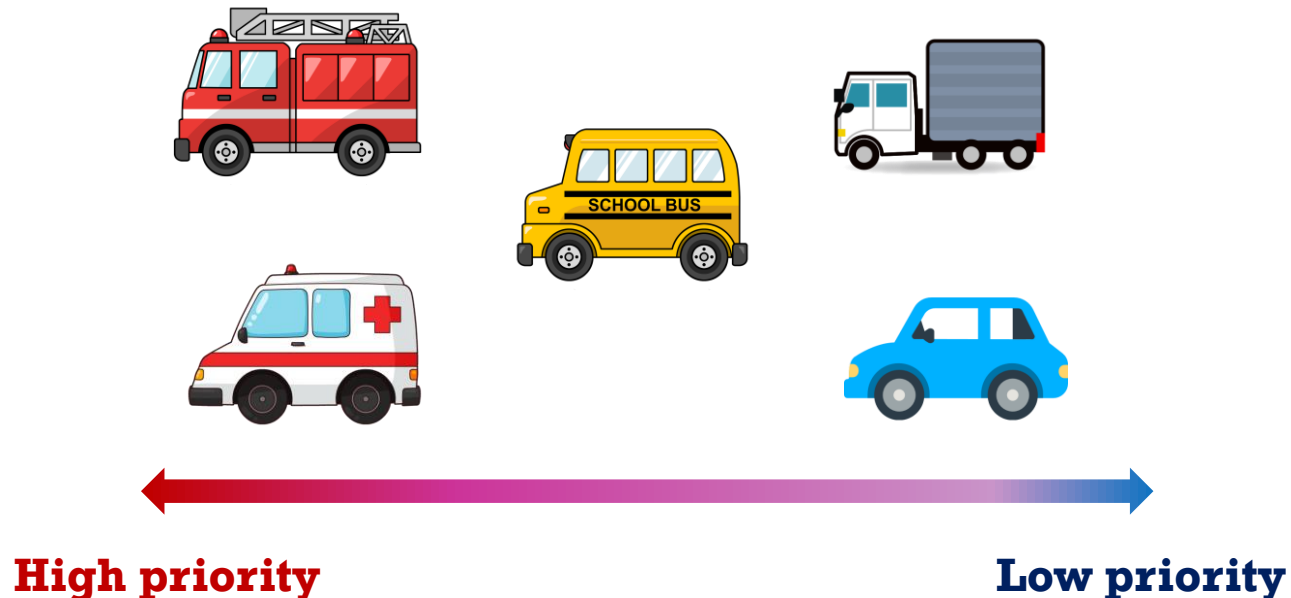


Priority Queue and Heap

What is Priority Queue?

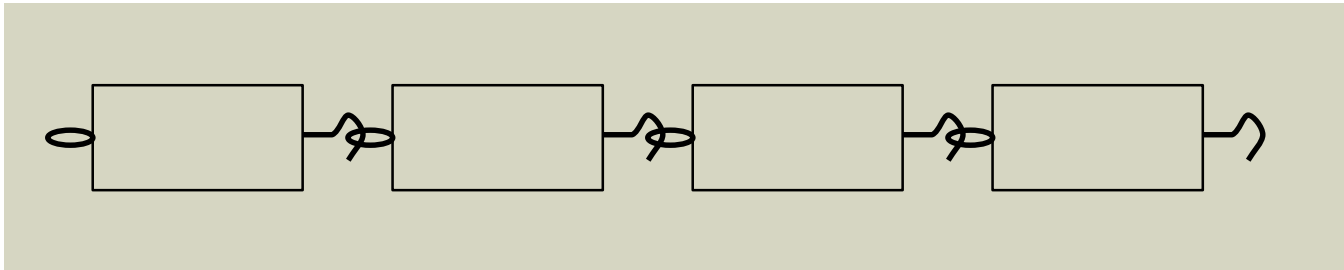
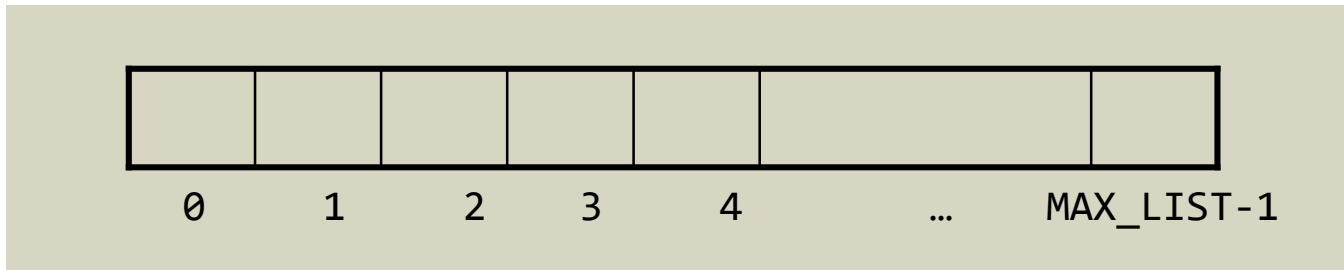
■ Definition

- Similar to a regular queue (FIFO)
- **The key difference is that each element has a **priority**.**
 - An element with high priority is served before an element with low priority.



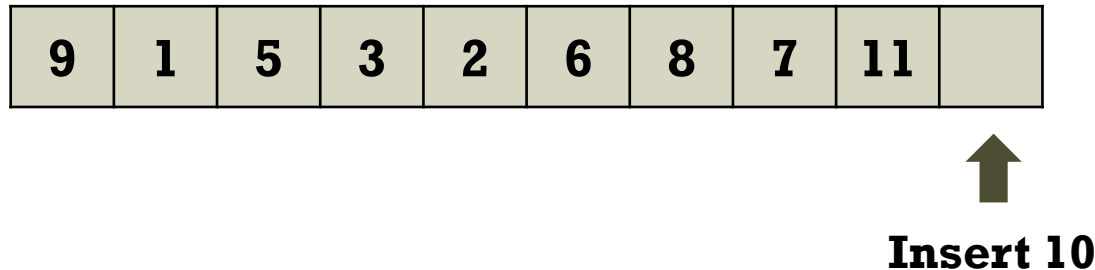
Implementing Priority Queue

- How to implement a priority queue
 - Array-based implementation
 - Linked-list-based implementation
 - Heap-based implementation



Implementing Priority Queue

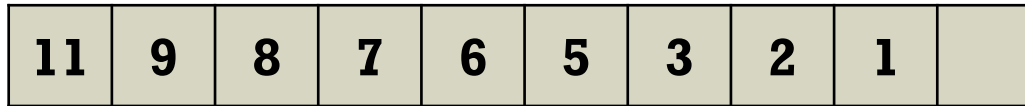
- How to use an **unsorted** array/linked list
 - Insert elements to the last position.
 - Search an element with the highest priority sequentially.



- Time complexity
 - Insert an arbitrary element: $O(1)$
 - Search the max: $O(n)$
 - Remove the max: $O(n)$

Implementing Priority Queue

- How to use a **sorted** array/linked list
 - Insert an element by descending order.
 - Search the first element simply.



- Time complexity
 - Insert an arbitrary element: $O(n)$
 - Search the max: $O(1)$
 - Remove the max: $O(n)/O(1)$

Implementing Priority Queue

■ Comparison for various implementations

Representation	Search max	Delete max	Insert
Unordered array	$O(n)$	$O(n)$	$O(1)$
Unordered linked list	$O(n)$	$O(n)$	$O(1)$
Ordered array	$O(1)$	$O(n)$	$O(n)$
Ordered linked list	$O(1)$	$O(1)$	$O(n)$
Max heap	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$



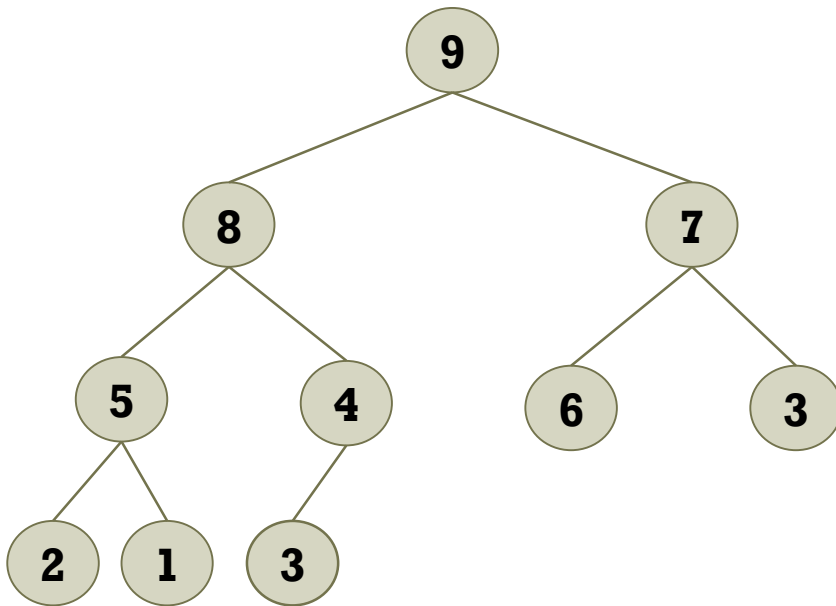
What is Heap?

- Definition (max heap)

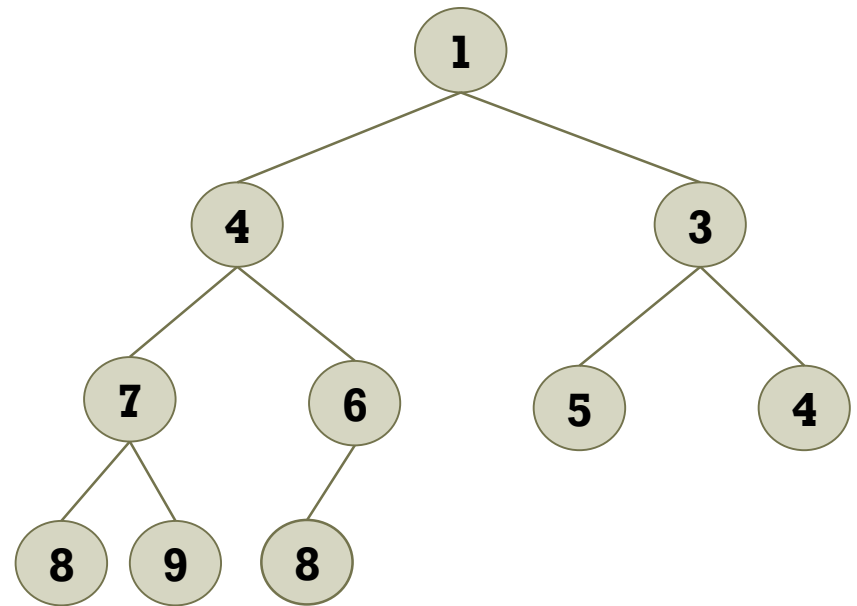
- **Complete binary tree**

- The value of root is the **maximum** in the tree.

- The subtrees are also heaps: **$key(parent) \geq key(child)$**



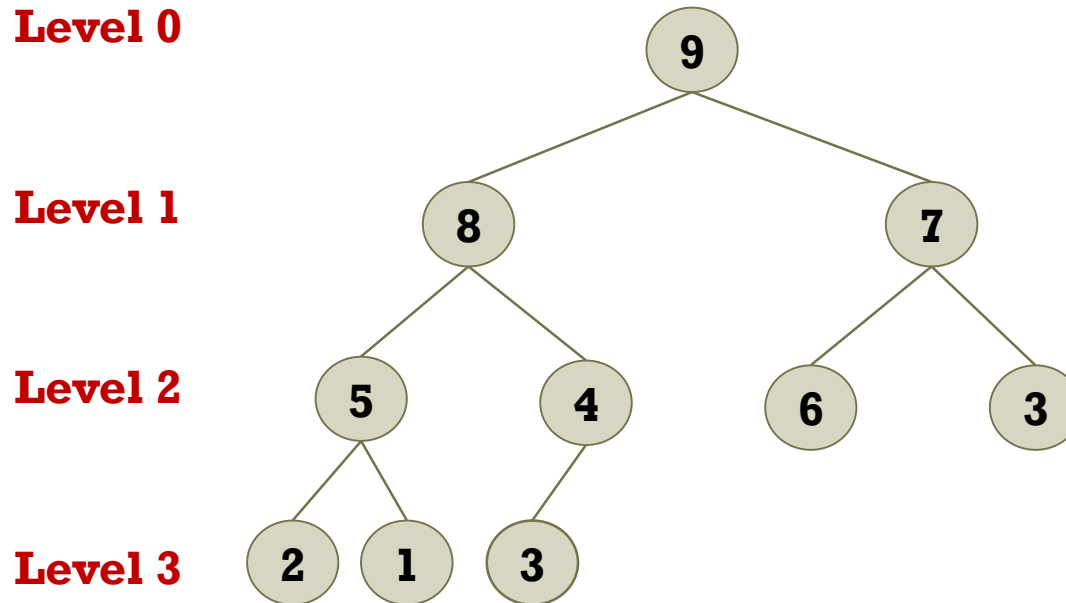
Max heap



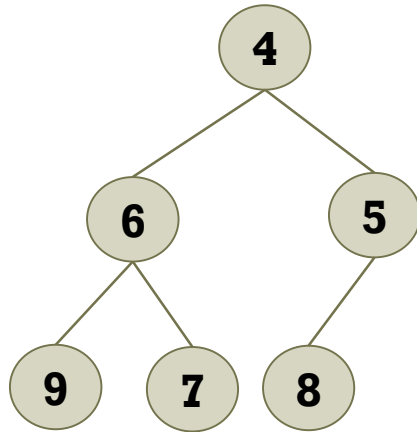
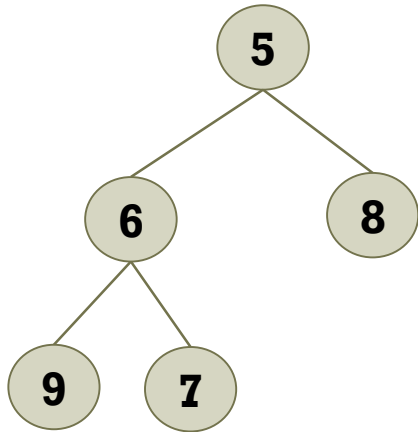
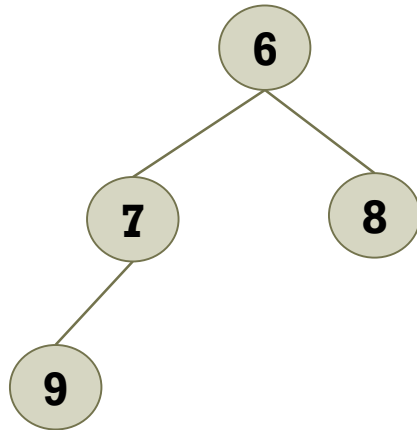
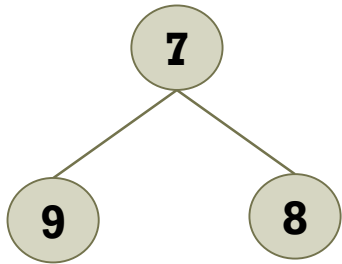
Min heap

What is Heap?

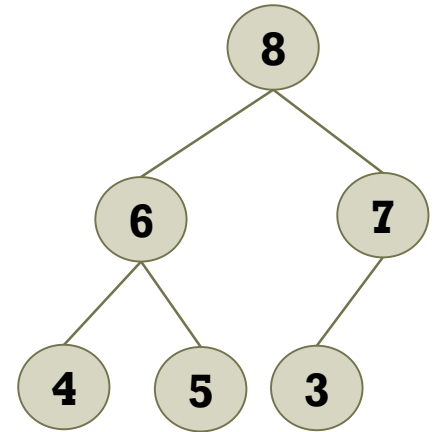
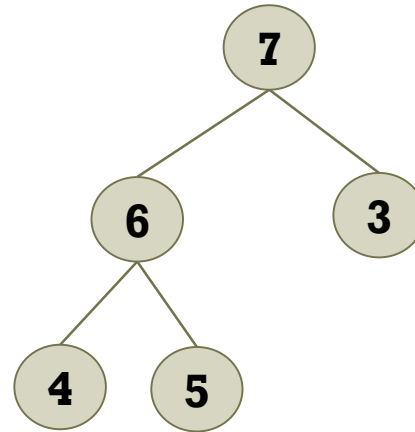
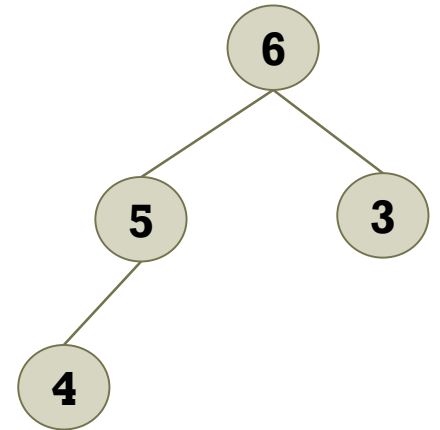
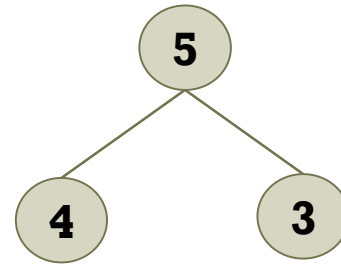
- Given the heap has n nodes, its height is $O(\log_2 n)$.
 - **A complete binary tree holds.**
 - For each level i , 2^i nodes exist except for the last level.



Examples of Heap



Min heaps

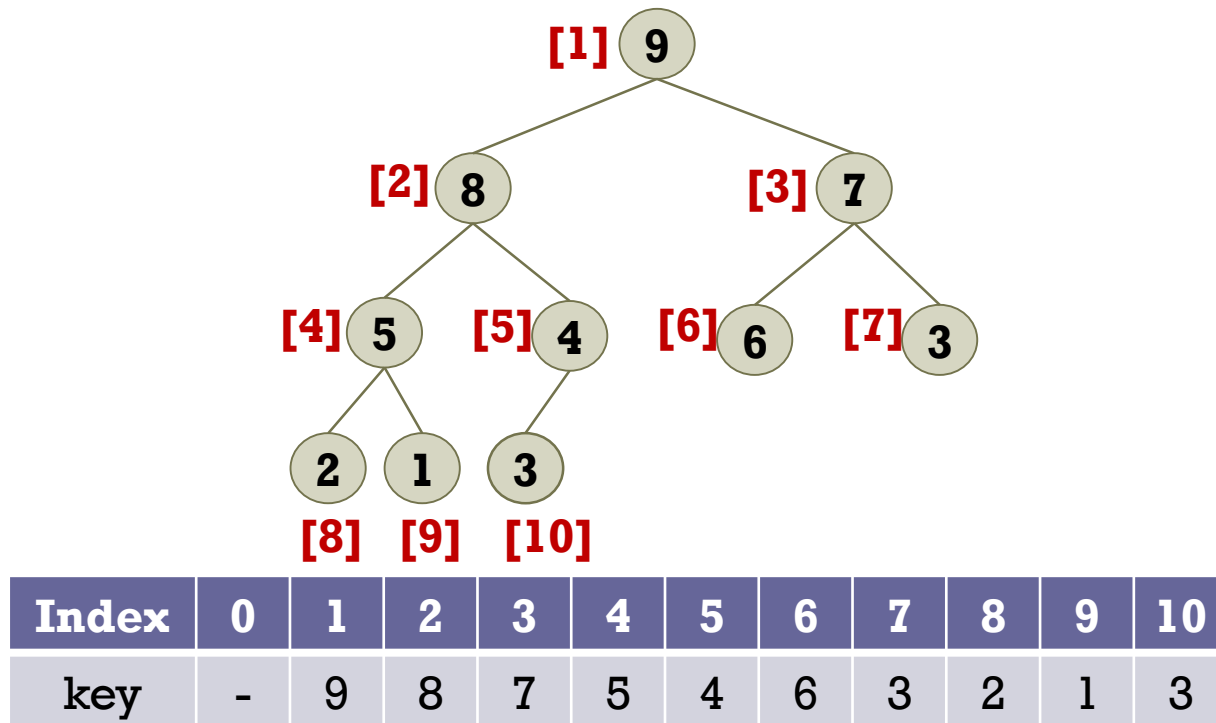


Max heaps

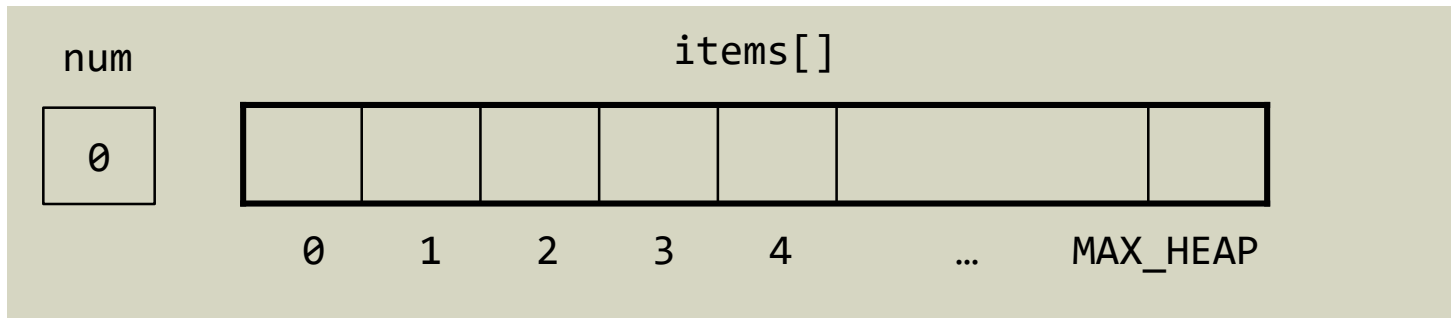
Array-based Implementation

■ Heap representation

- Because heap is a complete binary tree, each node has an index in sequence.
- Suppose that the index of a node is x .
 - **Index of a left-child node: $2x$**
 - **Index of a right-child node: $2x + 1$**



Array-based Implementation



```
#define MAX_HEAP    100

typedef enum { false, true } bool;
typedef char Data;

typedef struct {
    Data data;
    int priority;
} HNode;

typedef struct {
    HNode items[MAX_HEAP + 1];
    int num;
} Heap;
```

Array-based Implementation

```
/ Make a heap empty.
void InitHeap(Heap *pheap);
// check whether a heap is empty.
bool IsEmpty(Heap *pheap);
// Check whether a heap is full.
bool IsFull(Heap *pheap);

// Insert a node to the heap.
void Insert(Heap *pheap, Data data, int priority);
// Remove the maximum data from the heap.
Data Delete(Heap *pheap);

// Get a parent index for a given index.
int GetParent(int idx);
// Get a left child index for a given index.
int GetLChild(int idx);
// Get a right child index for a given index.
int GetRChild(int idx);
// Get a child index with high priority between two child nodes.
int GetHighPriorityChild(Heap* pheap, int idx);
```

Initializing Heap

■ InitHeap, IsEmpty, and IsFull operations

```
// Make a heap empty.
void InitHeap(Heap *pheap)
{
    pheap->num = 0;
}

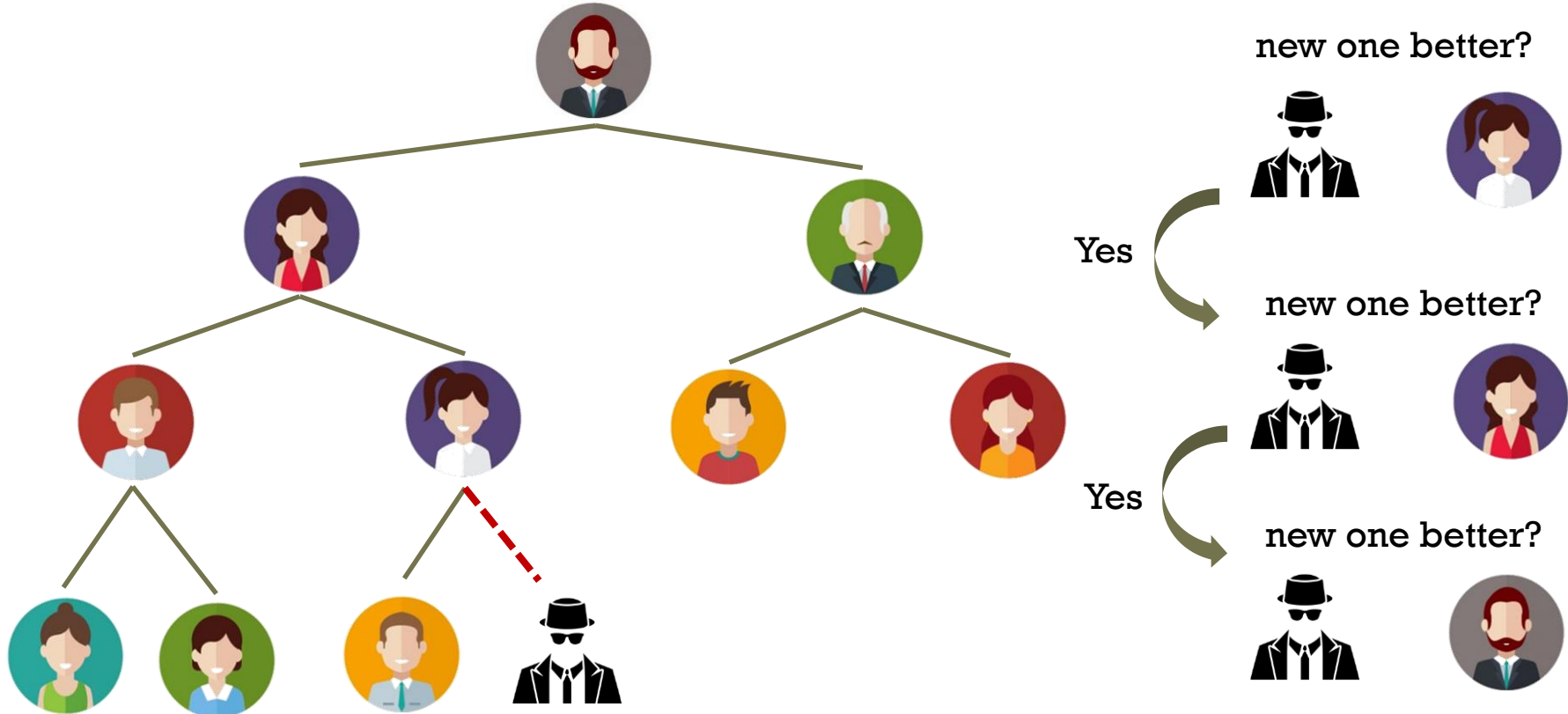
// check whether a heap is empty.
bool IsEmpty(Heap *pheap)
{
    return pheap->num == 0;
}

// Check whether a heap is full.
bool IsFull(Heap *pheap)
{
    return pheap->num == MAX_HEAP;
}
```

Insertion in Heap

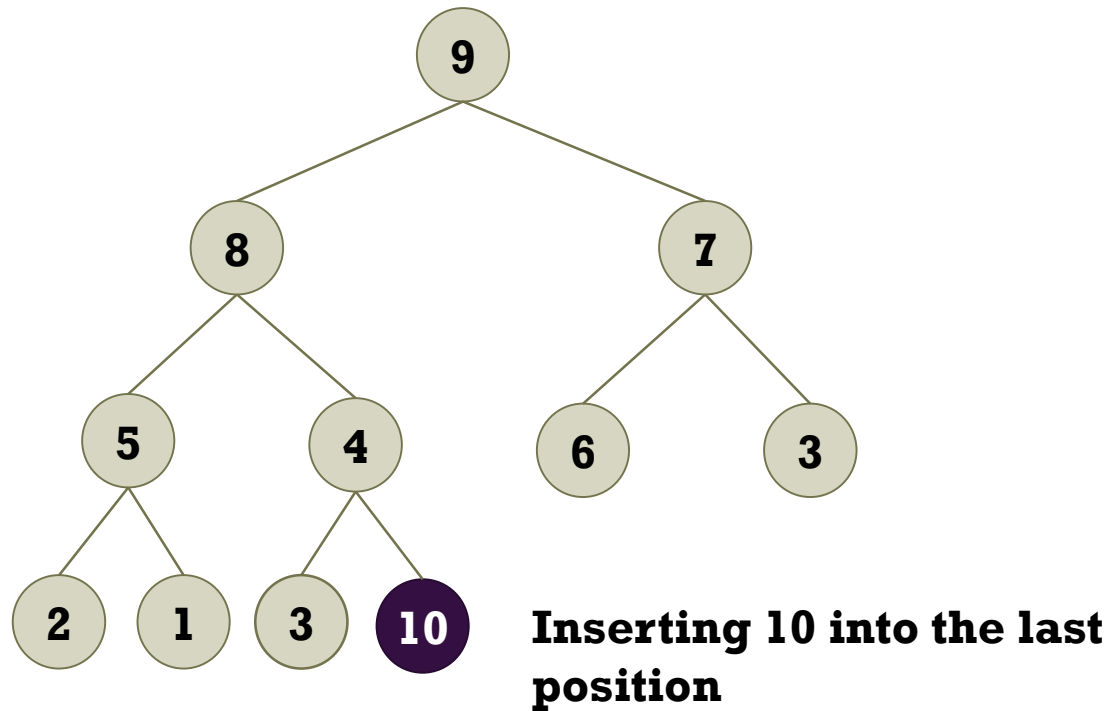
■ Key idea

- Insert a new node to the last position.
- Check if the new node is better than its ancestors.
- <https://visualgo.net/en/heap>



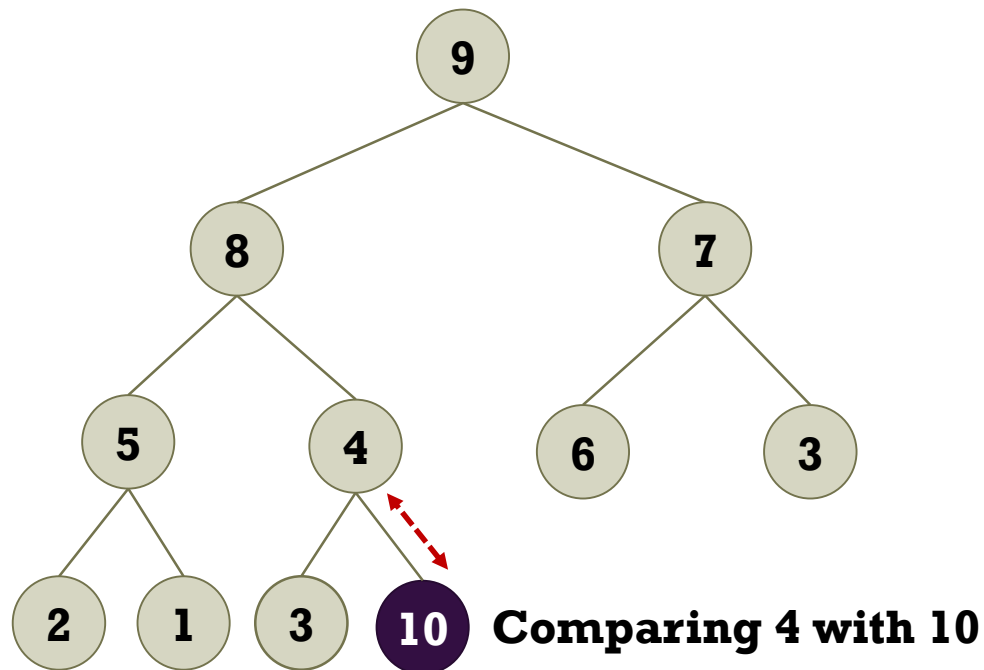
Insertion in Heap

- Step 1: Inserting a new node
 - Insert a new node into a bottom-rightmost leaf node available.



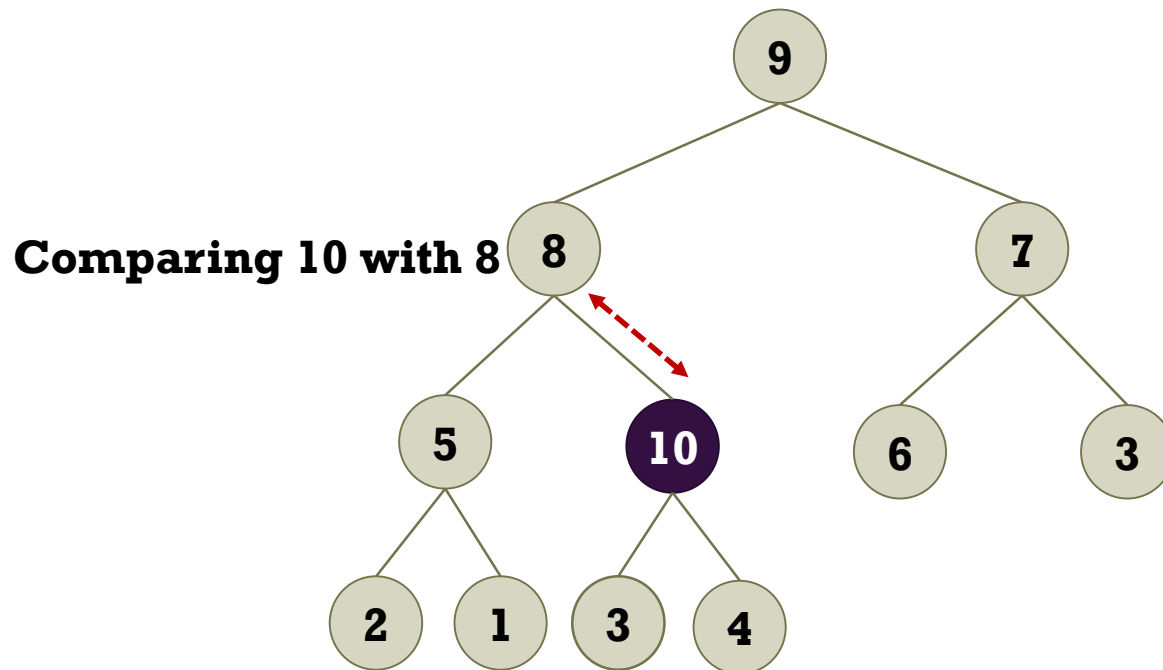
Insertion in Heap

- Step 2: Updating the heap
 - Compare the new node with its parent.
 - If the new node is higher priority than its parent, exchange them.
 - Do this until the new parent is greater than the new node or the new node becomes the root.



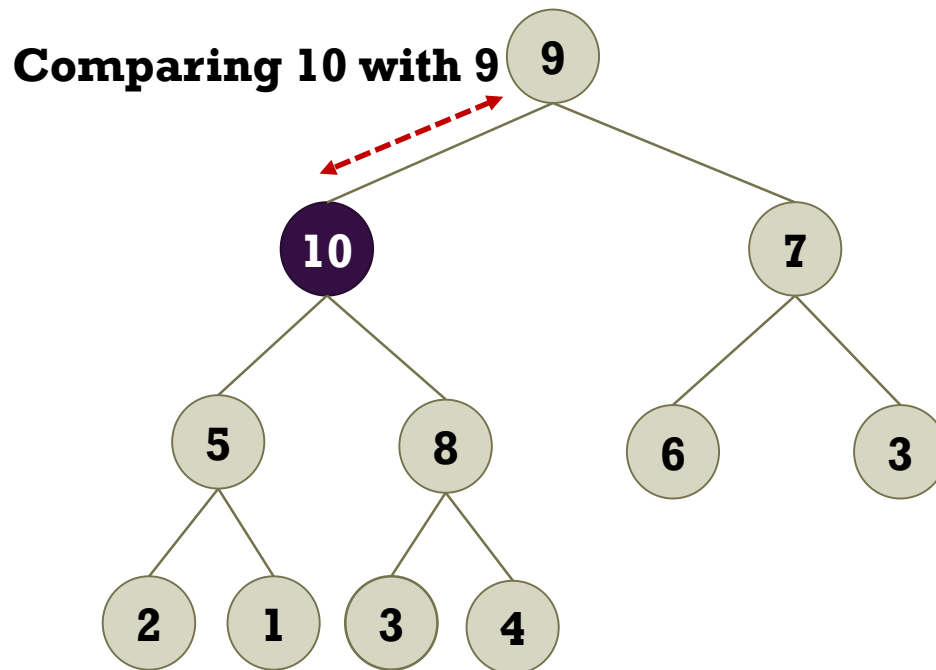
Insertion in Heap

- Step 2: Updating the heap
 - Compare the new node with its parent.
 - If the new node is higher priority than its parent, exchange them.
 - Do this until the new parent is greater than the new node or the new node becomes the root.



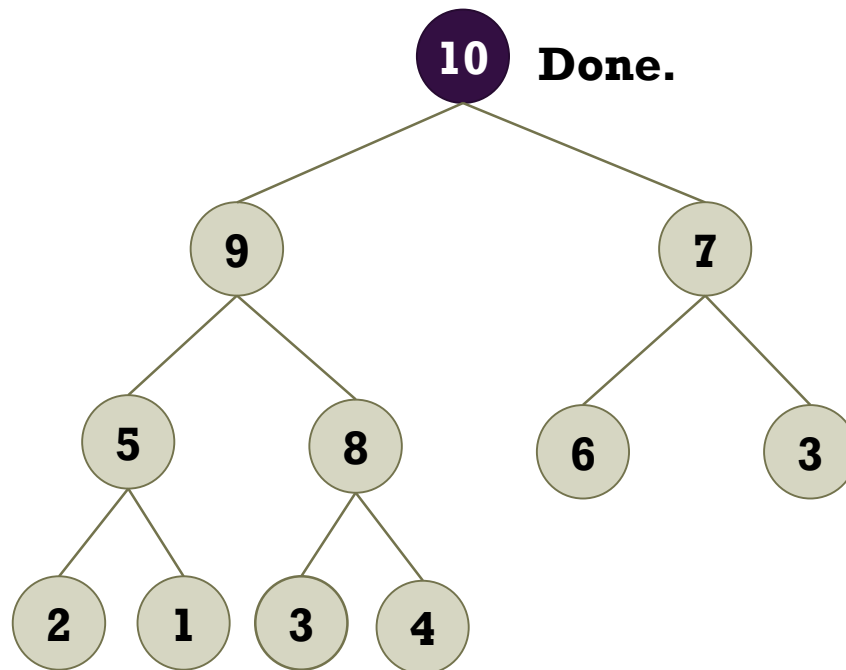
Insertion in Heap

- Step 2: Updating the heap
 - Compare the new node with its parent.
 - If the new node is higher priority than its parent, exchange them.
 - Do this until the new parent is greater than the new node or the new node becomes the root.



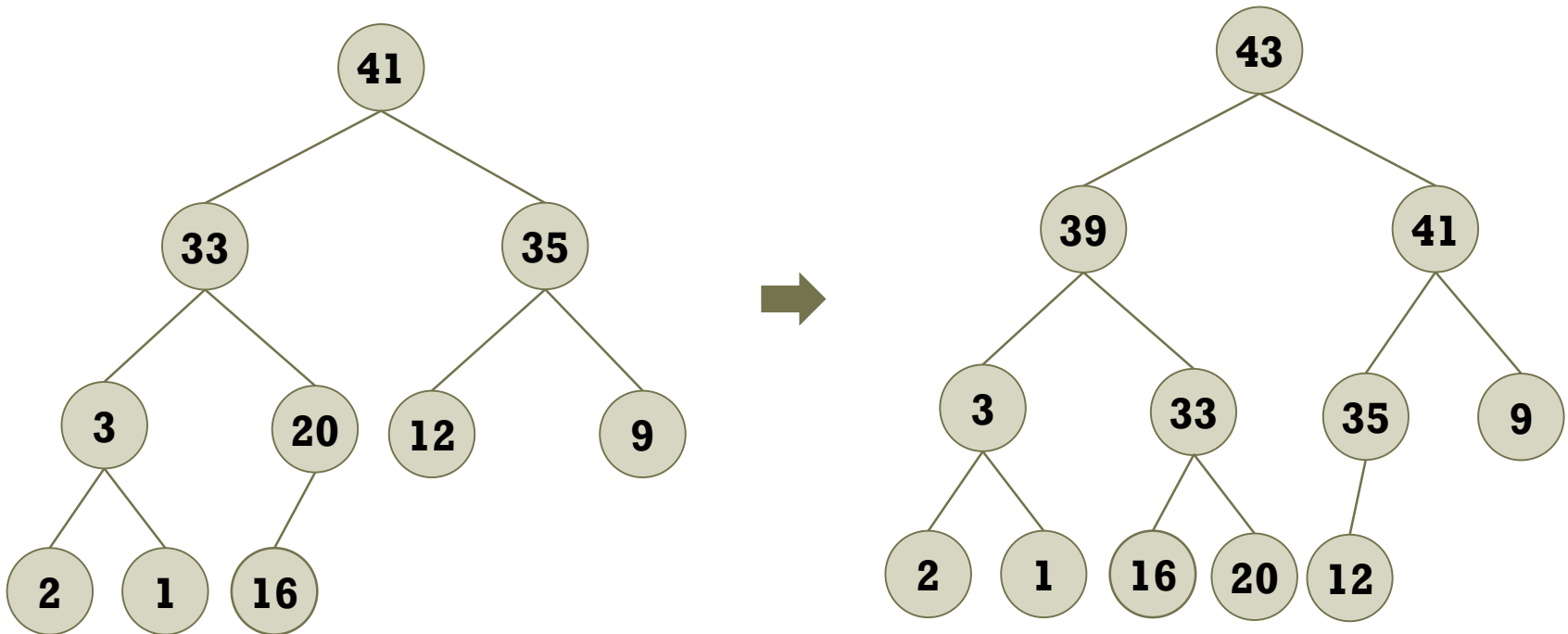
Insertion in Heap

- Step 2: Updating the heap
 - Compare the new node with its parent.
 - If the new node is higher priority than its parent, exchange them.
 - Do this until the new parent is greater than the new node or the new node becomes the root.



Exercise: Insertion in Heap

- Inserting 39 and 43 to the heap



- What is the time complexity for insertions?

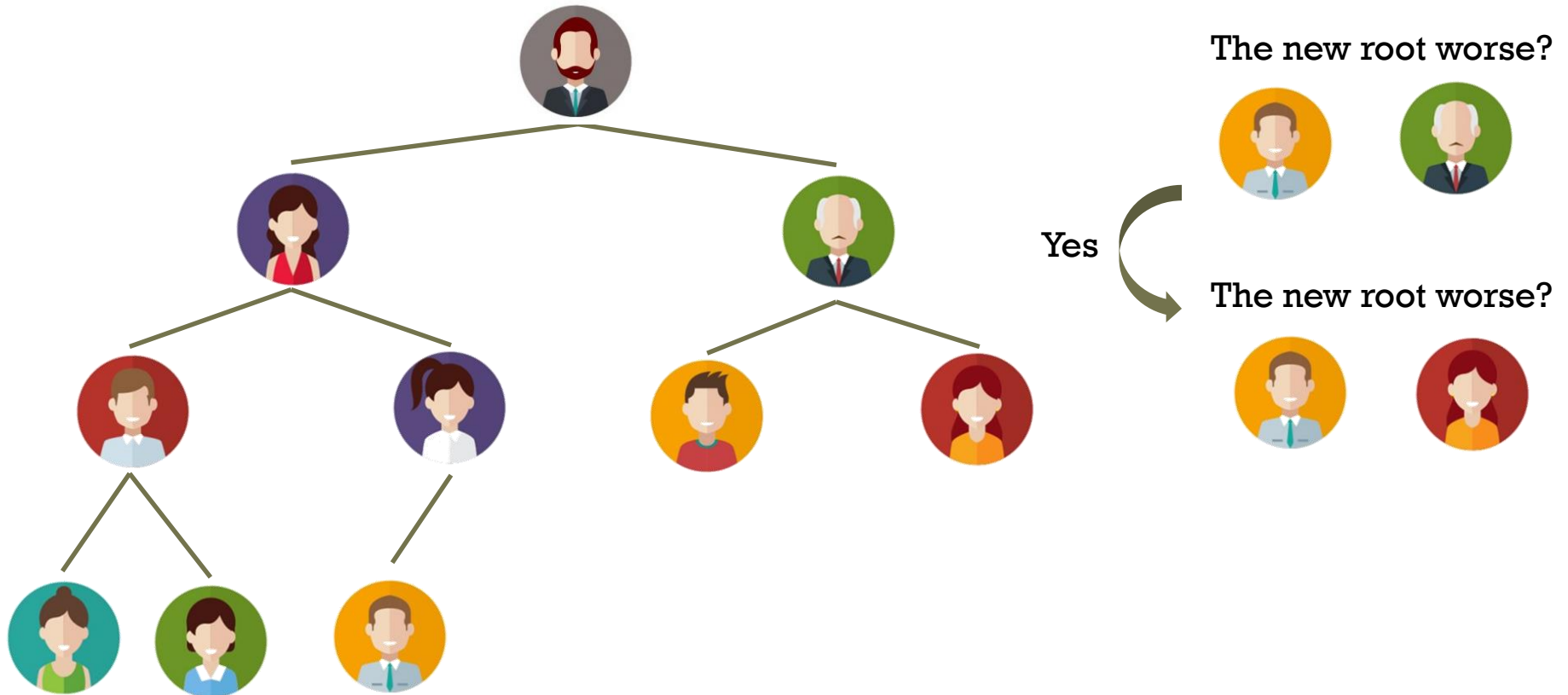
Implementation of Insertion

```
void Insert(Heap *pheap, Data data, int priority)
{
    HNode newNode;
    int idx = pheap->num + 1;
    if (IsFull(pheap)) exit(1);
    // Compare the new node with its parent.
    while (idx > 1) {
        int parent = GetParent(idx);
        if (priority > pheap->items[parent].priority) {
            pheap->items[idx] = pheap->items[parent];
            idx = parent;
        }
        else break;
    }
    newNode.data = data;
    newNode.priority = priority;

    pheap->items[idx] = newNode;
    pheap->num++;
}
```

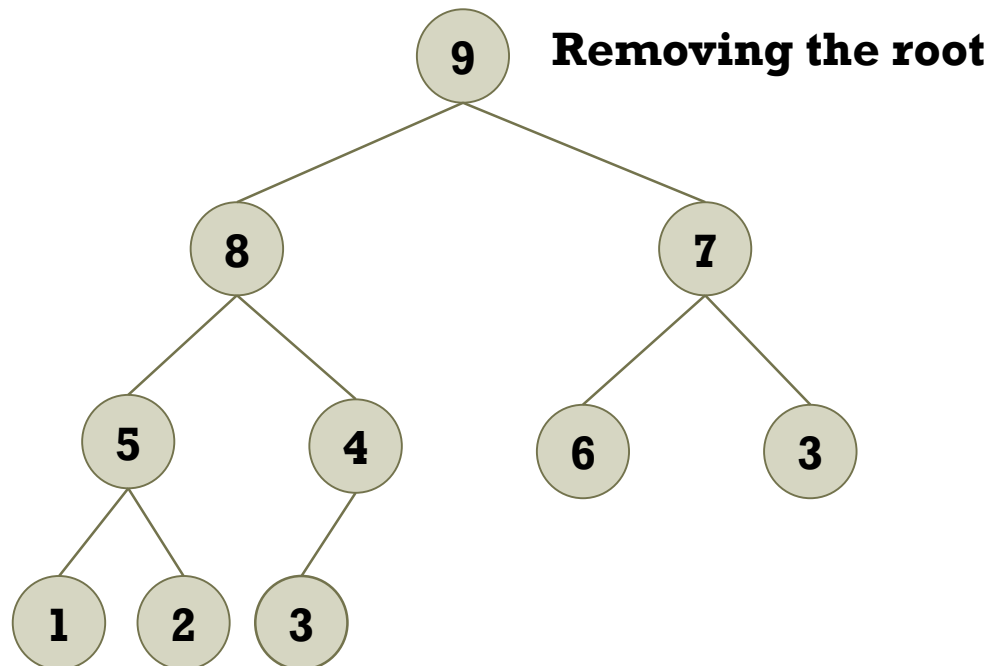
Deletion in Heap

- Key idea
 - Replace the last node into the root
 - Check if the new root is better than its descendants.
 - <https://visualgo.net/en/heap>



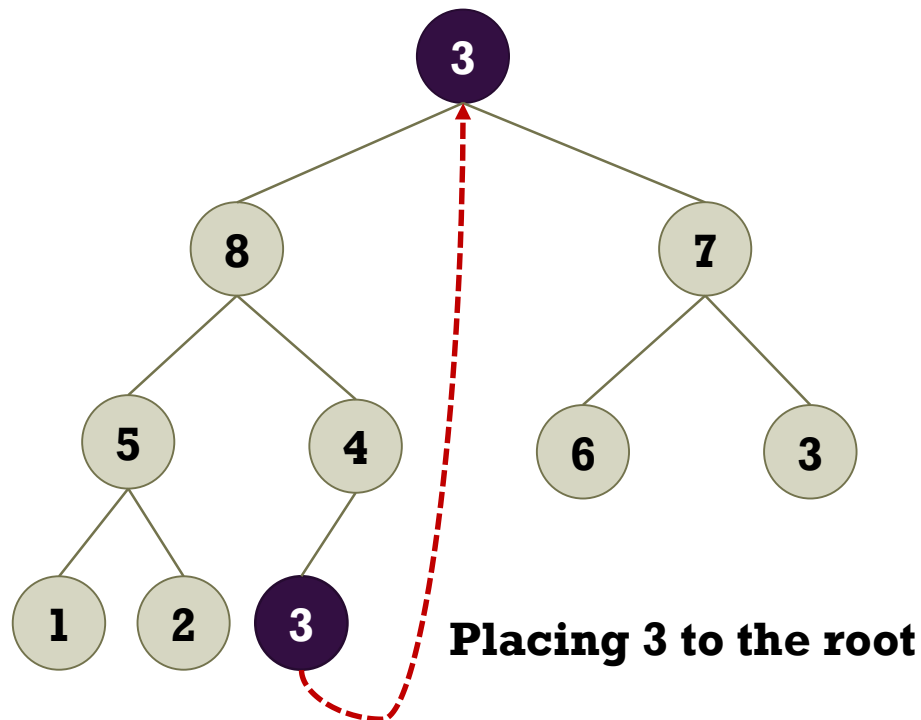
Deletion in Heap

- Step 1: Removing the root from the heap
 - Remove the max node from the root of the heap.
 - Place the last node to the root.



Deletion in Heap

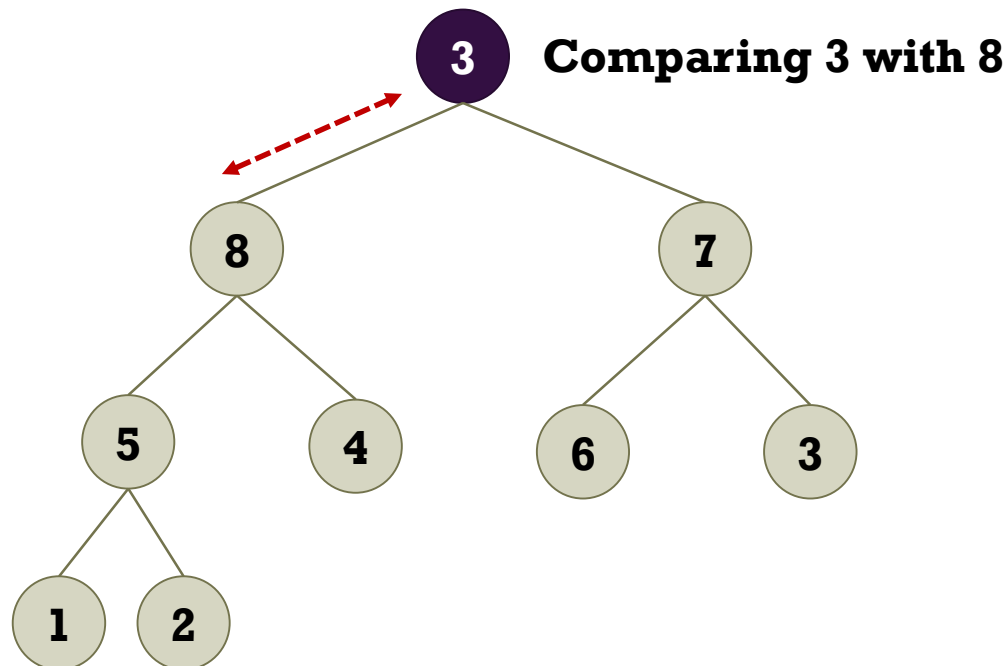
- Step 1: Removing the root from the heap
 - Remove the max node from the root of the heap.
 - Place the last node to the root.



Deletion in Heap

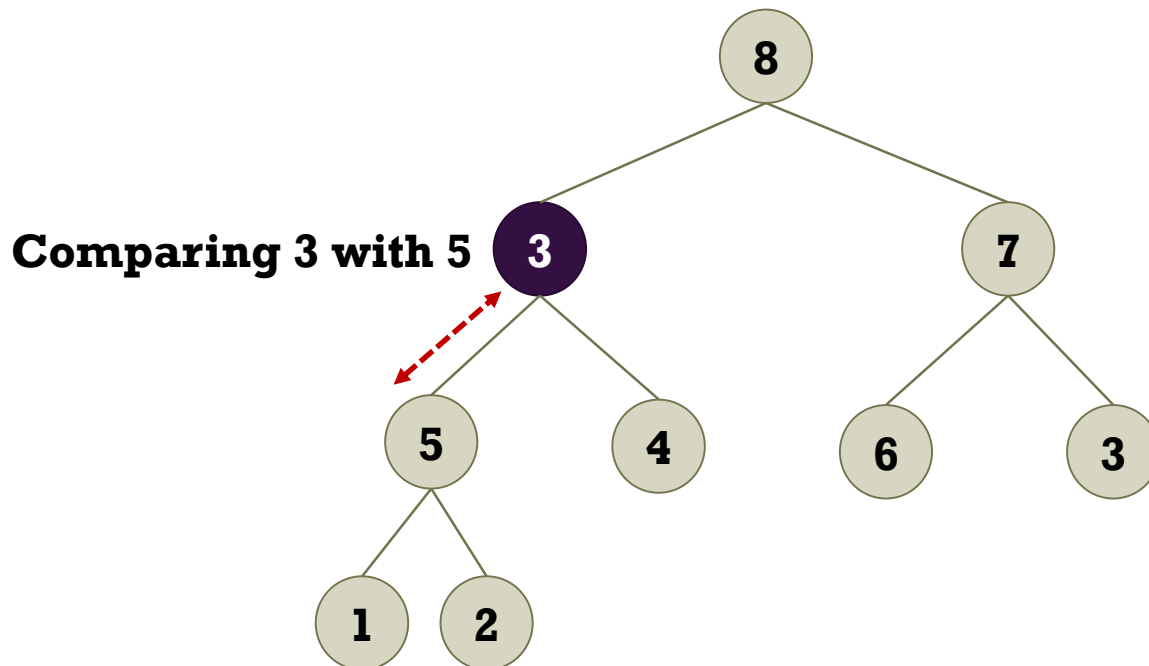
■ Step 2: Updating the heap

- Compare the root with its children.
 - If it is smaller than any of children, exchange the position with its max child.
- Do this until the heap is reestablished.



Deletion in Heap

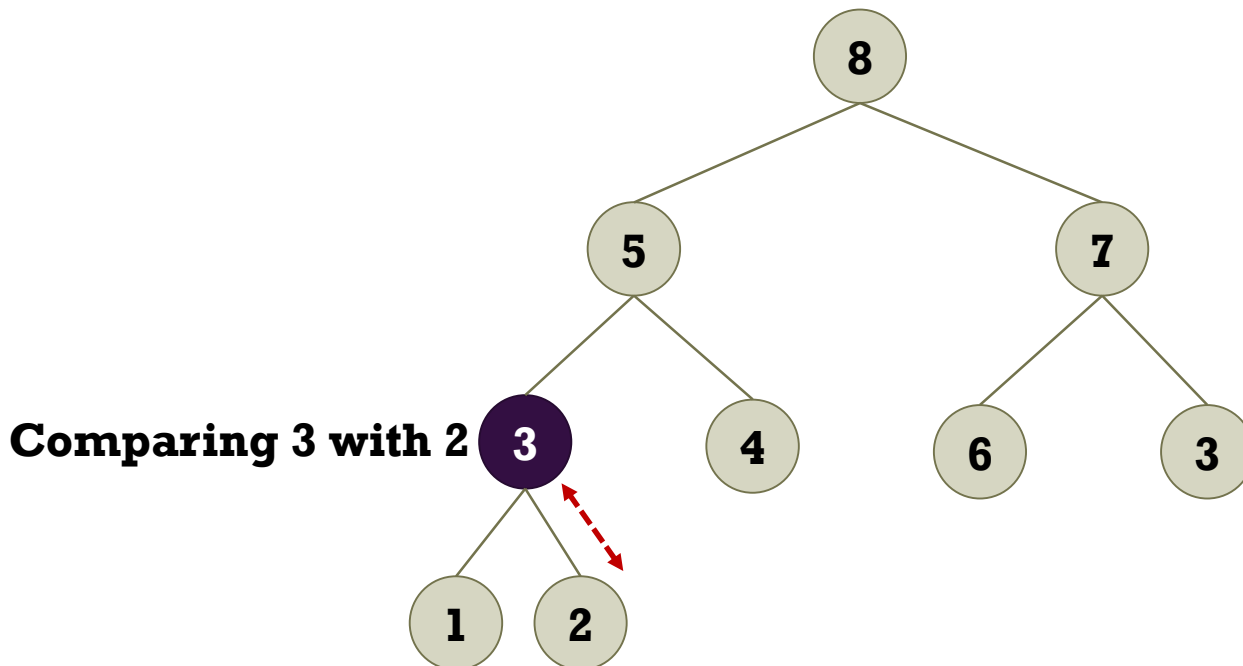
- Step 2: Updating the heap
 - Compare the root with its children.
 - If it is smaller than any of children, exchange the position with its max child.
 - Do this until the heap is reestablished.



Deletion in Heap

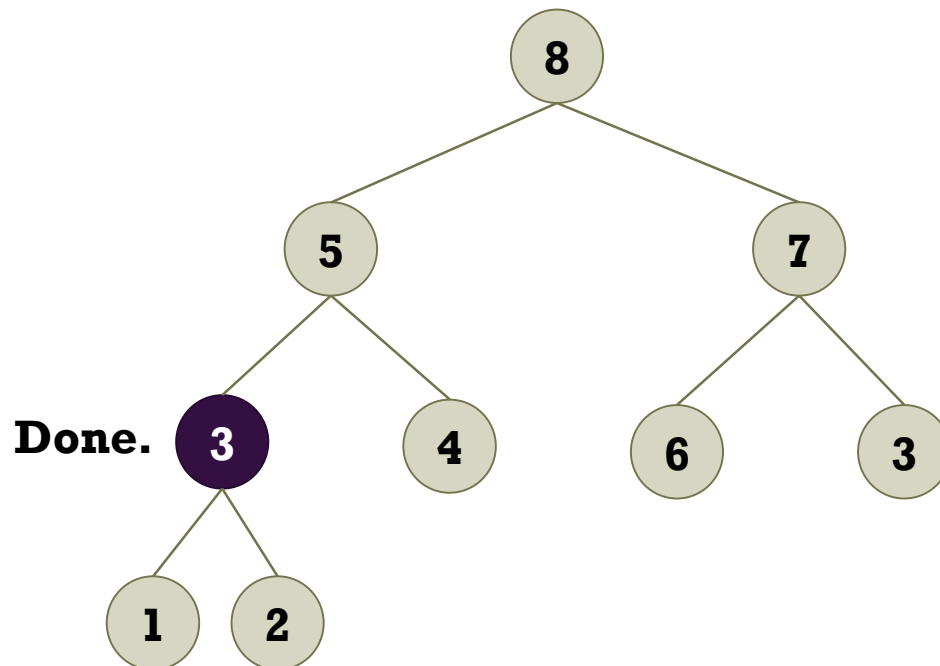
■ Step 2: Updating the heap

- Compare the root with its children.
 - If it is smaller than any of children, exchange the position with its max child.
- Do this until the heap is reestablished.



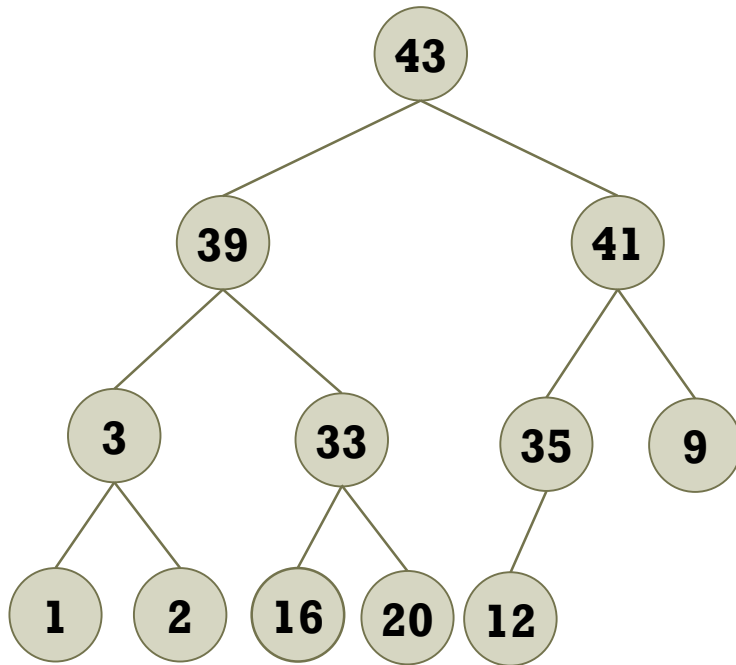
Deletion in Heap

- Step 2: Updating the heap
 - Compare the root with its children.
 - If it is smaller than any of children, exchange the position with its max child.
 - Do this until the heap is reestablished.



Exercise: Deletion in Heap

- Deleting 43 and 41 from the heap



- What is the time complexity for deletions?

Implementation of Deletion

```
Data Delete(Heap *pheap)
{
    Data max = pheap->items[1].data;
    HNode last = pheap->items[pheap->num];
    int parent = 1, child;
    // Compare the root with its child nodes.
    while (child = GetHighPriorityChild(pheap, parent)) {
        if (last.priority < pheap->items[child].priority) {
            pheap->items[parent] = pheap->items[child];
            parent = child;
        }
        else break;
    }

    pheap->items[parent] = last;
    pheap->num--;

    return max;
}
```

Utility Functions for Deletions

- Given an index, find a parent or a child index.

```
// Get a parent index for a given index.  
int GetParent(int idx)  
{  
    return idx / 2;  
}  
  
// Get a left child index for a given index.  
int GetLChild(int idx)  
{  
    return idx * 2;  
}  
  
// Get a right child index for a given index.  
int GetRChild(int idx)  
{  
    return idx * 2 + 1;  
}
```

Utility Functions for Deletions

- Given an index, find a child node with the highest priority among its child nodes.

```
int GetHighPriorityChild(Heap* pheap, int idx)
{
    if (GetLChild(idx) > pheap->num) // No child nodes exist.
        return 0;
    else if (GetLChild(idx) == pheap->num) // Exist a left child only.
        return GetLChild(idx);
    else // Choose a child node with the highest priority.
    {
        int left = GetLChild(idx), right = GetRChild(idx);
        if (pheap->items[left].priority > pheap->items[right].priority)
            return left;
        else
            return right;
    }
}
```


Implementing Priority Queue

- Heap-based implementation
 - The priority queue is simply implemented by using a heap.

```
#include "Heap.h"

typedef Heap PQueue;

void InitPQueue(PQueue* ppqueue);
bool IsPQEmpty(PQueue * ppqueue);
bool IsPQFull(PQueue* ppqueue);

// Insert an item to the priority queue.
void Enqueue(PQueue * ppqueue, Data data, int priority);
// Delete an item to the priority queue.
Data Dequeue(PQueue * ppqueue);
```

Implementing Priority Queue

■ Operations for priority queue

```
void InitPQueue(PQueue* ppqueue) {
    InitHeap(ppqueue);
}

bool IsPQEmpty(PQueue * ppqueue) {
    return IsEmpty(ppqueue);
}

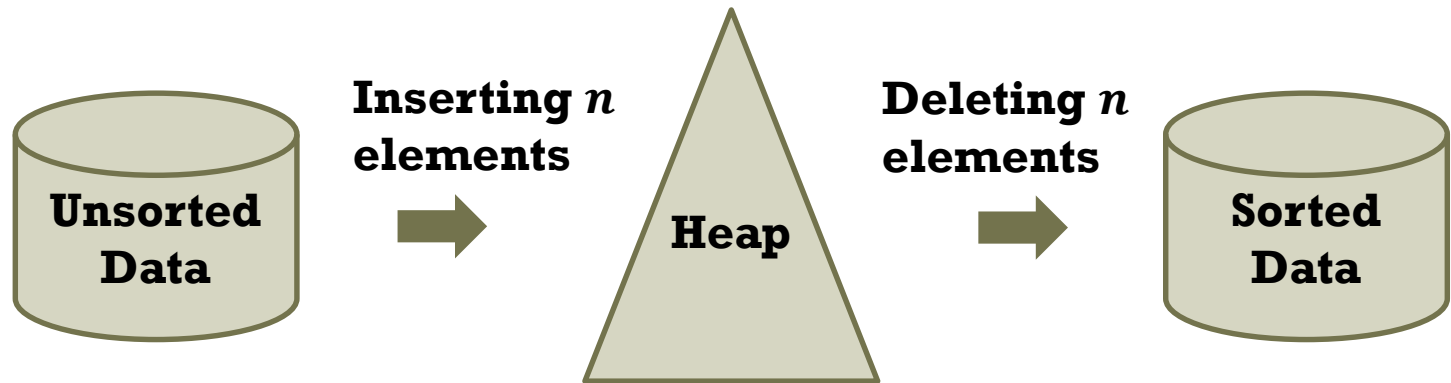
bool IsPQFull(PQueue* ppqueue) {
    return IsFull(ppqueue);
}

void Enqueue(PQueue * ppqueue, Data data, int priority) {
    Insert(ppqueue, data, priority);
}

Data Dequeue(PQueue * ppqueue) {
    return Delete(ppqueue);
}
```

Heap Sort

- Sorting elements by using a max heap
 - **Step 1:** Inserting each element to the heap
 - **Step 2:** Removing all elements from the heap in sequence
 - When deleting elements, it ensures to return the maximum element.



Implementing Heap Sort

- What is the time complexity of heap sort?

```
void HeapSort(Data a[], int n)
{
    Heap heap;
    InitHeap(&heap);

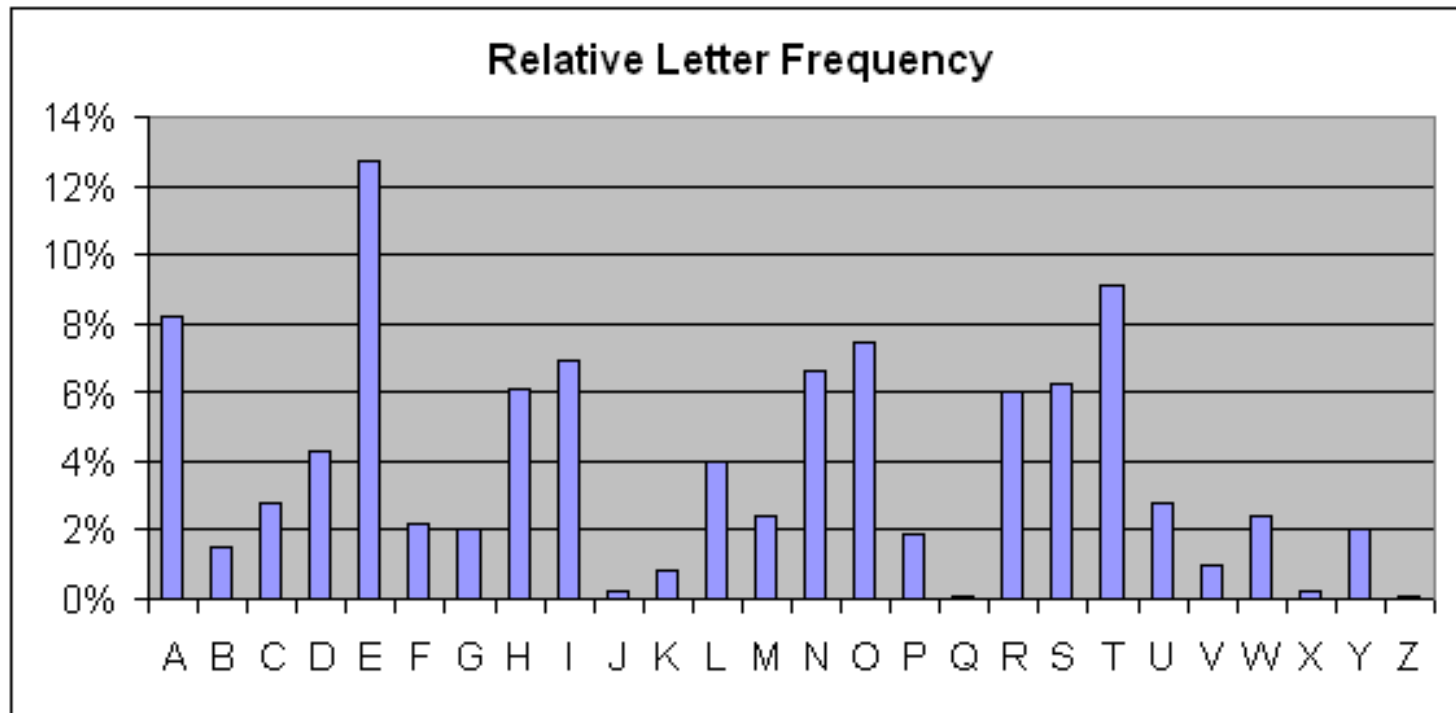
    // Insert all elements to the heap.
    for (int i = 0; i < n; i++)
        Insert(&heap, a[i], a[i]);

    // Remove all elements from the heap.
    for (int i = n - 1; i >= 0; i--)
        a[i] = Delete(&heap);
}
```

- How to find k largest elements?

Huffman Coding

- Motivation of Huffman coding
 - **Not all characters occur** with the same frequency.
 - It is inefficient that all characters are allocated to the same amount of space!



Huffman Coding

- Huffman coding is a form of statistical coding
 - The lengths of characters can vary.
 - The coding will be shorter for the more frequently used characters.

Character	Frequency	Bit	# of bits
E	29	0	29
A	11	10	22
D	5	110	15
C	3	1110	12
B	2	1111	8

- https://en.wikipedia.org/wiki/Huffman_coding

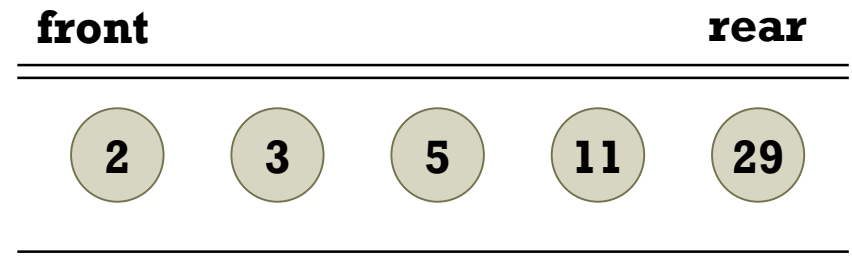
How to Implement Huffman Coding

- Overall procedure
 - **Step 1:** Scan text to be compressed and tally occurrence of all characters.
 - **Step 2:** Sort characters based on the number of occurrences in text.
 - **Step 3: Build a Huffman code tree based on prioritized list.**
 - **Step 4: Perform a traversal of tree to determine all codes.**
 - **Step 5:** Scan text again and create a new file using the Huffman codes.

Building a Huffman Code Tree

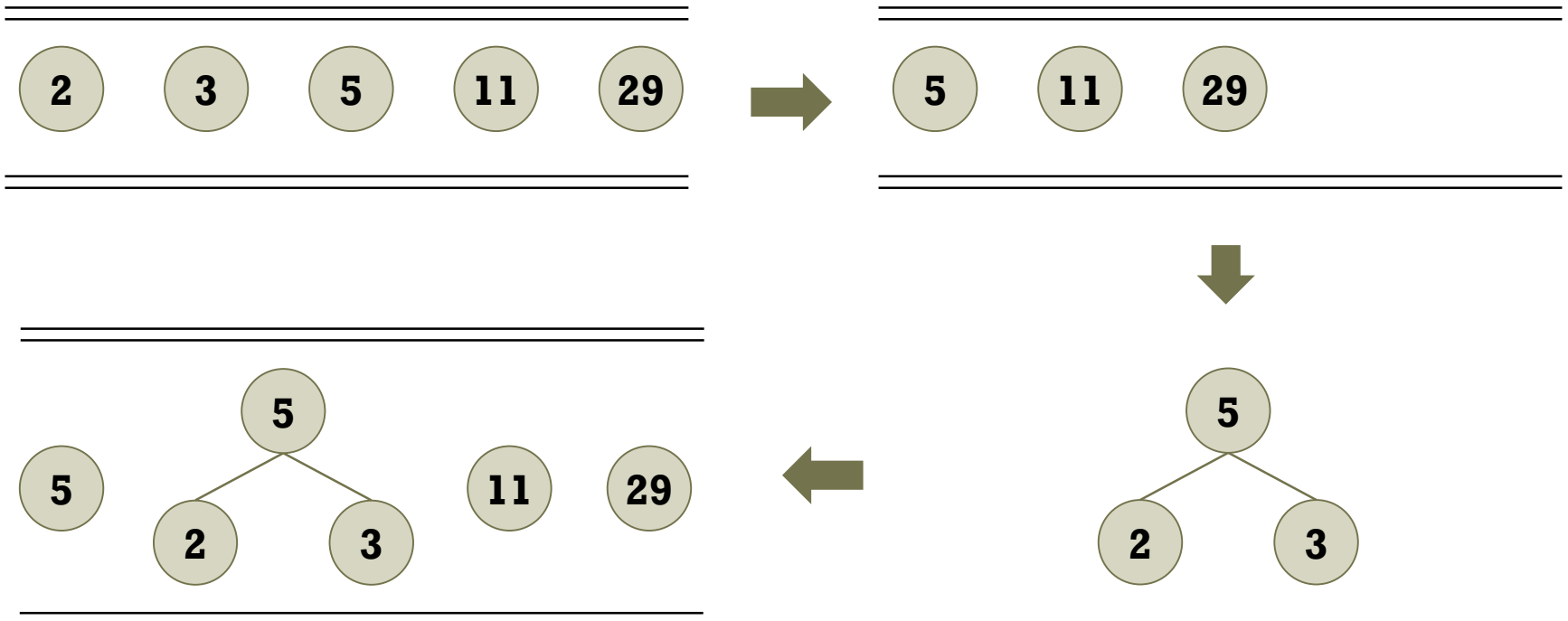
- How to build Huffman code based on the list?
 - Place nodes in a priority queue.
 - The lower the occurrence, the higher the priority in the queue.
- Step 3-1: Insert all nodes to the priority queue using the min heap.

Character	Frequency
B	2
C	3
D	5
A	11
E	29



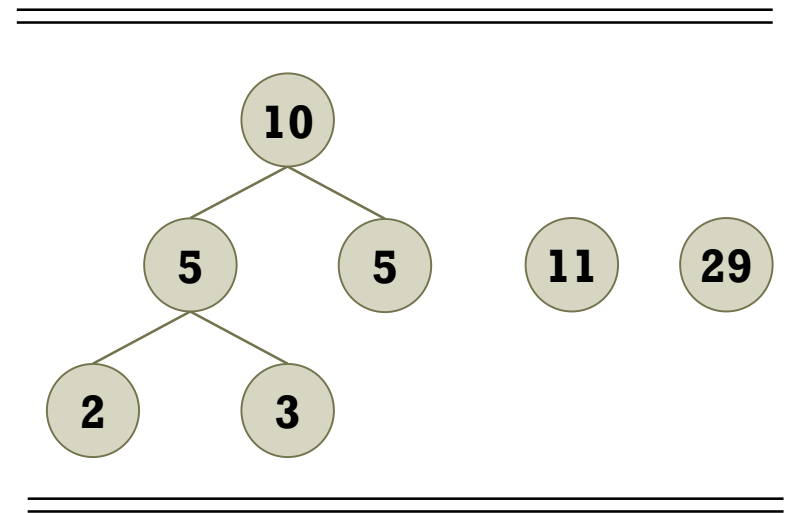
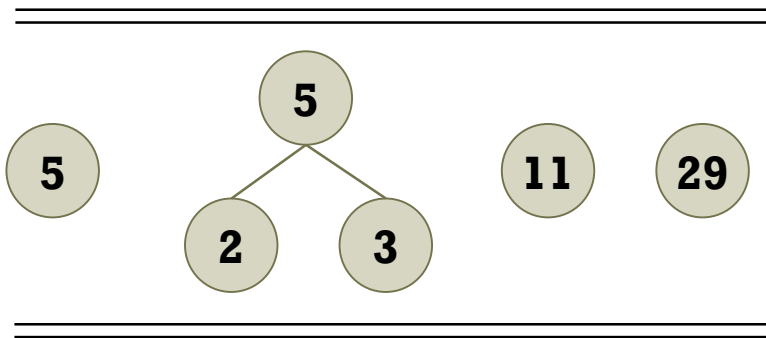
Building a Huffman Code Tree

- Step 3-2: Updating the priority queue
 - Dequeue two nodes from the priority queue.
 - Make a tree based on the frequency.
 - Enqueue the tree to the priority queue.



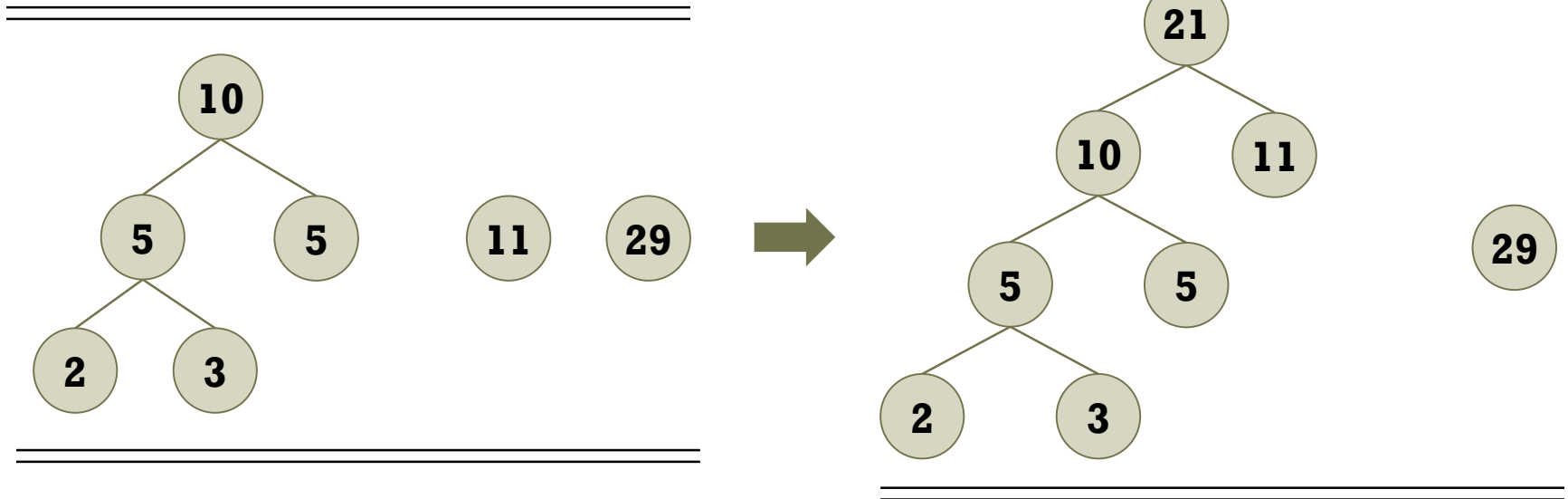
Building a Huffman Code Tree

- Step 3-2: Updating the queue until it is empty
 - When making a tree, the node with a higher level becomes a left-child node.



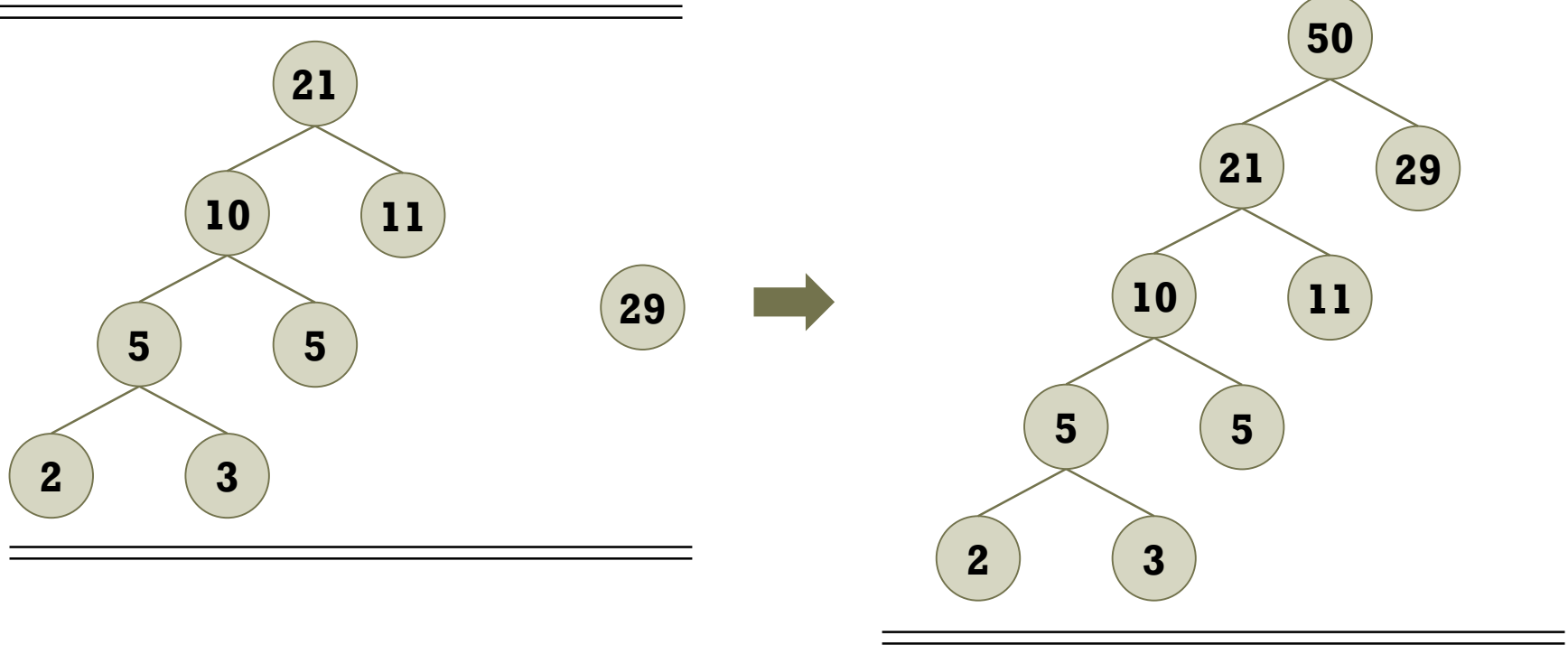
Building a Huffman Code Tree

- Step 3-2: Updating the queue until it is empty



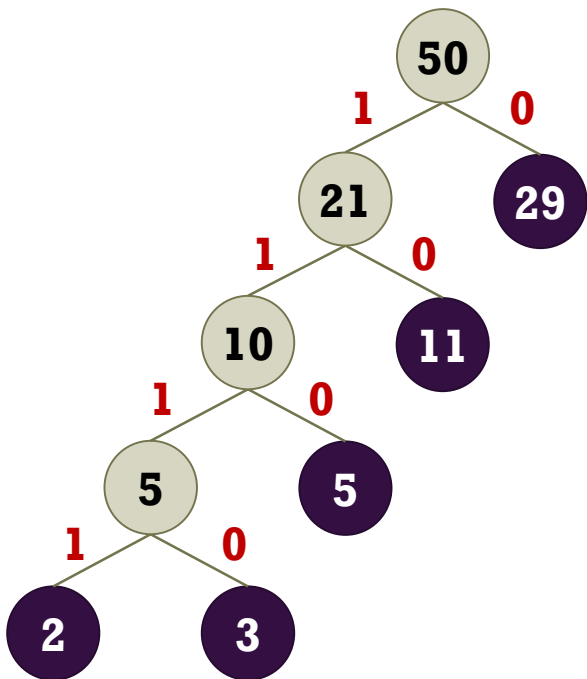
Building a Huffman Code Tree

- Step 3-2: Updating the queue until it is empty



Determining Bit Codes

- Step 4: Perform a traversal to determine all codes.
 - **Left-child node: 1, right-child node: 0**
 - Make a bit code for the path from the root to the leaf node.

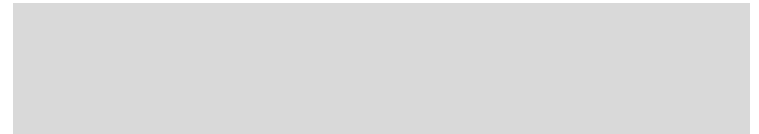


Character	Frequency	Bits
E	29	
A	11	
D	5	
C	3	
B	2	

Transforming Text into Bit Codes

- Step 5: Scan text again and create a new file using the Huffman codes.

EEEADACB



**Transform an alphabet
into its corresponding bit
code.**

Character	Frequency	Bit coding
E	29	0
A	11	10
D	5	110
C	3	1110
B	2	1111

Example of Huffman Coding

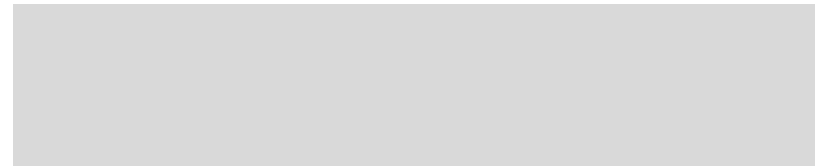
- How do we make a Huffman coding tree?

Character	Frequency
T	28
U	7
W	6
Y	5
V	4



- How do we encode the following text?

TUTWTYTV



Example of Huffman Coding

- How do we make a Huffman coding tree?

Character	Frequency
I	15
H	14
F	12
G	5
K	4



- How do we encode the following text?

KFIHHIFG

