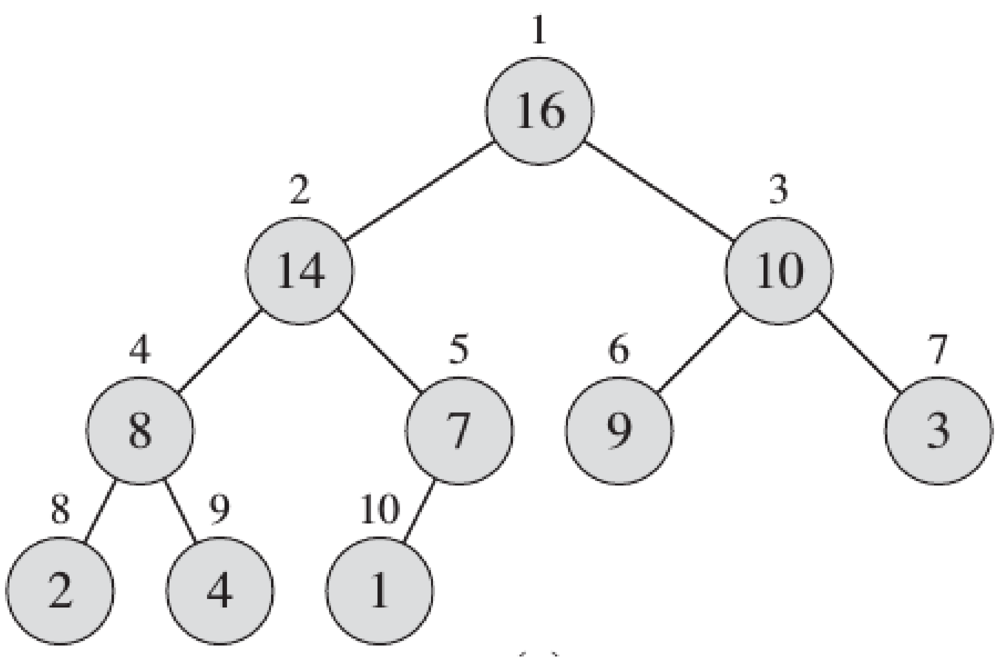
Section 7

Abstract Data Types: Heap & Priority Queue (ADT)

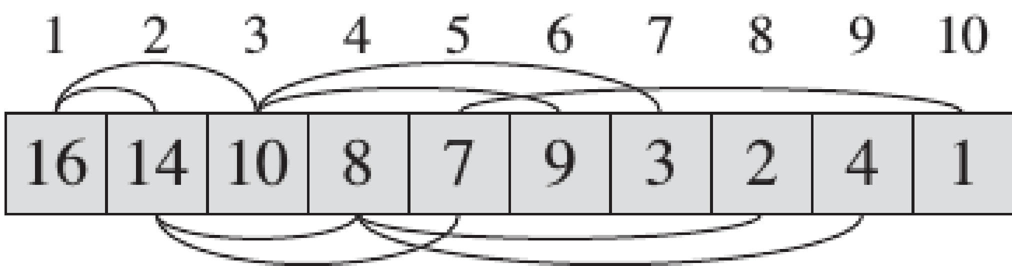
Presentation by *Asem Alaa*

Heaps

Max-Heap Logical Representation

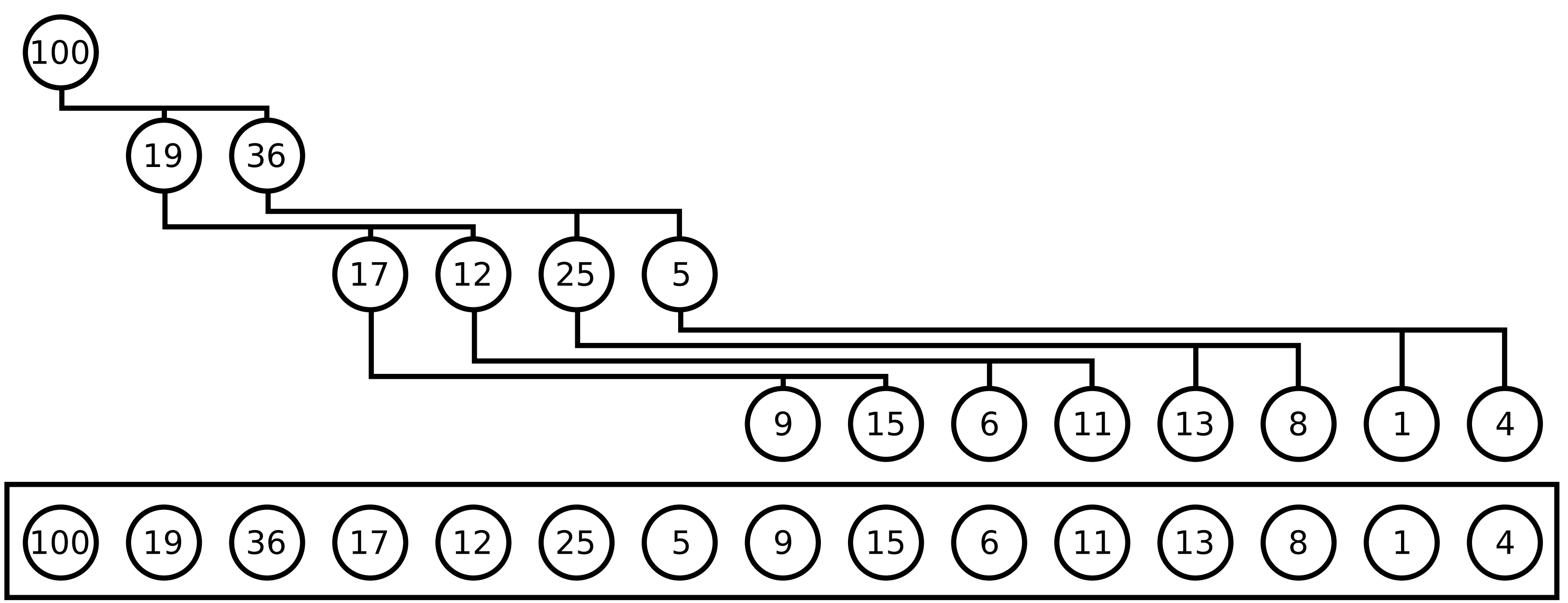


Max-Heap Storage



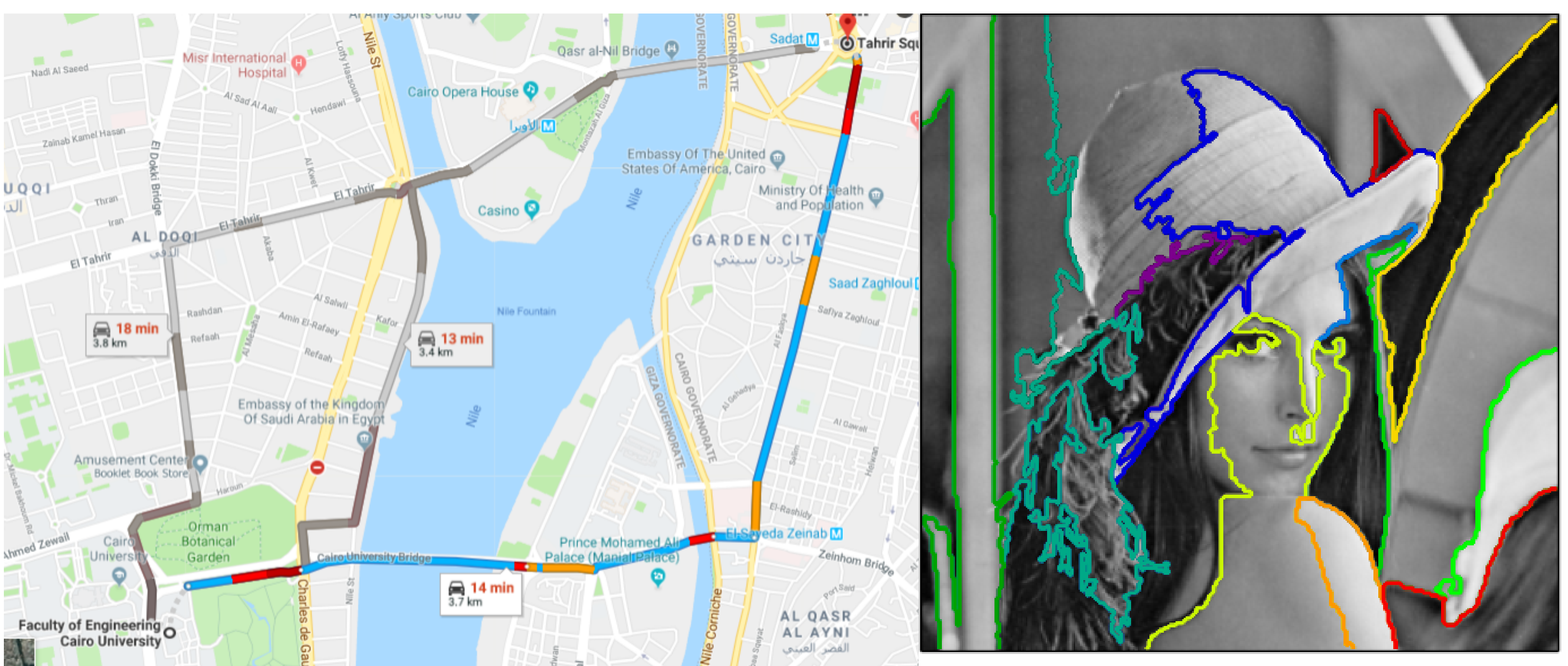
Heaps

Max-Heap



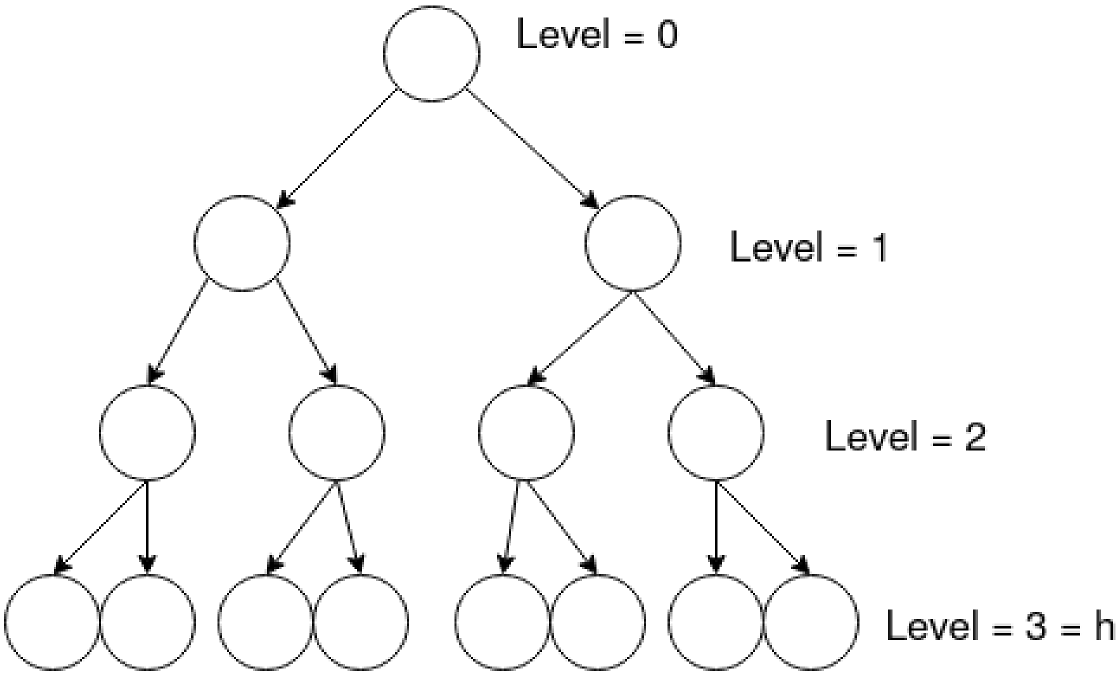
Heap Applications

- Sorting Algorithms (Heapsort)
- The Shortest Path Problem (Dijkstra's Algorithm)
- Data Compression Algorithms (Huffman Tree)
- Unsupervised Machine Learning (Agglomerative Clustering)



{A demo of structured Ward hierarchical clustering on Lena image}{Agglomerative Clustering Algorithm Using Heap}

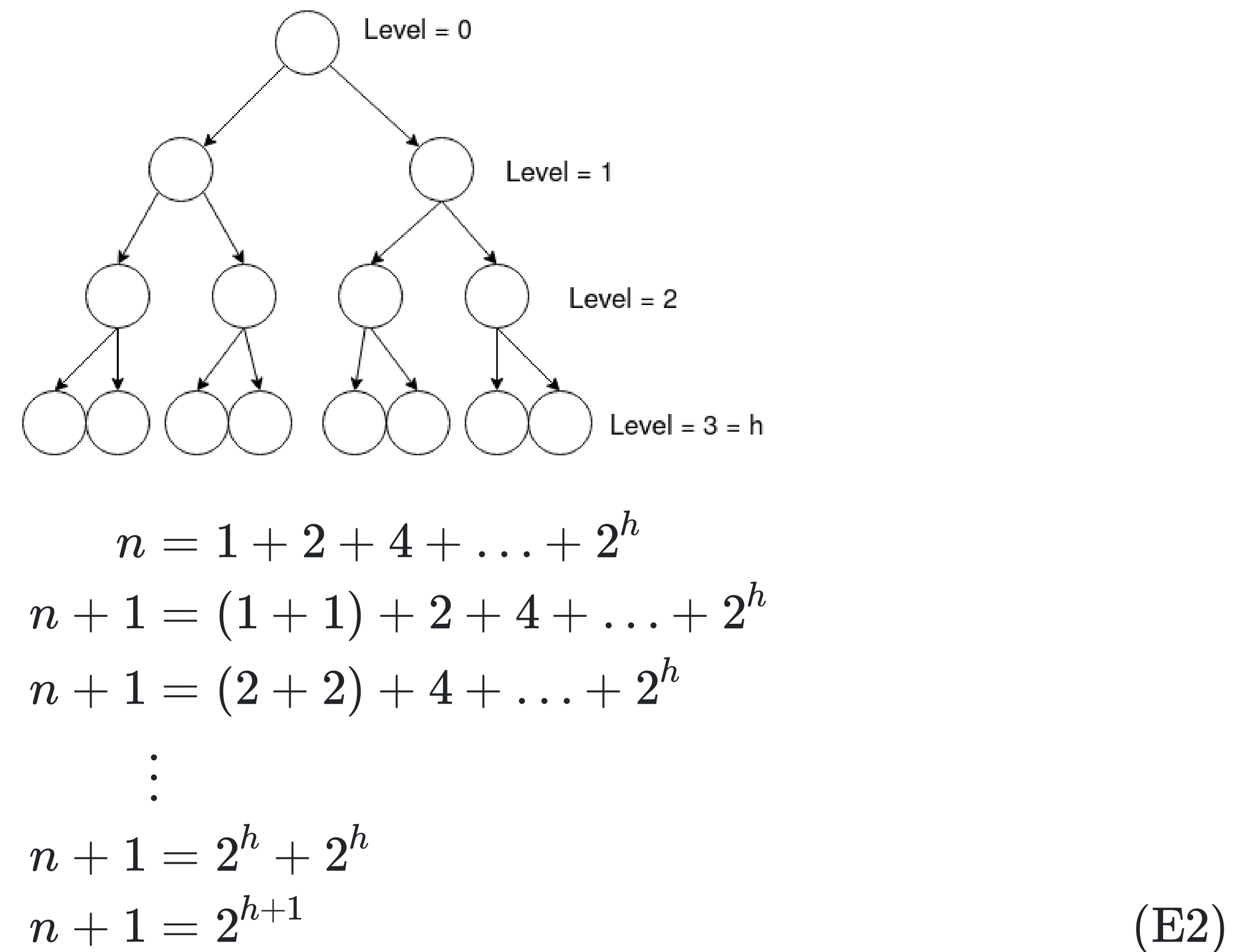
Glossary



- Complete Tree: A balanced tree in which the distance from the root to any leaf is either $\lfloor \log(n) \rfloor$ or $\lfloor \log(n) - 1 \rfloor$. source.

Complete Tree: Relating n to h

- h: the height of a full binary tree and
- n: the number of nodes



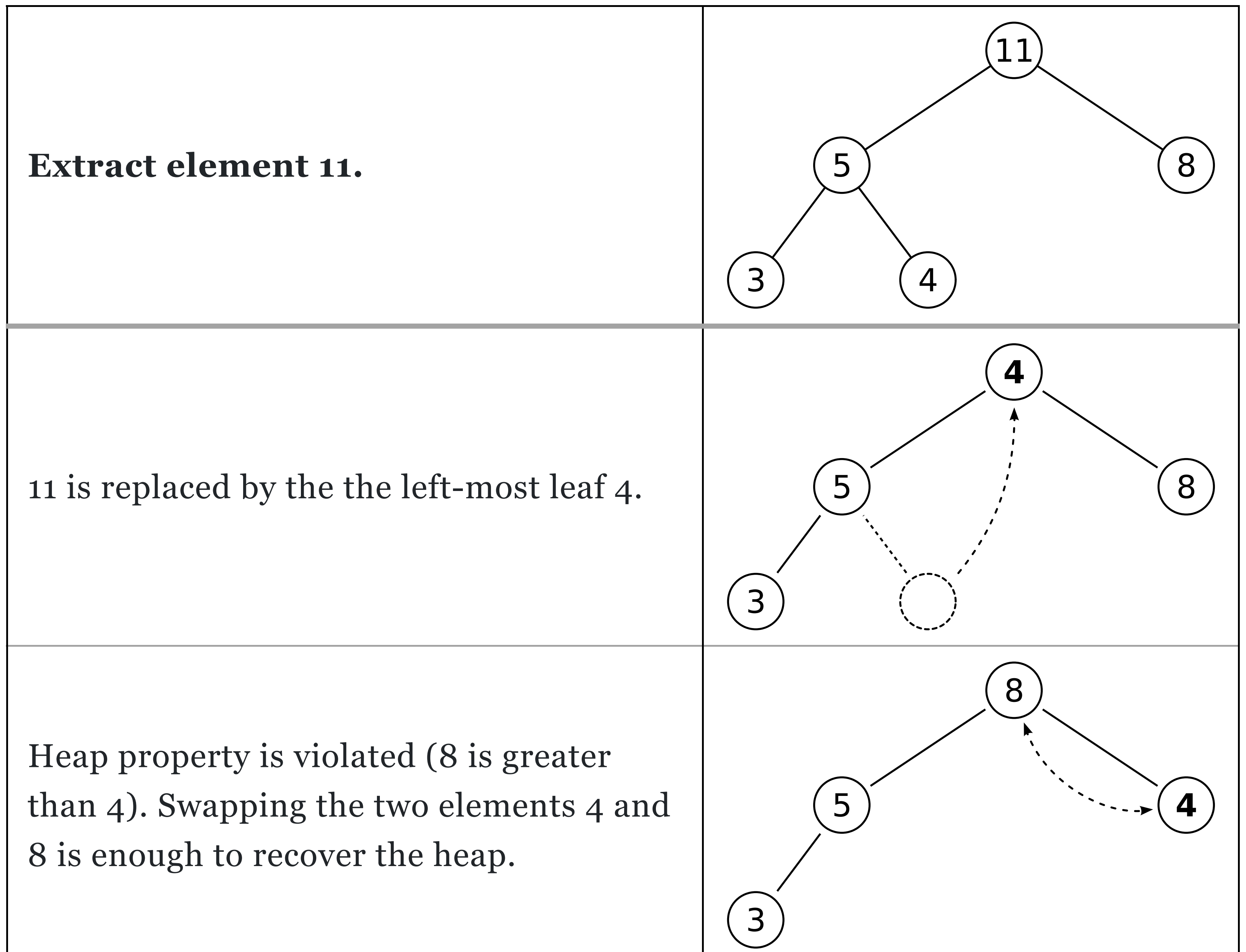
$$
\begin{aligned}
n &= 1 + 2 + 4 + \ldots + 2^h \\
n + 1 &= (1 + 1) + 2 + 4 + \ldots + 2^h \\
n + 1 &= (2 + 2) + 4 + \ldots + 2^h \\
&\vdots \\
n + 1 &= 2^h + 2^h \\
n + 1 &= 2^{h+1} \qquad \text{(E2)}
\end{aligned}
$$

Heap Operations: Insert

We first place the new element 15 in the position marked by the X as a leaf.	
However, the heap property is violated since 15 > 8, so we need to swap the 15 and the 8	
The heap property is still violated since 15 > 11, so we need to swap again	

{Binary_heap - Wikipedia}

Heap Operations: Extract



Extract element 11.	
11 is replaced by the the left-most leaf 4.	
Heap property is violated (8 is greater than 4). Swapping the two elements 4 and 8 is enough to recover the heap.	

{Binary_heap - Wikipedia}

Min-heap Implementation

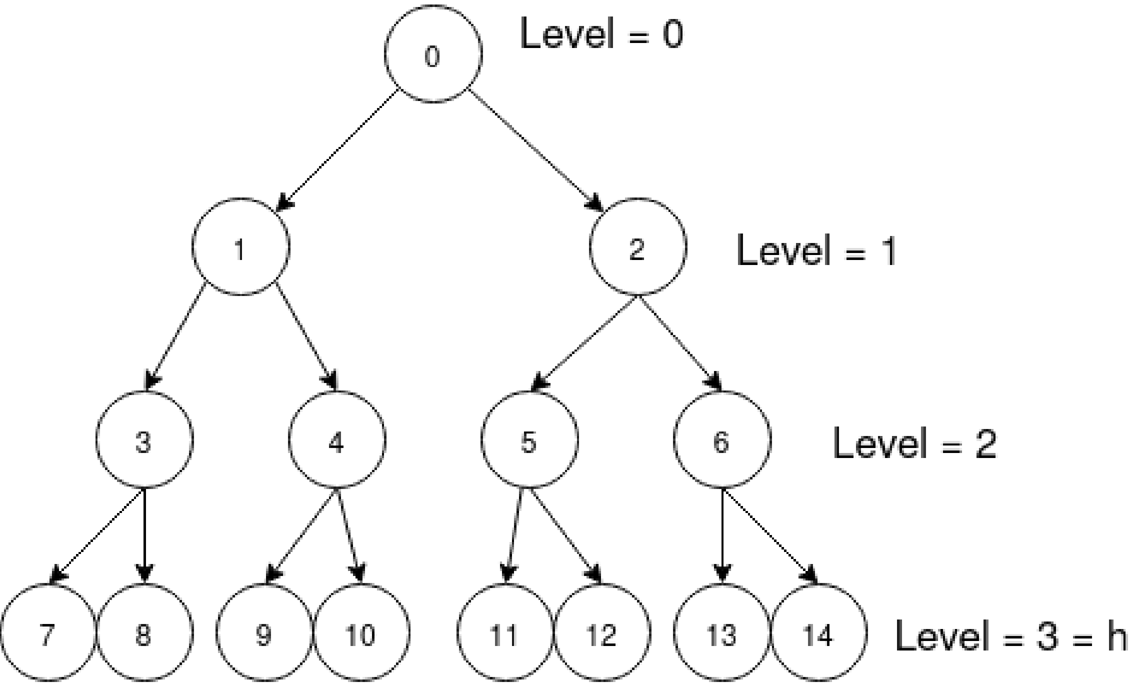
+Priority Queue Interface

```
template< typename T >
class Heap
{
public:
    // Return heap size
    size_t size() const {}
    // 1. Insert as leaf
    // 2. Recover heap properties
    void insert(T value){}
    // 1. Extract the root
    // 2. Recover heap properties
    T extract(){}
private:
    // Private details
};
```

Implementation: Heap Storage

```
template< typename T >
class Heap
{
public:
    size_t size() const {}
    void insert(T value){}
    T extract(){}
private:
    std::vector< T > data;
};
```

Implementation: From Parent to Child (+vice versa)



- From parent to left child: `parentIdx * 2 + 1`
- From parent to right child: `parentIdx * 2 + 2`
- From left child (odd index) to parent: `(childIdx - 1) / 2`
- From right child (even index) to parent: `(child - 2) / 2`

Implementation: From Parent to Child (+vice versa)

```
template< typename T >
class Heap {
public:
    size_t size() const {}
    void insert(T value){}
    T extract(){}
private:
    static size_t leftChildIdx(size_t parent){
        return parent * 2 + 1;
    }
    static size_t rightChildIdx(size_t parent){
        return parent * 2 + 2;
    }
    static size_t parentIdx(size_t child){
        if (child % 2 == 1) return (child - 1) / 2;
        else return (child - 2) / 2;
    }
    std::vector< T > data;
};
```

Implementation: Heap size

```
template< typename T >
class Heap
{
public:
    size_t size() const { return data.size();}
    void insert(T value){}
    T extract(){}
private:
    // Private methods
    static size_t leftChildIdx(size_t parent){... }
    static size_t rightChildIdx(size_t parent){... }
    static size_t parentIdx(size_t child){... }
private:
    // Private data members
    std::vector< T > data;
};
```

Implementation: Insert & SiftUp

```
template< typename T >
class Heap{
public:
    void insert(T value){
        data.push_back(value);
        size_t childIdx = size() - 1;
        siftUp( childIdx ); // Recover heap
    }
private:
    void siftUp( size_t child ){
        auto parent = parentIdx(child);
        if( child > 0  && data[child] < data[parent]){
            std::swap(data[child], data[parent]);
            siftUp( parent );
        }
    }
};
```

- Worst case time: $O(T(n)) = O(h) = O(\log(n))$

Implementation: Extract & SiftDown

```
template< typename T > class Heap{
public:
    T extract(){
        if( data.empty()) exit( 1 ); // Crash
        size_t child = size() - 1;
        std::swap(data[child], data[0]);
        T value = data.back(); data.pop_back();
        siftDown(0);
        return value;
    }
private:
    void siftDown( size_t parent){
        size_t left = leftChildIdx(parent), right = rightChildIdx(parent);
        size_t min = parent;
        if (left < size()  && data[left]  < data[min]) min = left;
        if (right < size() && data[right] < data[min]) min = right;
        if (min != parent){
            std::swap(data[min], data[parent]);
            siftDown( min );
        }
    }
};
```

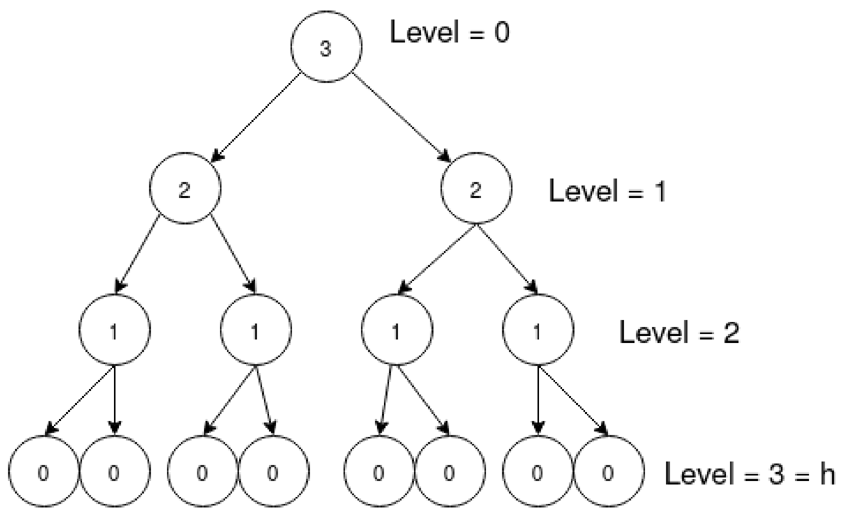
- Worst case time: $O(T(n)) = O(h) = O(\log(n))$

Implementation: Heapifying Arbitrary Array

```
template< typename T >
class Heap
{
public:
    static Heap make( std::vector< T > data )
    {
        Heap h;
        h.data.swap( data ); // O(1)
        if( h.size() <= 1 ) return h;

        auto lastChild = h.size() - 1;
        for( int subHp = parentIdx( lastChild ); subHp >= 0 ; --subHp )
            h.siftDown( subHp );
        return h;
    }
};
```

Complexity Analysis: Heapifying Arbitrary Array



Level	#Sub_Heaps	Heapify Cost
h	2^h	0
$h-1$	2^{h-1}	1
...	...	...
1	2	$h-1$
0	1	h

$$T(n) = 2^h \times 0 + 2^{h-1} \times 1 + \ldots + 2^0 \times h = \sum_{j=0}^{h} j2^{h-j} \quad \text{(E1)}$$

Useful Equations

- The power series:

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}; |x| < 1 \tag{PS1}$$

- Differentiating Equation (PS1) with respect to x yields:

$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1-x)^2}$$

multiplying by x:

$$\sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2} \tag{PS2}$$

Evaluating $T(n)$ & $O(n)$

From (E1) we estimated $T(n)$ as:

$$\begin{aligned} T(n) &= \sum_{j=0}^{h} j 2^{h-j} \\ &= \sum_{j=0}^{h} j \frac{2^h}{2^j} \\ &= 2^h \sum_{j=0}^{h} \frac{j}{2^j} \end{aligned}$$

Evaluating $T(n)$ & $O(n)$

$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j}$	$n + 1 = 2^{h+1}$ (E2)	$\sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2}$ (PS2)

plug $x = \frac{1}{2}$ in (PS2):

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \sum_{j=0}^{\infty} j(\frac{1}{2})^j = \frac{\frac{1}{2}}{(1 - \frac{1}{2})^2} = 2$$

therefore:

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j} < 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} = 2^h(2)$$

therefore:

$$T(n) < 2^{h+1}$$

From (E2):

$$T(n) < n + 1$$

therefore, the big-O notation of $T(n)$:

$$O(T(n)) = O(n)$$

Heap Applications: Heapsort

```cpp
template< typename T >
class Heap
{
public:
    ...
    static Heap make( std::vector< T > data )
    {
        Heap h;
        h.data.swap( data ); // O(1)
        ...
    }
    ...
};
```

```cpp
std::vector< int > heapSort( std::vector< int > a )
{
    auto h = Heap<int>::make( a ); // Heapify: O(n)
    a.clear();
    while( h.size() > 0 ) // O( n * log(n) )
        a.push_back( h.extract()); // O(log(n))
    return a;
}
```

Time Complexity: $O(T(n)) = O(n) + O(n \log(n)) = O(n \log(n))$

Heap Applications: Heapsort (avoiding $O(n)$ deep copy)

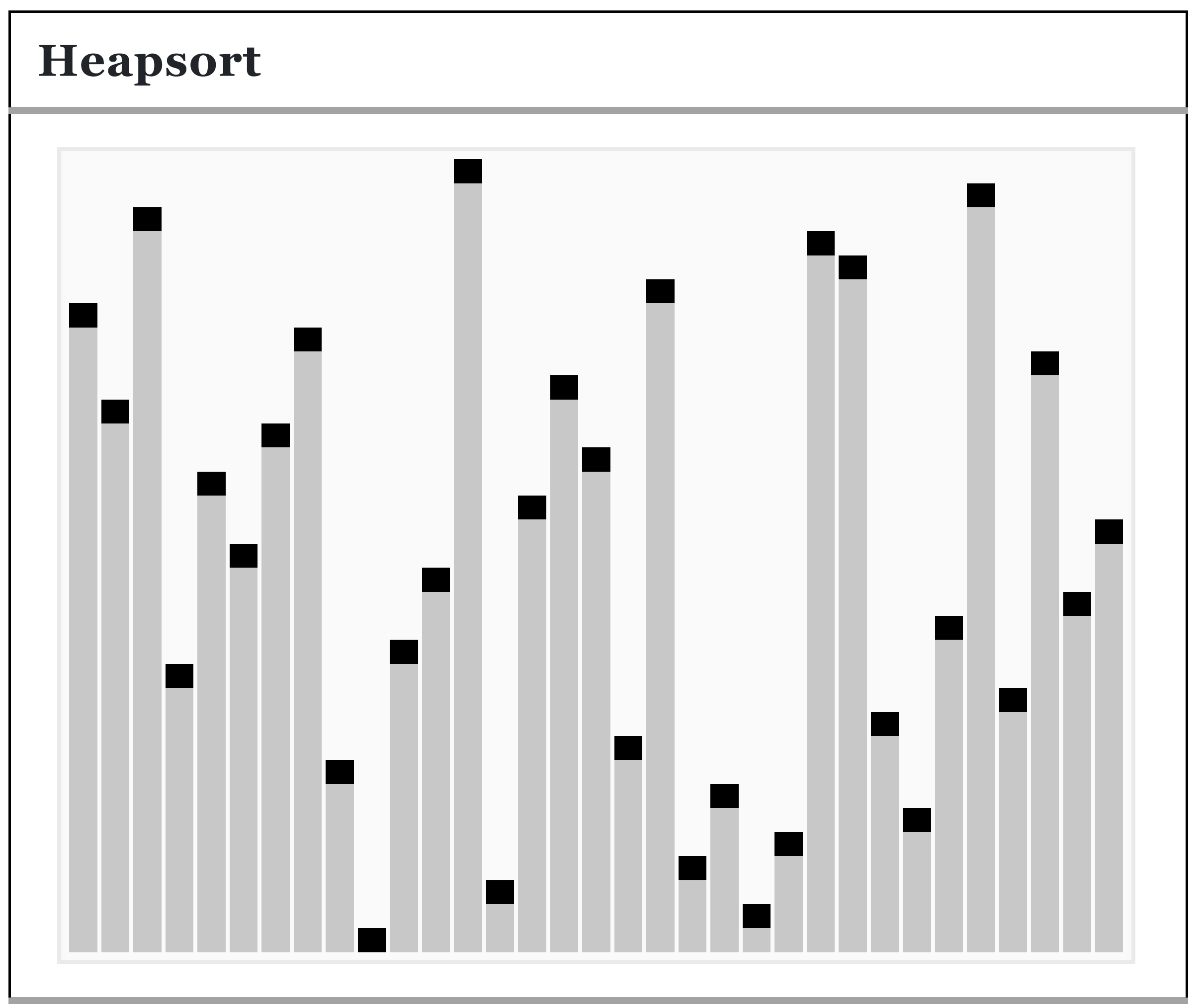
```
std::vector< int > heapSort( std::vector< int > a )
{
    auto h = Heap<int>::make( std::move( a ));
    while( h.size() > 0 )
        a.push_back( h.extract());
    return a;
}
```

To understand what is happening:

- {Advanced C++: Understanding rvalue and lvalue}
- {C++ 11: Rvalue Reference -- Move Semantics}

Visualization & Links

Heapsort



Visualization & Links

- {Heaps and Heap Sort}

Read the Notes

Read the notes for more details and to download the source files.

{sbme-tutorials.github.io/2020/data-structures/notes/week07.html}

Thank you