Data Structures and Algorithms
(資料結構與演算法)

Lecture 10: Heap

Hsuan-Tien Lin (林軒田)
htlin@csie.ntu.edu.tw

Department of Computer Science
& Information Engineering

National Taiwan University
(國立台灣大學資訊工程系)

motivation

Visual Intuition of Priority Queue

figure by NewbieRunner, licensed under CC BY-NC-ND 2.0 via Flickr

max-priority-out

- priority boarding
- bandwidth management
- job scheduler

priority queue: ‘extension’ of queue for more realistic use

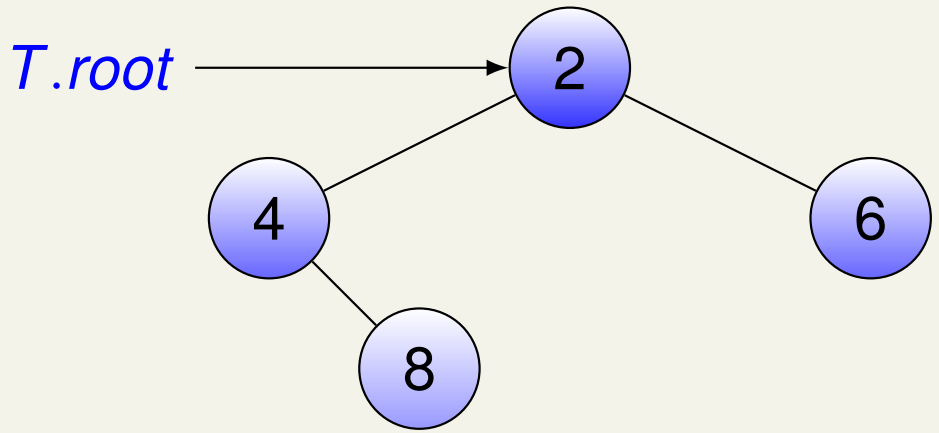
Priority Queue with Binary Tree

previously

data	
left	*right*

next

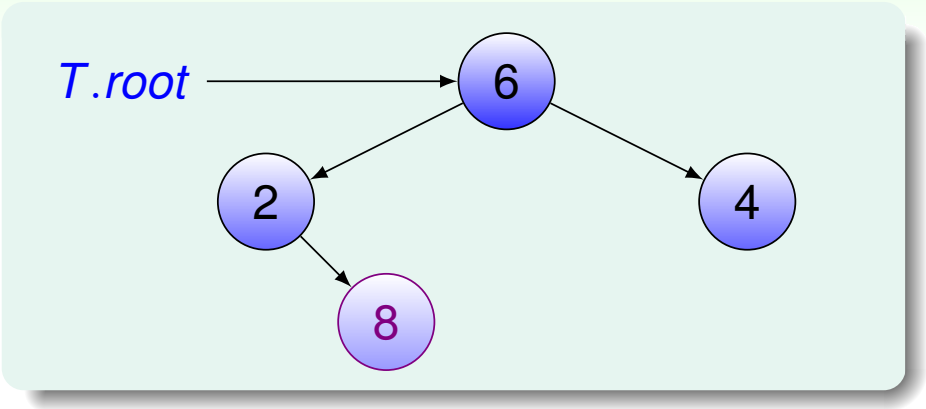
key	*data*
left	*right*

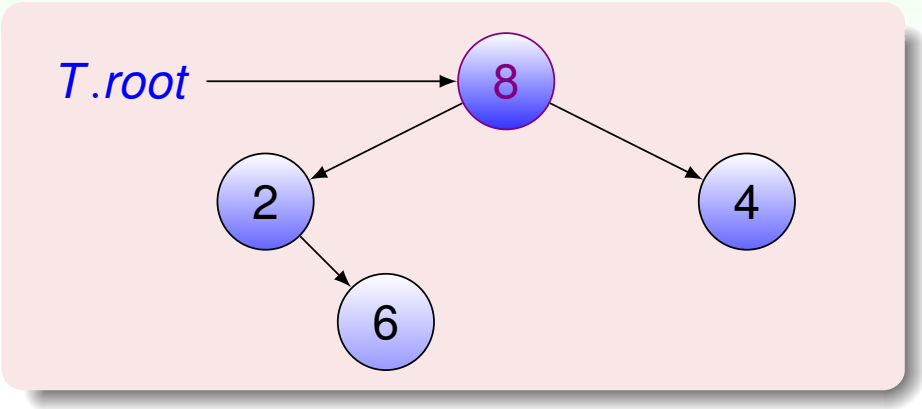


- *key*: priority (the larger the better)
- *data*: item in todo list
  —will show *key* only for simplicity

goal: get the node with largest key fast (& remove it)

Max-Root Tree

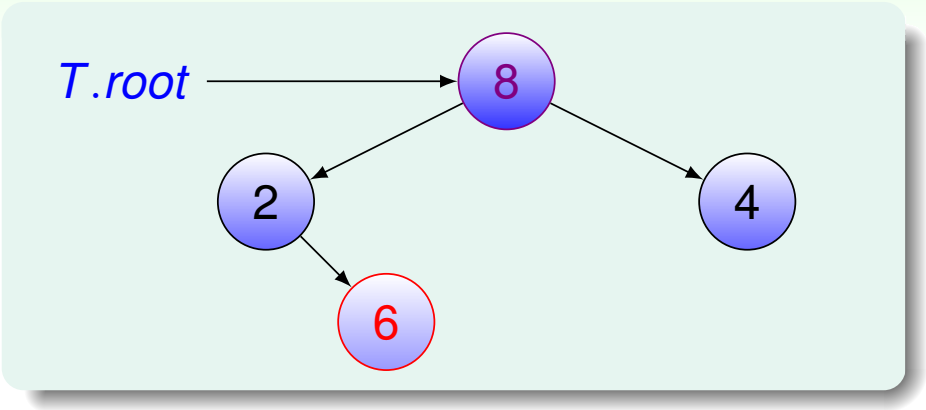


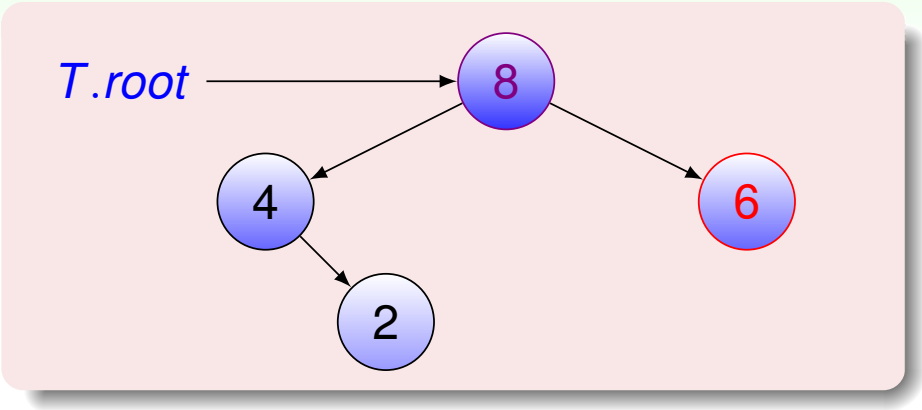


faster access to largest key if put at *T.root*

max-root (binary) tree: largest key @ root

Max Tree

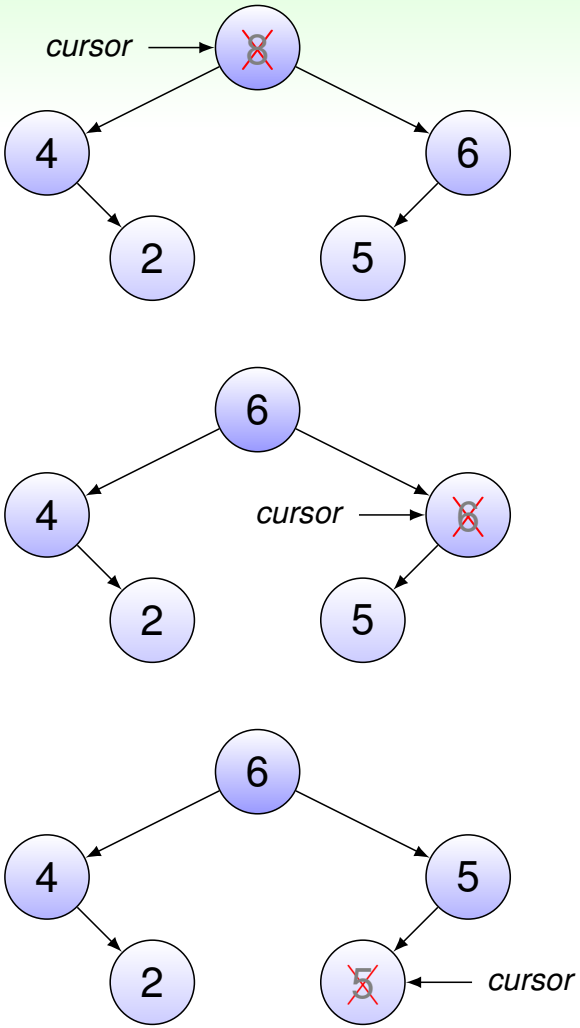




hard to remove largest fast if cannot easily locate second largest

max (binary) tree: largest key @ root
for every subtree

Simple Removal for Max Tree



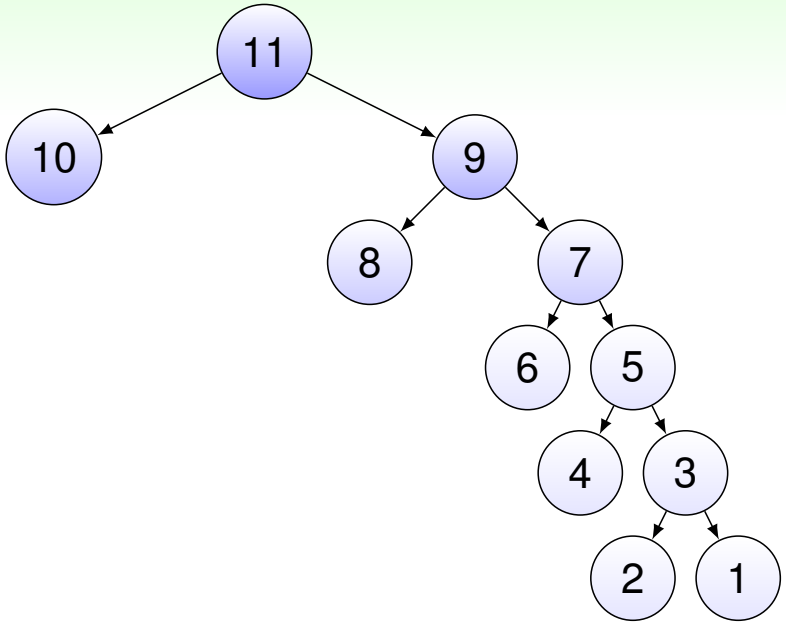
REMOVE-LARGEST(max tree T)

```
1  result = T.data
2  cursor = T.root
3  while [cursor not NIL]
4        larger =
                 COMPARE(cursor.left, cursor.right)
5        if [larger not NIL]
6              cursor.key = larger.key
7              cursor.data = larger.data
8        cursor = larger
9  return result
```

REMOVE-LARGEST for height-h max-tree takes $O(h)$ time

max heap

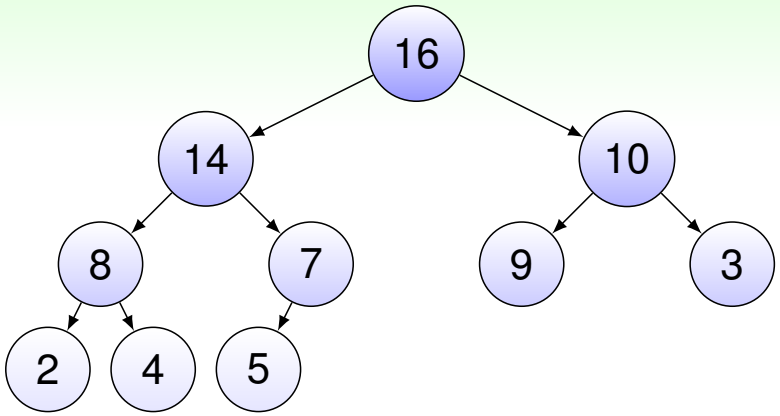
Worst Case of REMOVE-LARGEST for Max Tree




```
REMOVE-LARGEST(max tree T)
1  result = T.data
2  cursor = T.root
3  while [cursor not NIL]
4        larger =
                 COMPARE(cursor.left, cursor.right)
5        if [larger not NIL]
6              cursor.key = larger.key
7              cursor.data = larger.data
8        cursor = larger
9  return result
```

$h = O(n)$ and hence REMOVE-LARGEST is $O(n)$:-(

Max Heap: "Shortest" Max Tree



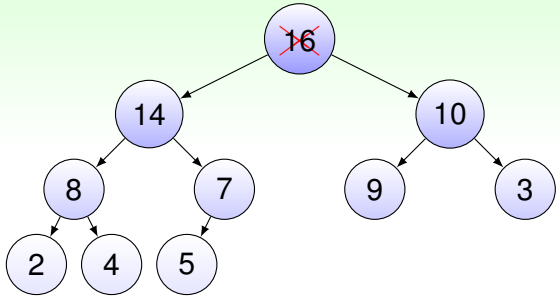
max heap = max tree
+ complete binary tree

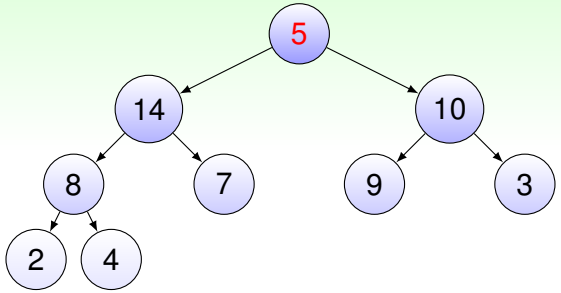
REMOVE-LARGEST(max tree T)

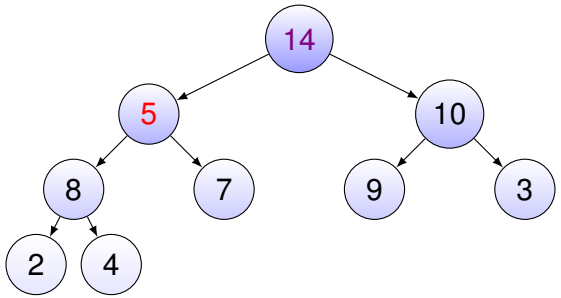
```
1  result = T.data
2  cursor = T.root
3  while [cursor not NIL]
4        larger =
                 COMPARE(cursor.left, cursor.right)
5        if [larger not NIL]
6              cursor.key = larger.key
7              cursor.data = larger.data
8        cursor = larger
9  return result
```

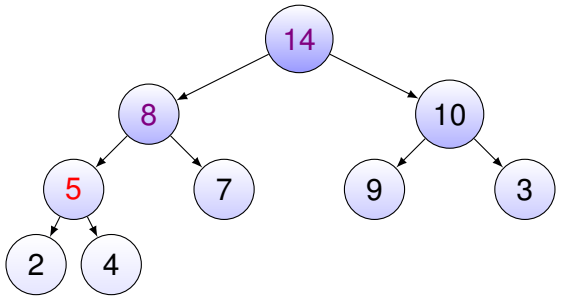
but REMOVE-LARGEST cannot
preserve **complete binary tree** easily

REMOVE-LARGEST for Max Heap









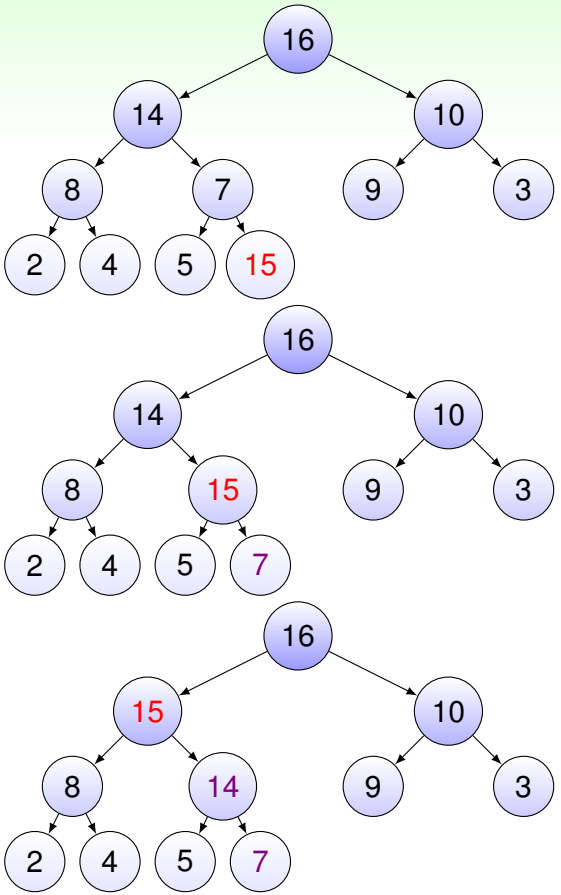
REMOVE-LARGEST(max heap T)

```
 1  result = T.data
 2  remove T.tail & copy it to T.root
 3  cursor = T.root
 4  repeat
 5        largest =
 6           LARGEST(cursor, cursor.left, cursor.right)
 7        if [largest equals cursor ]
 8              break the loop
 9        else
10              SWAP(cursor, largest)
11              cursor = largest
12  until cursor = NIL
13  return result
```

lines 3-12: HEAPIFY; can similarly DECREASE-KEY

INSERT for Max Heap

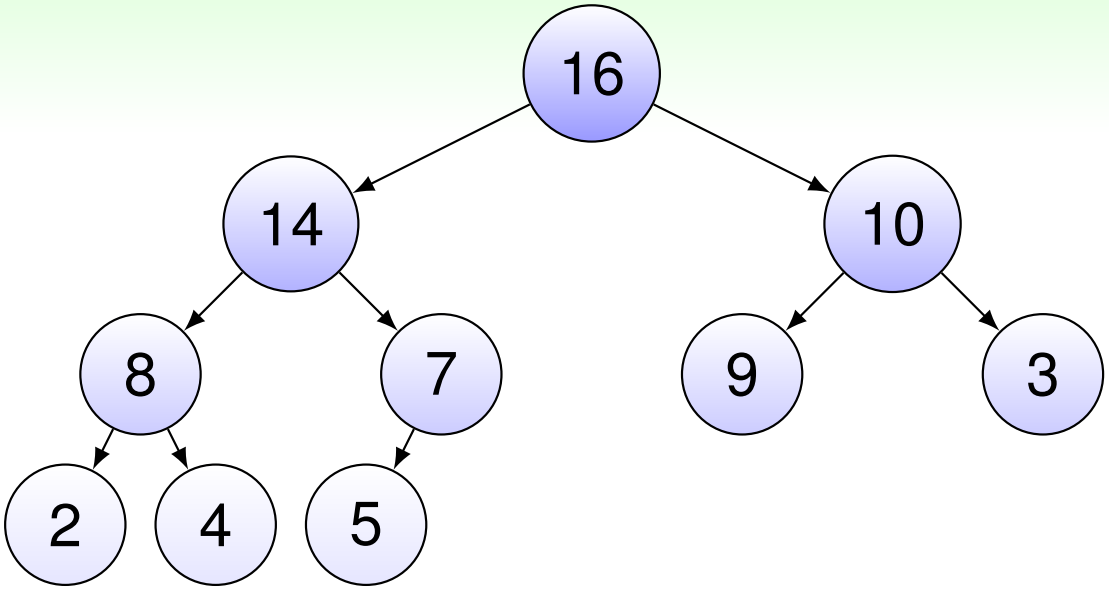
```
INSERT(max heap T, key, data)
 1  insert (key, data) to T.tail
 2  cursor = T.tail
 3  repeat
 4       larger =
 5            LARGER(cursor, cursor.parent)
 6       if [larger equals cursor ]
 7            break the loop
 8       else
 9            SWAP(cursor, larger)
10            cursor = larger
11  until cursor = NIL
```



can similarly INCREASE-KEY

heap sort

Heap $\Longleftrightarrow$ Partially Ordered Array



index	1	2	3	4	5	6	7	8	9	10
key	16	14	10	8	7	9	3	2	4	5

unordered array $\rightarrow$ heap
selection sort $\rightarrow$ heap sort

Selection Sort versus Heap Sort

```
SELECTION-SORT(A)
1  for i = 1 to A.length
2        m = GET-MIN-INDEX(A, i, A.length))
3        SWAP(A[i], A[m])
4  return A // which has been sorted in place
```

time: $O(n) \cdot O(n)$

without any preprocessing

```
HEAP-SORT(A)
1  for i = A.length downto 1
2        (key, data)
           = REMOVE-LARGEST(A, 1, A.length))
3        copy (key, data) to A[i]
         // original A[i] is already in A[1]
4  return A // which has been sorted in place
```

time: $O(n) \cdot O(\lg n)$

after building heap

heap sort: faster selection sort algorithm
with help of heap data structure

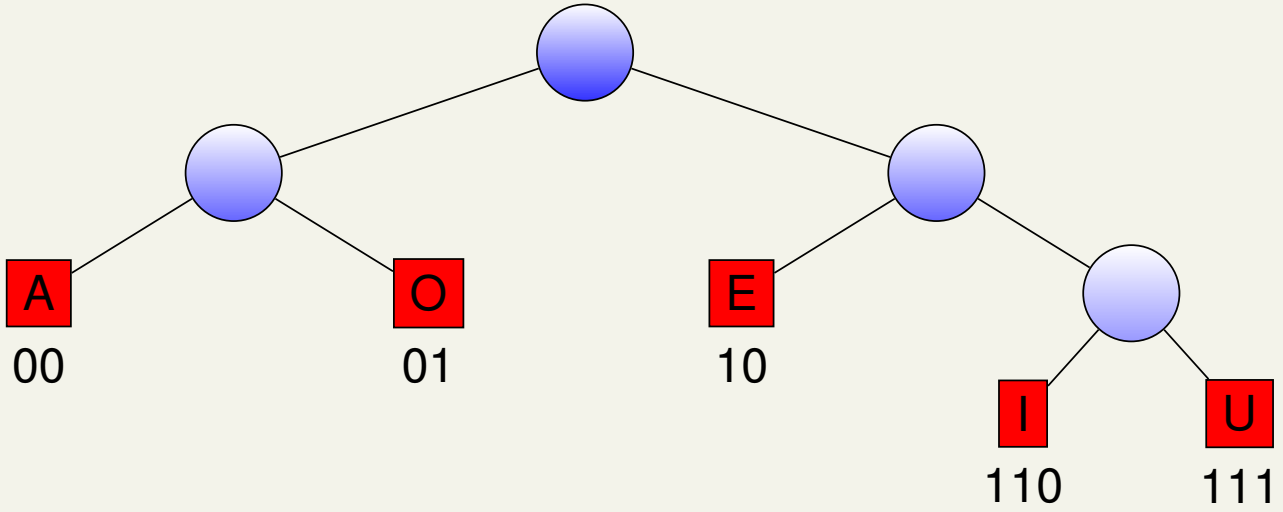
Missing Piece: Building Heap from Unordered Array

check textbook for details:

- suffices to heapify first half of all elements;
- total time (careful calculation) is $O(n)$

application: huffman tree building

Huffman Tree



Huffman Tree

a special binary tree where

- each leaf node stores a symbol (i.e. ‘A’, ‘E’)
- path to leaf node $\equiv$ code of symbol
- symbol string IEEEAI $\Leftrightarrow$ code stream 110101010000110
- compress symbol string under some assumptions

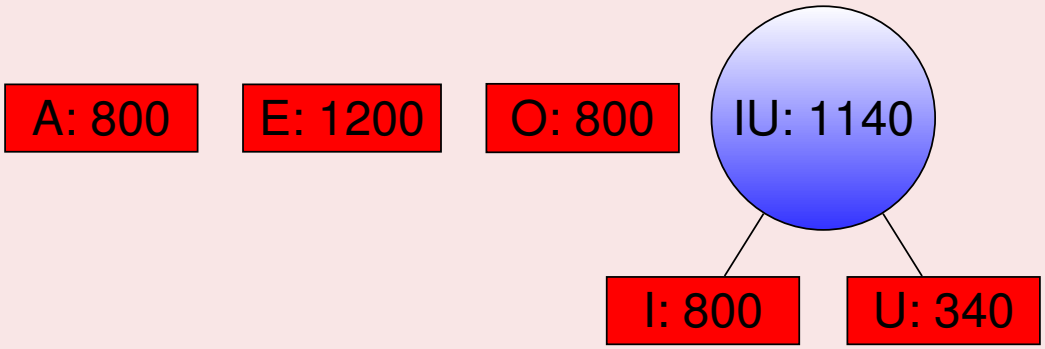
important behind lossless compression (zip, inside jpeg)

Huffman Tree Construction (1/2)

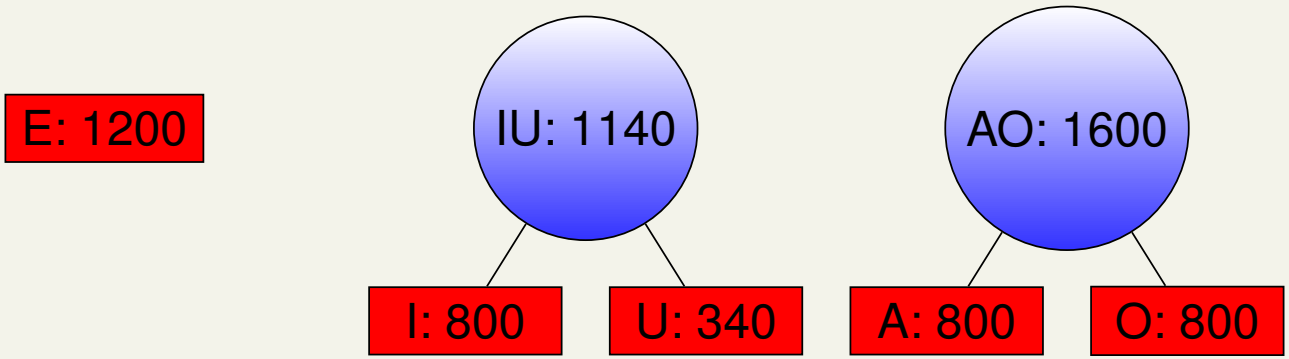
compute frequency of each symbol

A: 800 E: 1200 I: 800 O: 800 U: 340

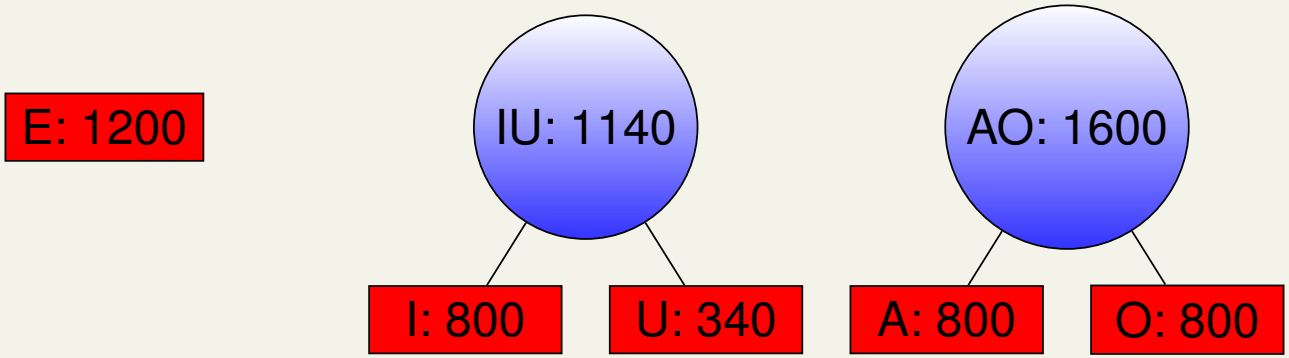
remove two smallest-frequency nodes; build sub-tree & insert



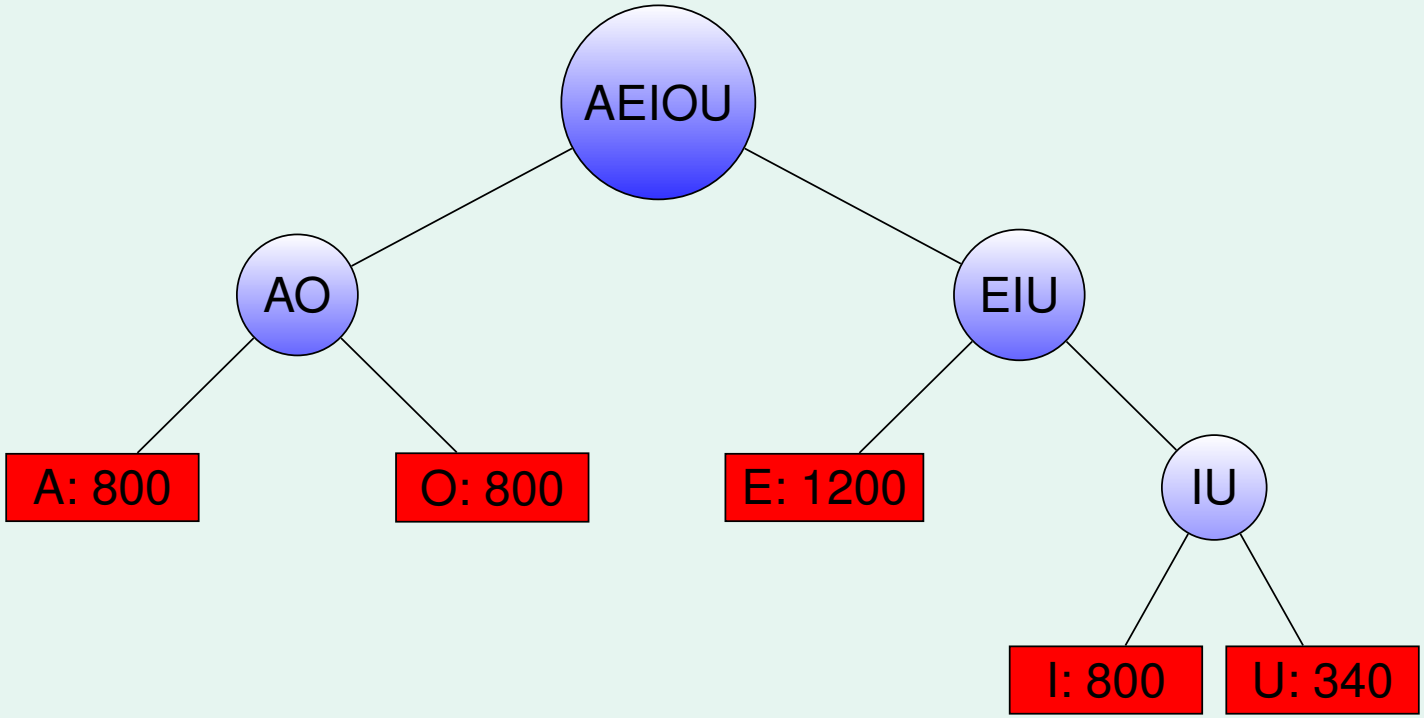
repeat the step



Huffman Tree Construction (2/2)



repeat two more steps and done!



Take-Home Message

DSA is useful

- binary tree is useful
  —huffman tree!
- heap is useful
  —'remove two smallest-freq' when constructing huffman tree

and connects to other classes (e.g. Information Theory)

Summary

Lecture 10: Heap

- motivation

  possibly efficient priority queue by max tree
- max heap

max + complete bin. tree for $O(\lg n)$ removal/insertion

- heap sort

  max heap in array to speed up selection sort
- application: huffman tree building

priority queue to build huffman tree for compression