# Heaps

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2024

# Outline

# Outline

# Heaps

## Max Tree

A max tree is a tree in which

- the key value in each node $\geq$ the key values in its children.

# Heaps

## Max Tree

A max tree is a tree in which

- the key value in each node $\geq$ the key values in its children.

## Min Tree

A min tree is a tree in which

- the key value in each node $\leq$ the key values in its children.

# Heaps

## Max Tree

A max tree is a tree in which

- the key value in each node $\geq$ the key values in its children.

## Min Tree

A min tree is a tree in which

- the key value in each node $\leq$ the key values in its children.

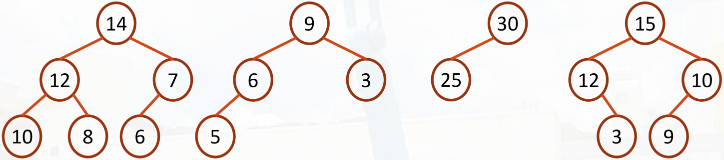
## Max Heap

A complete binary tree that is also a max tree.

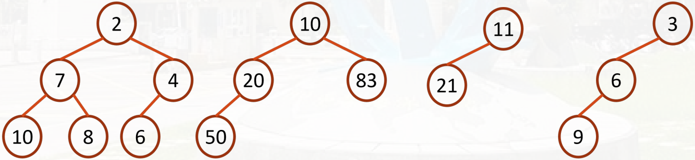
## Min Heap

A complete binary tree that is also a min tree.

# Examples: Max & Min Trees

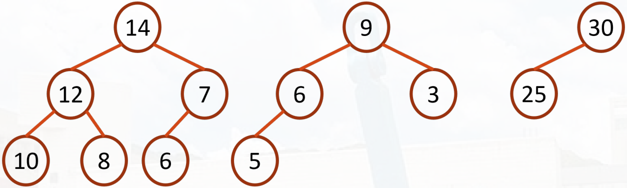



Max Trees

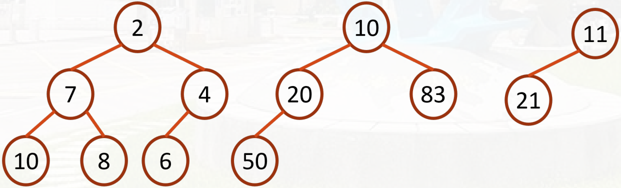



Min Trees

# Examples: Max & Min Heaps




Max Heaps




Min Heaps

# The Key Application: Priority Queues

- Heaps are frequently used to implement priority queues.
- In this kind of queue,
  - the element to be deleted is the one with highest (or lowest) priority.
  - at **any time**, an element with arbitrary priority can be **inserted** into the queue.

# Some Important Notes

- It's straightforward to implement a heap using an array (WHY?).

# Some Important Notes

- It's straightforward to implement a heap using an array (WHY?).
- Insert the new node next to the last element in the array.

# Some Important Notes

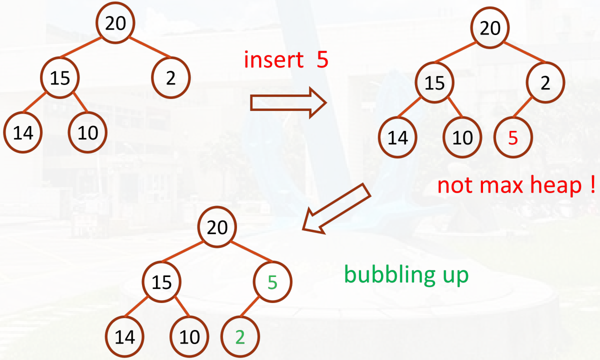
- It's straightforward to implement a heap using an array (WHY?).
- Insert the new node next to the last element in the array.
- A heap is a complete binary tree.

# Insertion into a Max Heap

- The bubbling process.
  - It begins at the new node of the tree and moves toward the root.

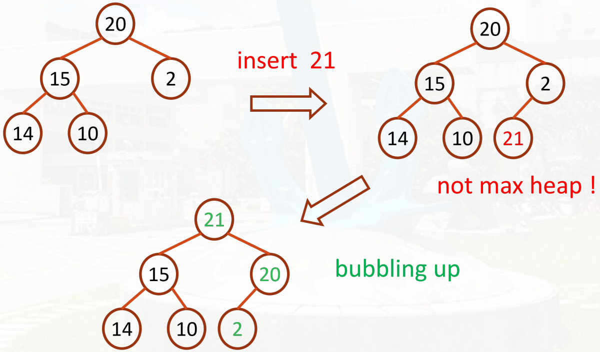
# Insertion into a Max Heap

- The bubbling process.
  - It begins at the new node of the tree and moves toward the root.



# Insertion into a Max Heap

- The bubbling process.
  - It begins at the new node of the tree and moves toward the root.



# The Code for Insertion into a Max Heap

- Consider the following declarations:

```
#define MAX_ELEMENTS 200  /* maximum heap size+1 */
#define HEAP_FULL (n) (n == MAX_ELEMENTS -1)
#define HEAP_EMPTY (n)  (!n)
typedef struct {
    int key;
    /* other fields */
} element;
element heap[MAX_ELEMENTS];
int n = 0;
```

# The Code for Insertion into a Max Heap

```c
void push (element item, int *n) {
/* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        printf("The heap is full.\n");
        exit(EXIT_FAILURE);
    } // O(1) time
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    } //  O(lg n) time
    heap[i] = item; // O(1) time
}
```

- The time complexity of the insertion: $O(\lg n)$.

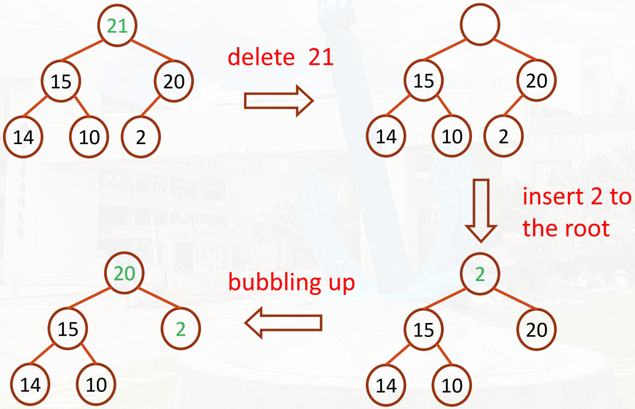
# Deletion from a Max Heap

- When an element is to be deleted from a max heap, it is ALWAYS taken from the root of the heap.

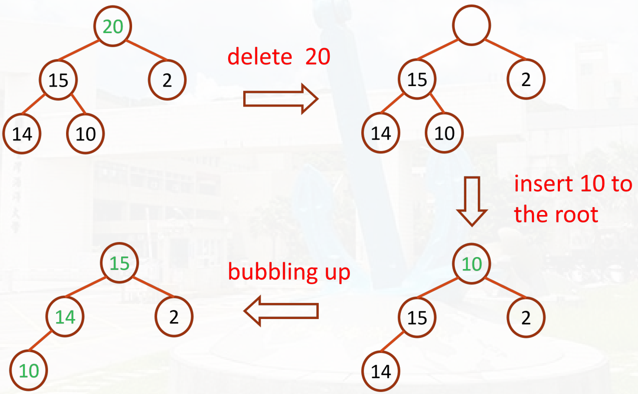
# Deletion from a Max Heap

- When an element is to be deleted from a max heap, it is ALWAYS taken from the root of the heap.
- The steps of deletion from a Max heap:
  - delete the root node.
  - insert the last node into the root (say $r$).
  - use the bubbling up process to ensure that the resulting heap remains a max heap (a.k.a. heapify at $r$).

# Illustration of Deletion from a Max Heap



# Illustration of Deletion from a Max Heap



# The Code for Deletion from a Max Heap

```
element pop(int *n) {
/* delete element with the highest key from the heap */
    int parent, child;
    element item, temp;
    if (HEAP_EMPTY(*n)) {
        fprintf(stderr, "The heap is empty\n");
        exit(EXIT_FAILURE);
    }
    /* save value of the element with the highest key */
    item = heap[1];
    /* use last element in heap to adjust heap */
    temp = heap[(*n)--];
    parent = 1;
    child = 2; // default: the left child
    while (child <= *n) { // O(lg n) time
    /* find the larger child of the current parent */
        if ((child < *n) && (heap[child].key < heap[child+1].key))
            child++; // okay, it's the right child!
        if (temp.key >= heap[child].key) break; // the new root is the maximum!
        /* if the max-child gets larger key, move to the next lower level */
        heap[parent] = heap[child];
        parent = child;
        child *= 2;
    }
    heap[parent] = temp;
    return item;
}
```

# Time Complexity of the Deletion from a Max Heap

- Delete the root node: $O(1)$.
- Insert the last node to the root: $O(1)$.
- Since the height of the heap is $\lceil \lg(n+1)) \rceil$, the while loop is iterated for $O(\lg n)$ times.
- Thus, the overall time complexity: the time complexity of the deletion: $O(\log n)$.

# Outline

1. Introduction
   - Building a heap

# How to build a heap for a set of $n$ input numbers?

- For each input number $x$, execute `push`(x).

# How to build a heap for a set of $n$ input numbers?

- For each input number $x$, execute `push`$(x)$.

- The above process is correct and requires

# How to build a heap for a set of $n$ input numbers?

- For each input number $x$, execute `push`(x).

- The above process is correct and requires $O(n \log n)$ time.

# An $O(n)$ time algorithm for building a (max) heap

Input: $n$ numbers: $x_1, x_2, \ldots, x_n$.

## Efficient Heap Construction

1. For each input number $x_i$, insert $x_i$ into array $A$ at $A[i-1]$ one by one.
2. For $i = \lfloor n/2 \rfloor - 1$ down to 0:
   - Run $\texttt{heapify}(A, i)$

# An $O(n)$ time algorithm for building a (max) heap

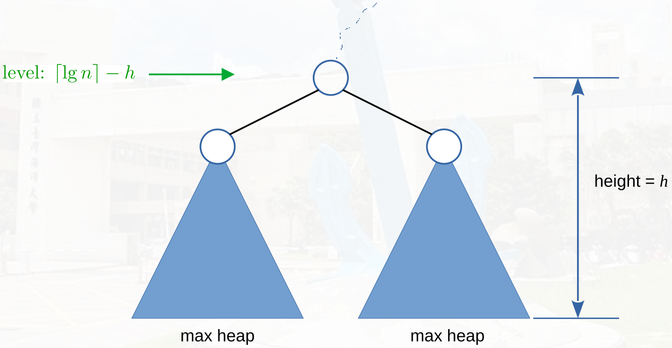
Input: $n$ numbers: $x_1, x_2, \ldots, x_n$.

## Efficient Heap Construction

1. For each input number $x_i$, insert $x_i$ into array $A$ at $A[i-1]$ one by one.
2. For $i = \lfloor n/2 \rfloor - 1$ down to 0:
   - Run $\texttt{heapify}(A, i)$

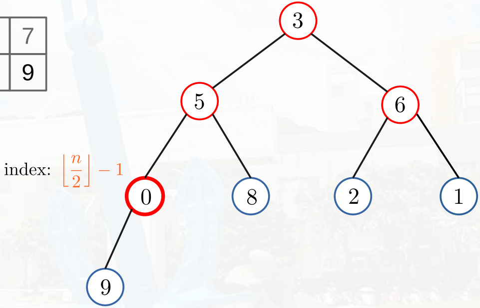
- That is, we build a heap in a bottom-up fashion!

# Heap recursive view (bottom-up)



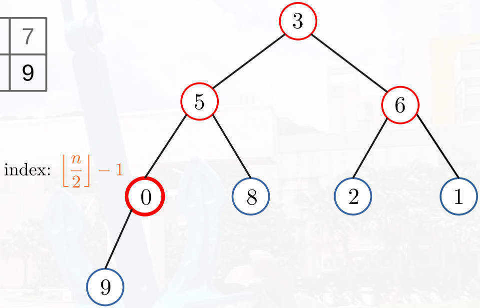
# Nodes to be Heapified

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 3 | 5 | 6 | 0 | 8 | 2 | 1 | 9 |



# Nodes to be Heapified

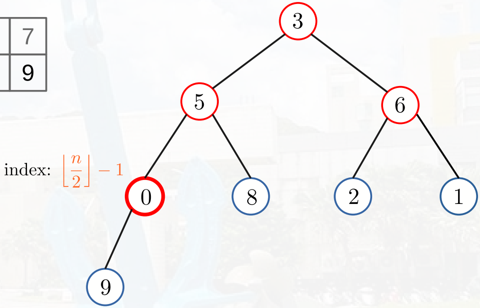
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 3 | 5 | 6 | 0 | 8 | 2 | 1 | 9 |




- # Heapify steps: $\leq \sum_{h=1}^{\lg_n -1} h \cdot n_h =$

# Nodes to be Heapified

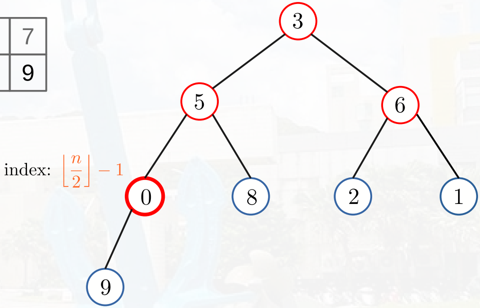
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 3 | 5 | 6 | 0 | 8 | 2 | 1 | 9 |




- # Heapify steps: $\leq \sum_{h=1}^{\lg_n -1} h \cdot n_h = \sum_{h=1}^{\lg_n -1} h \cdot 2^{\lceil \lg n \rceil - h}$

# Nodes to be Heapified

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 3 | 5 | 6 | 0 | 8 | 2 | 1 | 9 |




- # Heapify steps: $\leq \sum_{h=1}^{\lg_n -1} h \cdot n_h = \sum_{h=1}^{\lg_n -1} h \cdot 2^{\lceil \lg n \rceil - h} \leq 2n \sum_{h=1}^{\lg n - 1} \frac{h}{2^h}$.
  - $n_h$: the number of nodes at level $h$.

# Discussions