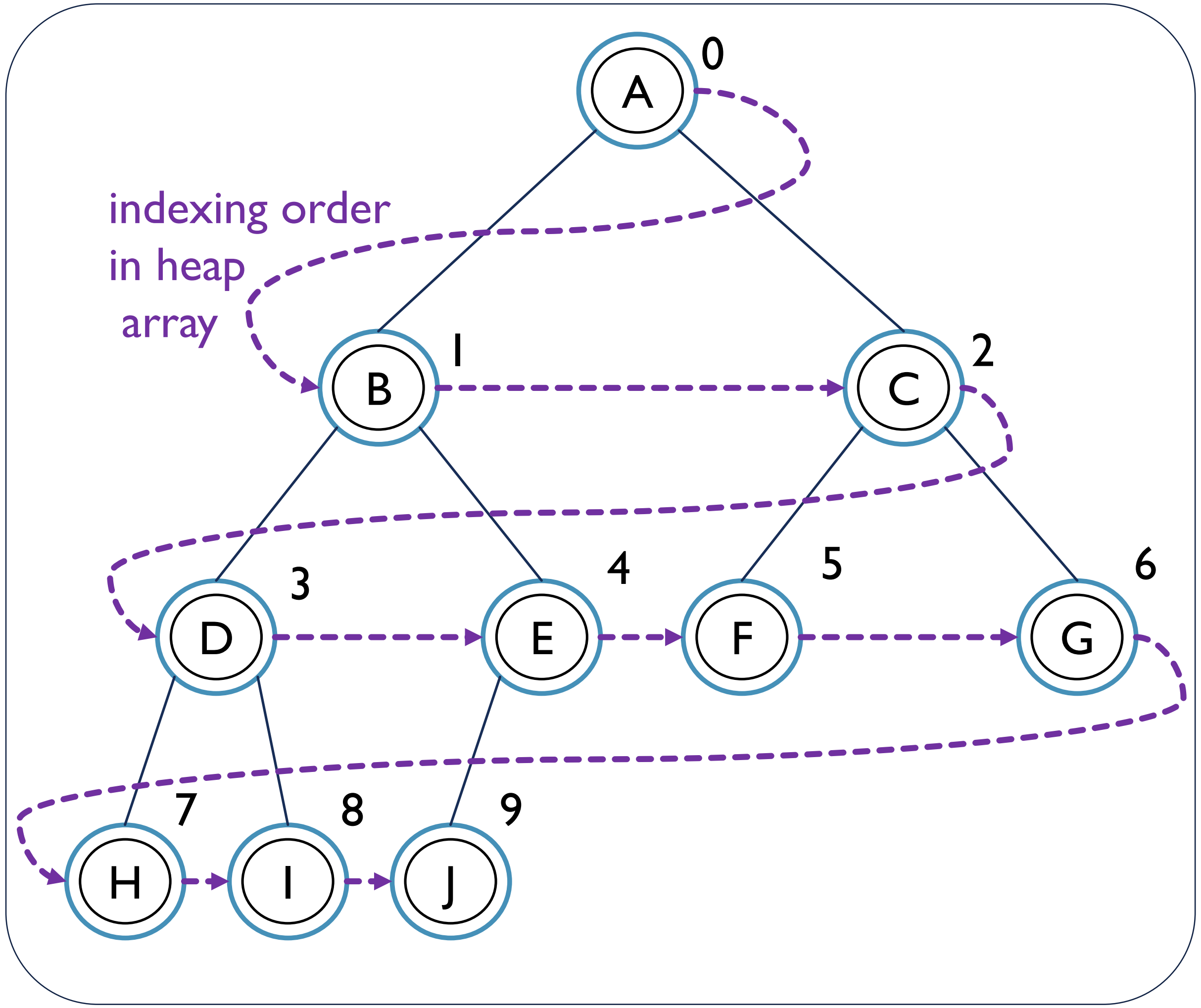
STRUCTURE PROPERTY OF HEAPS

Indexation in binary heap structure

- Unlike BST heap do not use pointers for successors, but stores elements in array
- $2i + 1$ is the left child
- $2i + 2$ is the right child
- $(i - 1)/2$ is the parent *(integer division !!!)*

For the heap implementation sometimes programmers use numeration not from zero, but from 1, then Left successor calculated by (2 * i), right = (i * n +1) and parent = (i / 2).

A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9

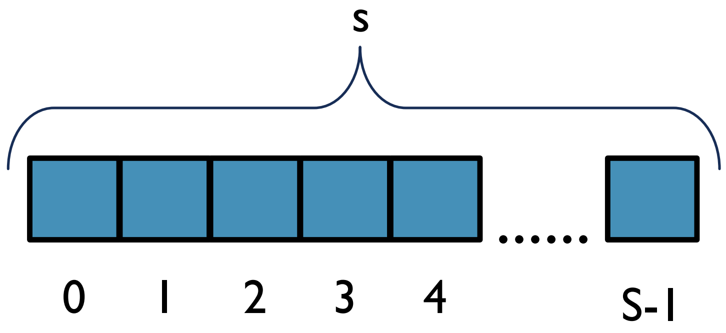


Array implementation requires an estimate on max-size

IMPLEMENTATION: HEAP STRUCTURE

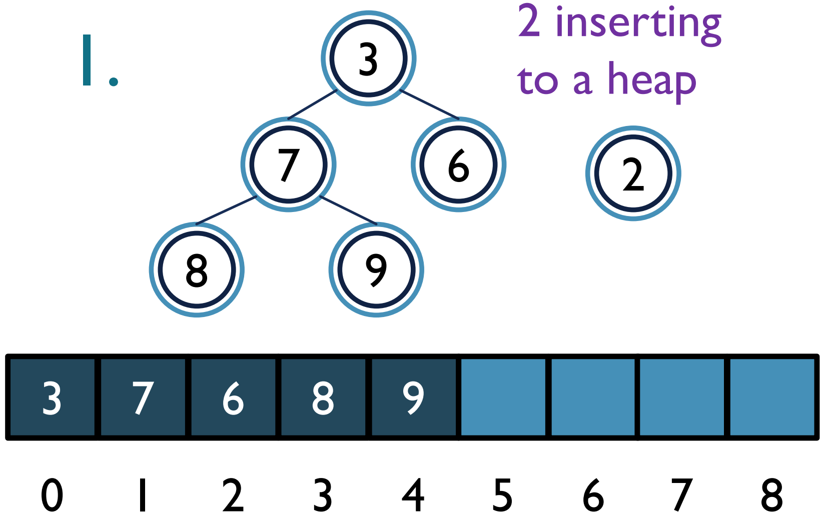
```
class MinHeap{
private:
  int *heap;
  int size;
  int max_size;
public:
MinHeap(int s);
  int left(int i){return  2*i + 1; }
  int right(int i){return 2*i+2;   }
  int parent(int i){ return (i-1)/2; }
  int get_min_heap(){ return heap[0]; }

  int extract_min();
  void decrease_key(int i, int new_value);
  void delete_key(int i);
  void insert_key(int k);
  void min_heapify(int i);
};
```

```
MinHeap::MinHeap(int s){
  size = 0;
  max_size = s;
  heap = new int[max_size];
}
```



HEAP: INSERTING ELEMENT

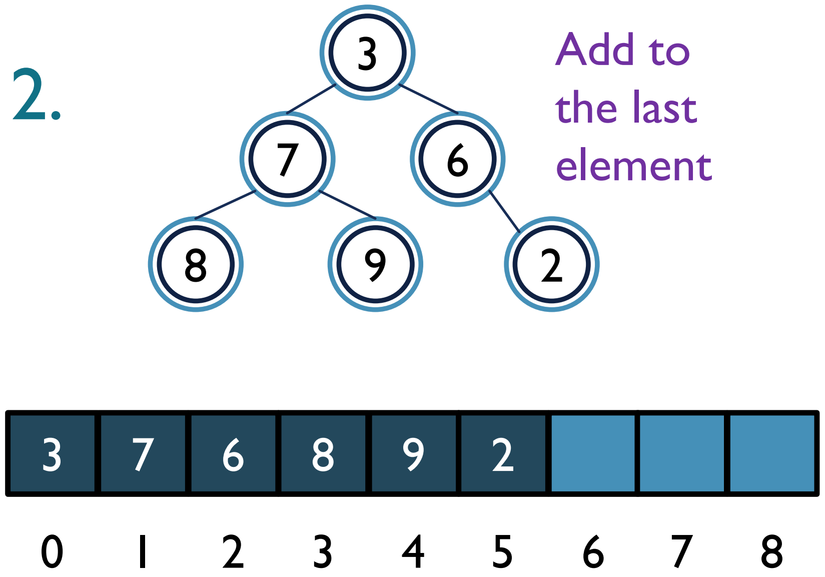
1.



2 inserting to a heap

0	1	2	3	4	5	6	7	8
3	7	6	8	9				

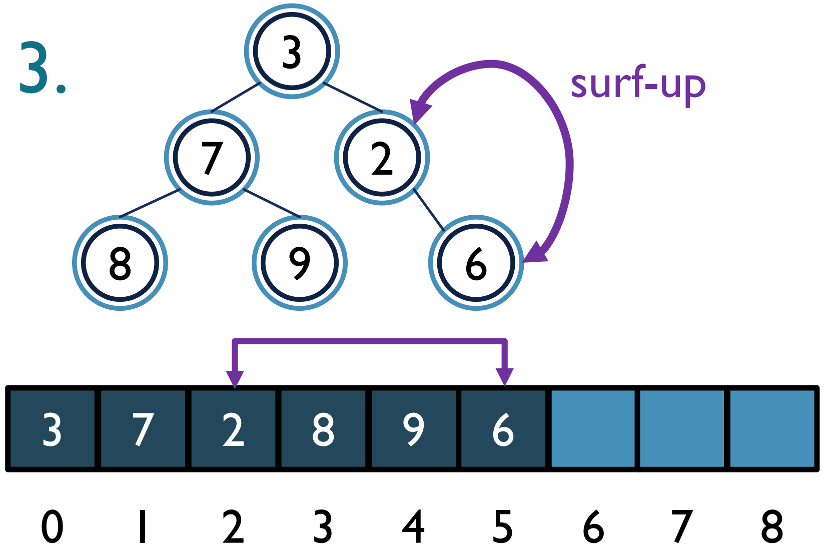
2.



Add to the last element

0	1	2	3	4	5	6	7	8
3	7	6	8	9	2			

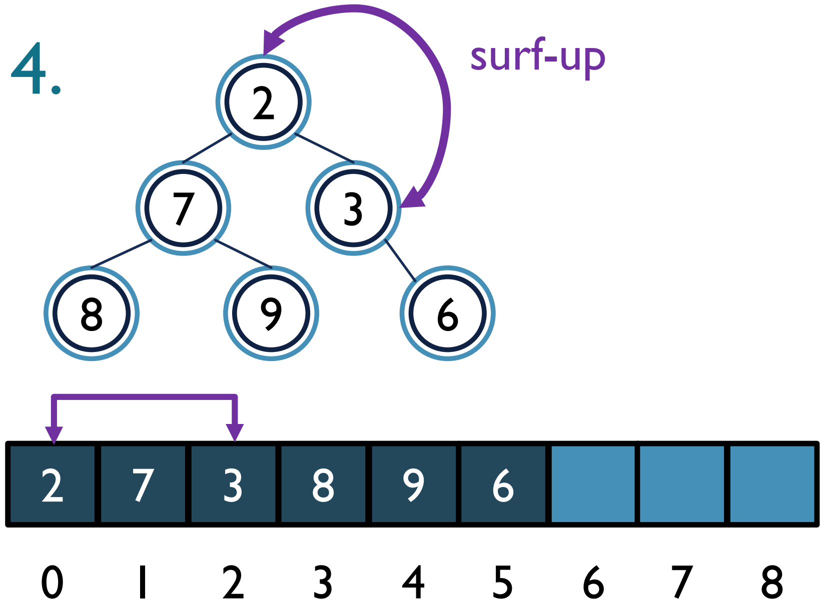
3.



surf-up

0	1	2	3	4	5	6	7	8
3	7	2	8	9	6			

4.



surf-up

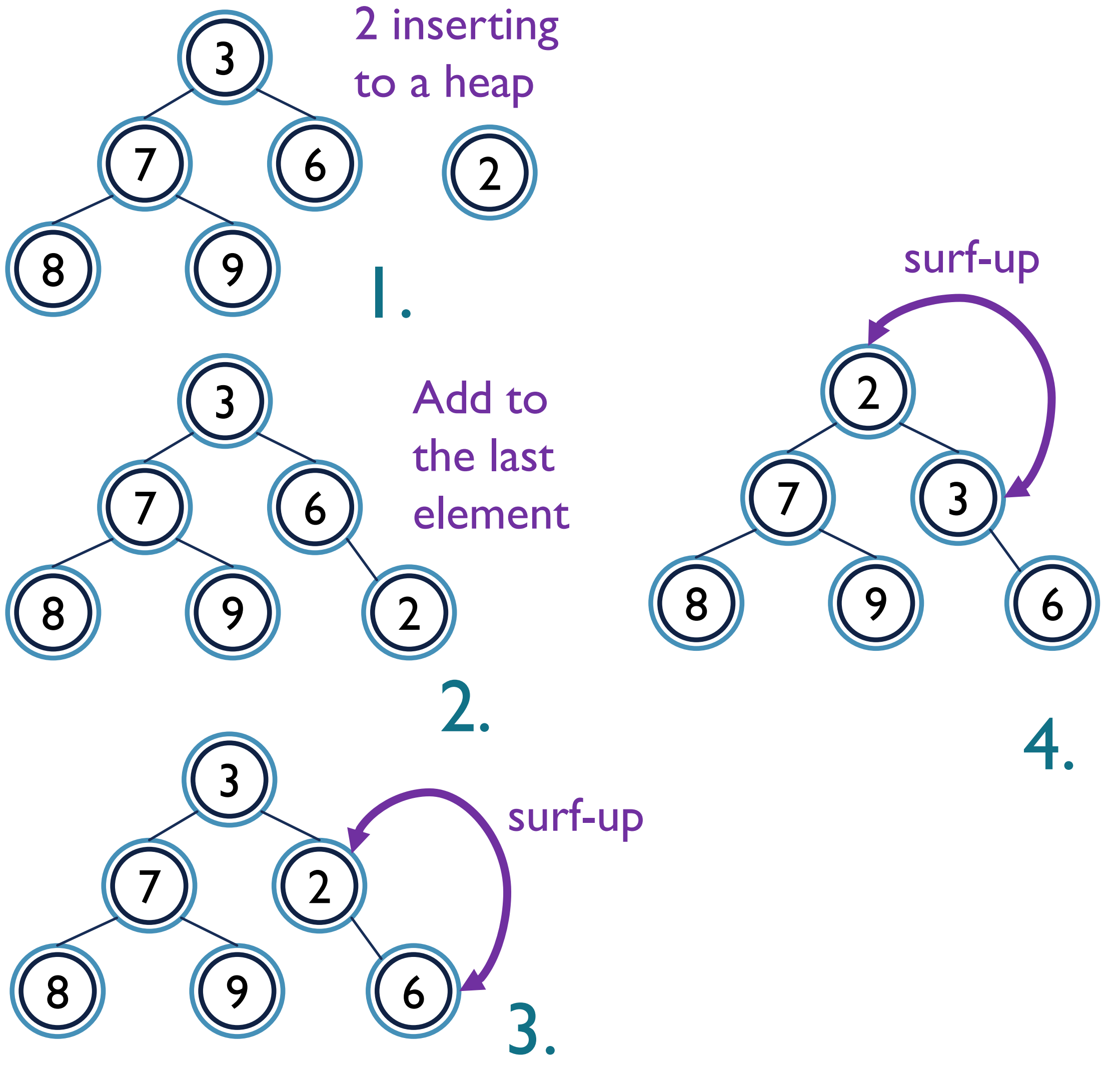
0	1	2	3	4	5	6	7	8
2	7	3	8	9	6			

Algorithm.

- Add the element to the end of array
- Then it checks the key of added element with it parent = (i-1)/2
- If (parent > added element) it swaps the it with parent. And continues to do it till the parent is less that added element or the position of the parent's element is equal to zero
  - This operation calls surf up

HEAP: INSERTION

```
void MinHeap::insert_key(int k){
  if(size == max_size){
      return;
  }
  size++;
  int i = size - 1;
  heap[i] = k;

  while (i != 0 && heap[parent(i)] > heap[i])
  {
    swap(heap[parent(i)], heap[i]);
    i = parent(i);
  }
}
```



2 inserting to a heap

1.

Add to the last element

2.

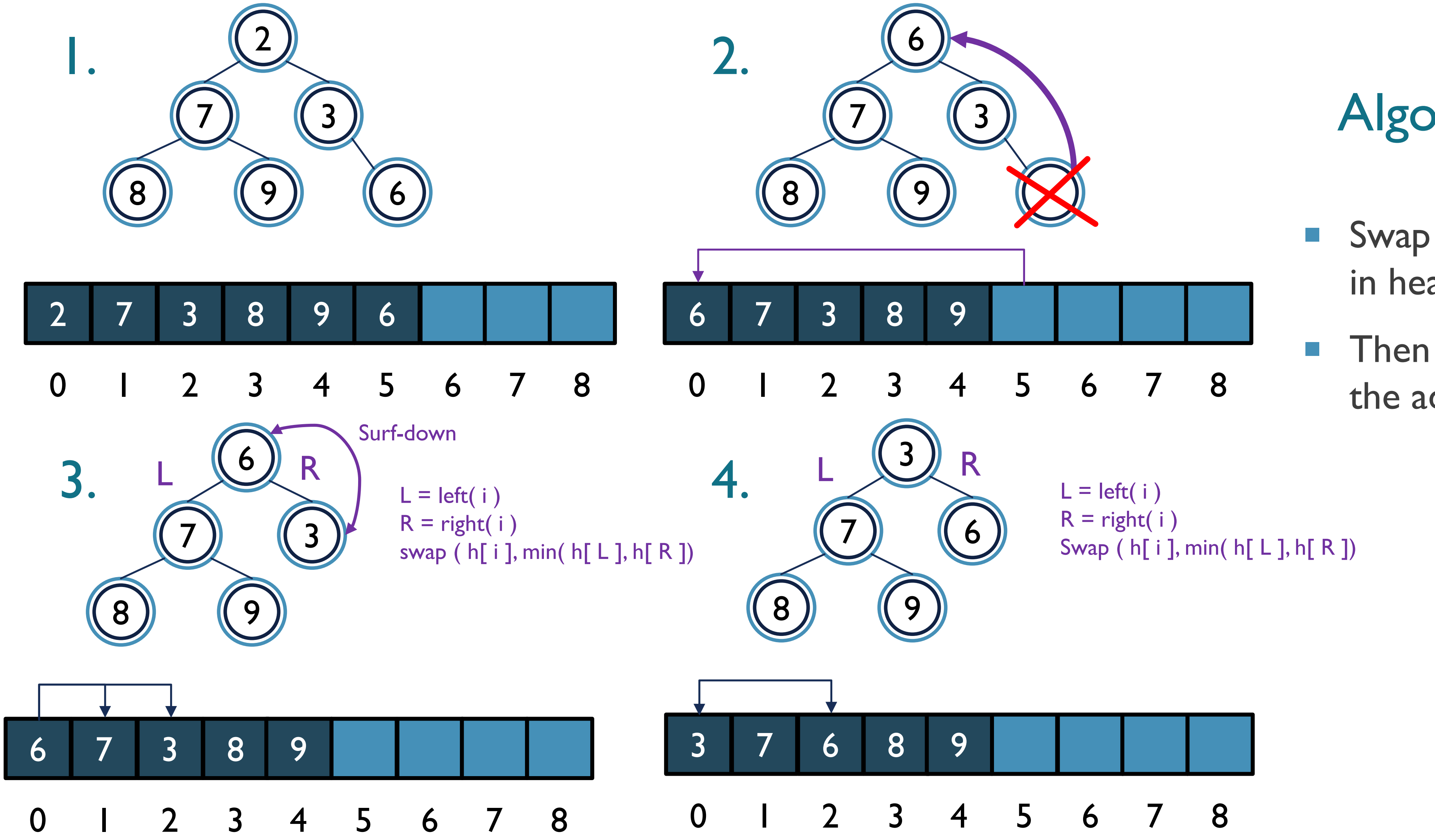
surf-up

3.

surf-up

4.

REMOVE THE MİNİMUM NODE



1.

0	1	2	3	4	5	6	7	8
2	7	3	8	9	6			

2.

0	1	2	3	4	5	6	7	8
6	7	3	8	9				

3.

```
L = left( i )
R = right( i )
swap ( h[ i ], min( h[ L ], h[ R ])
```

0	1	2	3	4	5	6	7	8
6	7	3	8	9				

4.

```
L = left( i )
R = right( i )
Swap ( h[ i ], min( h[ L ], h[ R ])
```

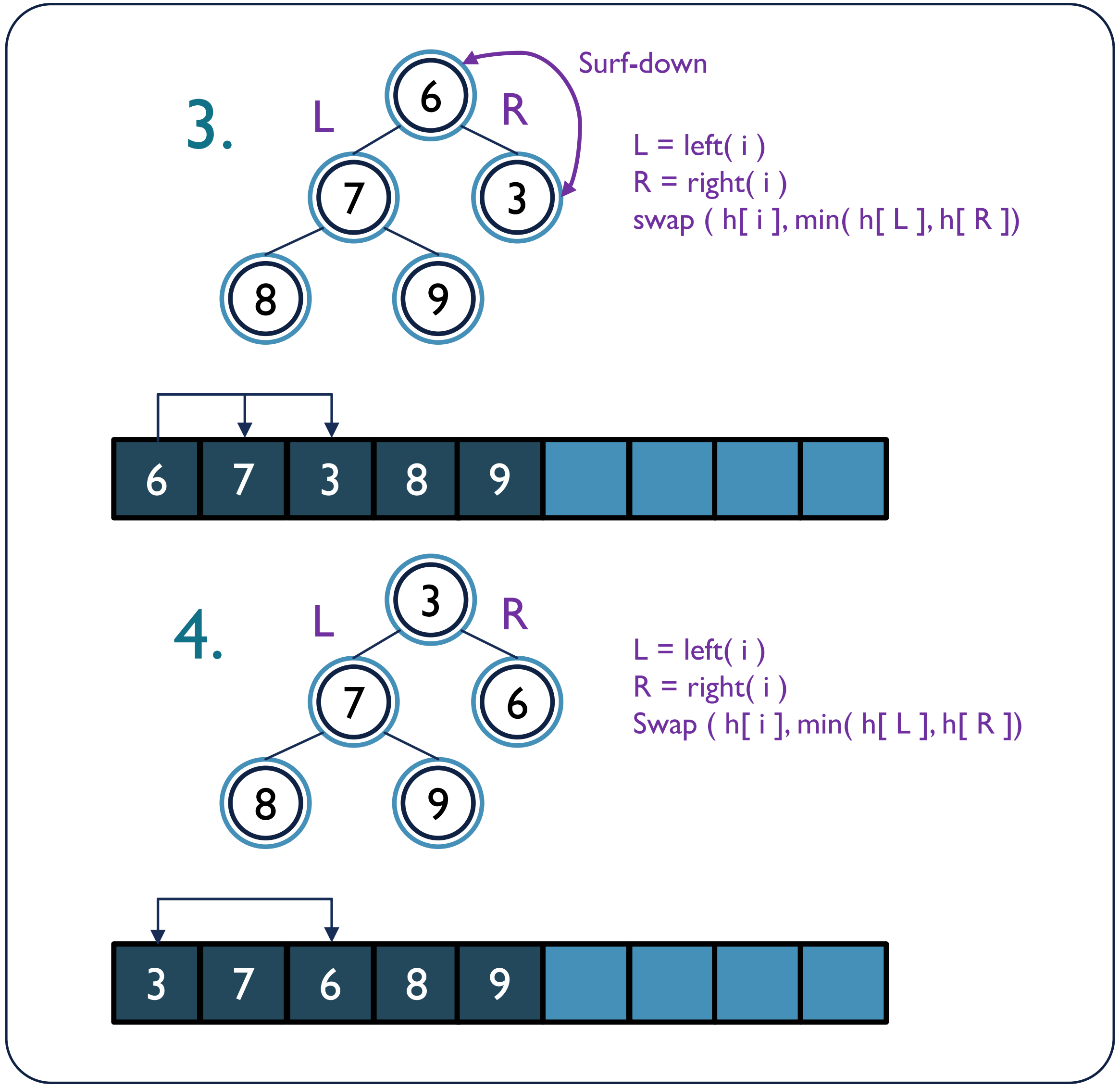
0	1	2	3	4	5	6	7	8
3	7	6	8	9				

Algorithm.

- Swap first and last elements in heap array (1→ 2)
- Then surf-down (heapify) the added element

HEAP: MIN HEAPIFY

```
void MinHeap::min_heapify(int i){

  int l = left(i);
  int r = right(i);
  int smallest = i;
  if( l < size && heap[l] < heap[i])
     smallest = l;
  if(r < size && heap[r] < heap[smallest])
     smallest = r;
  if(smallest != i ){
      swap(heap[i], heap[smallest]);
      min_heapify(smallest);
  }
}
```



L = left(i)
R = right(i)
swap (h[i], min(h[L], h[R])

L = left(i)
R = right(i)
Swap (h[i], min(h[L], h[R])

HEAP: EXTRACT MIN

```
int MinHeap::extract_min(){

   if(size <= 0)
      return 1e9;

   if(size == 1){
      size--;
      return heap[0];
   };

   int root = heap[0];
   heap[0] = heap[size-1];
   size--;
   min_heapify(0);

   return root;
}
```

Algorithm. (detailed in previous slides)

- Check the heap array if size is zero or less return infinity (very big number)
- For case when there is only one element return it
- Otherwise
  - swap first and last element
  - and surf-down (min_heapify) the root

HEAP: DECREASE KEY AND DELETE

```
void MinHeap::decrease_key(int i, int new_value){
  heap[i] = new_value;
  while (i != 0 && heap[parent(i)] > heap[i])
  {
    swap(heap[parent(i)], heap[i]);
    i = parent(i);
  }
}
```

```
void MinHeap::delete_key(int i){
  decrease_key(i, 1e9);
  extract_min();
}
```

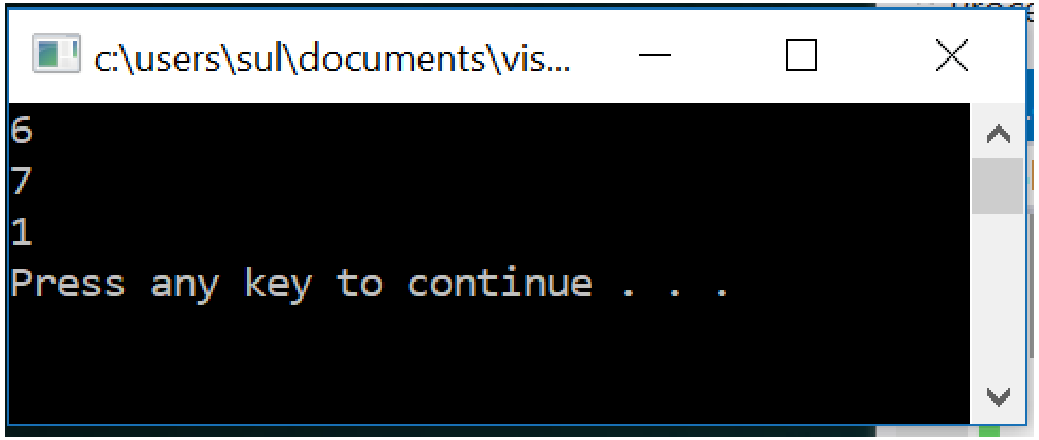
Purposes.

Updates a value at index I, then surfs up (works iff the new value is less than updated) otherwise it should surf-down

This function deletes key at index i. It first reduced value to infinite, then calls extract_min()

USING THE HEAP

```
int main(){

    MinHeap *h = new MinHeap(11);
    h->insert_key(34);
    h->insert_key(6);
    h->insert_key(543);
    h->insert_key(7);
    h->insert_key(543345);
    h->insert_key(22);

    cout<<h->extract_min()<<endl;
    cout<<h->get_min_heap()<<endl;
    h->decrease_key(2,1);
    cout<<h->get_min_heap()<<endl;
    system("pause");
    return 0;
}
```



```
6
7
1
Press any key to continue . . .
```

HEAP SORT

USES HEAPIFY

HEAP SORT

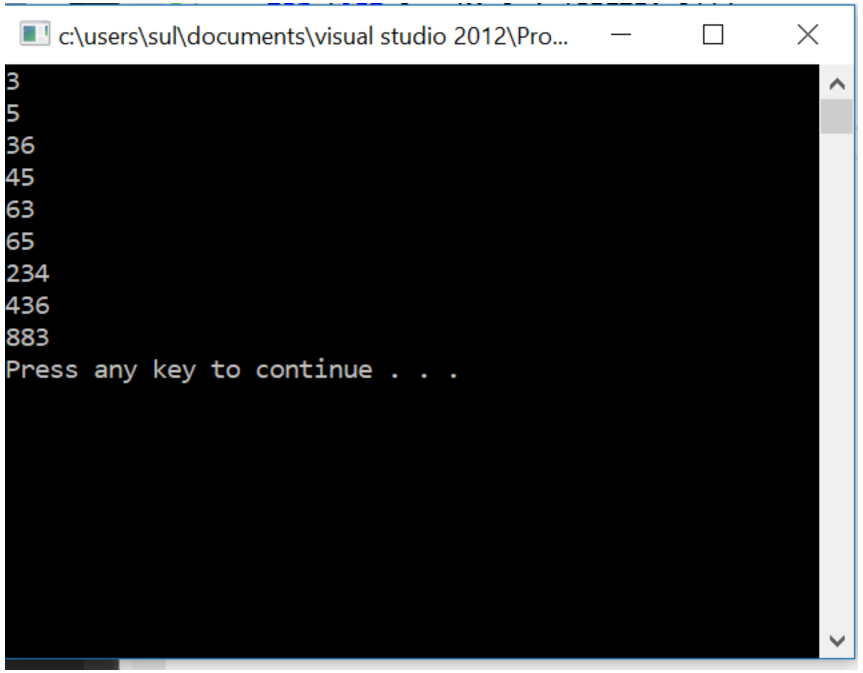
```
void heapify(int arr[], int n, int i){

  int largest = i;
  int l = 2*i + 1;
  int r = 2*i + 2;

  if(l < n && arr[l] > arr[largest])
    largest = l;

  if(r < n && arr[r] > arr[largest])
    largest = r;

  if(largest != i){
    swap(arr[i], arr[largest]);
    heapify(arr, n, largest);
  }
}
```

```
void heap_sort(int arr[], int n){

  for (int i = n/2-1; i >= 0; i--)
    heapify(arr,n,i);

  for (int i = n-1; i >= 0; i--){
    swap(arr[0], arr[i]);
    heapify(arr,i,0);
  }
}
```

Sorted by weight.

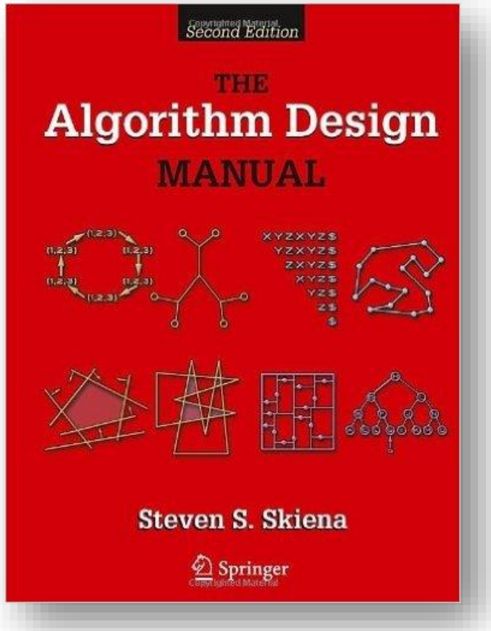
HEAP SORT

```
int main(){

  int arr[] = {883,3,5,63,65,436,45,234,1,36};
  int length = sizeof(arr)/sizeof(int);

  heap_sort(arr, length);

  for (int i = 0; i < length; i++)
  {
    cout<<arr[i]<<endl;
  }
  system("pause");
}
```

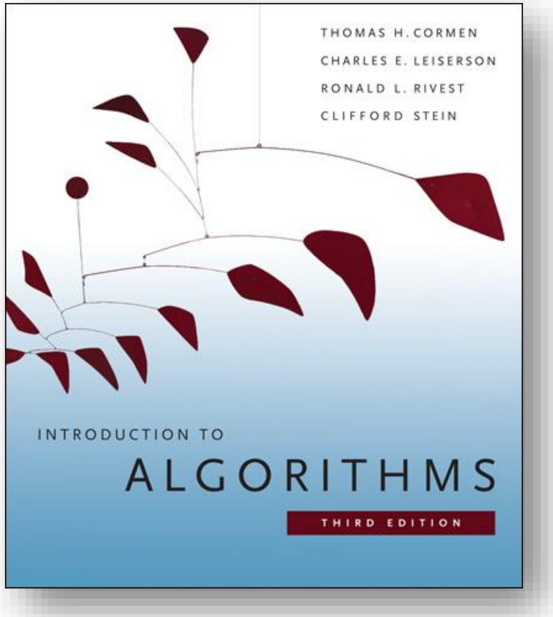


```
3
5
36
45
63
65
234
436
883
Press any key to continue . . .
```

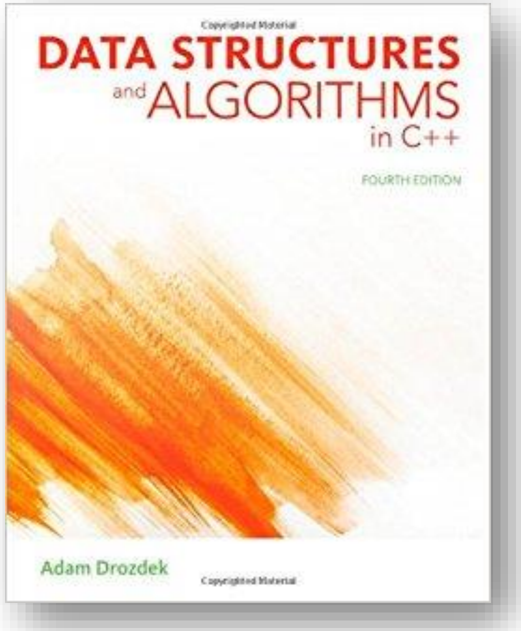
LITERATURE



Stieven Skienna
Algorithms design manual
3.5 Priority Queue
Page 83



Thomas H. Cormen
Introduction to Algorithms
Chapter 6.1 Heaps
Page 151.



Adam Drozdek
Data structures and Algorithms in C++
Chapter 6.9 Heaps
Page 268