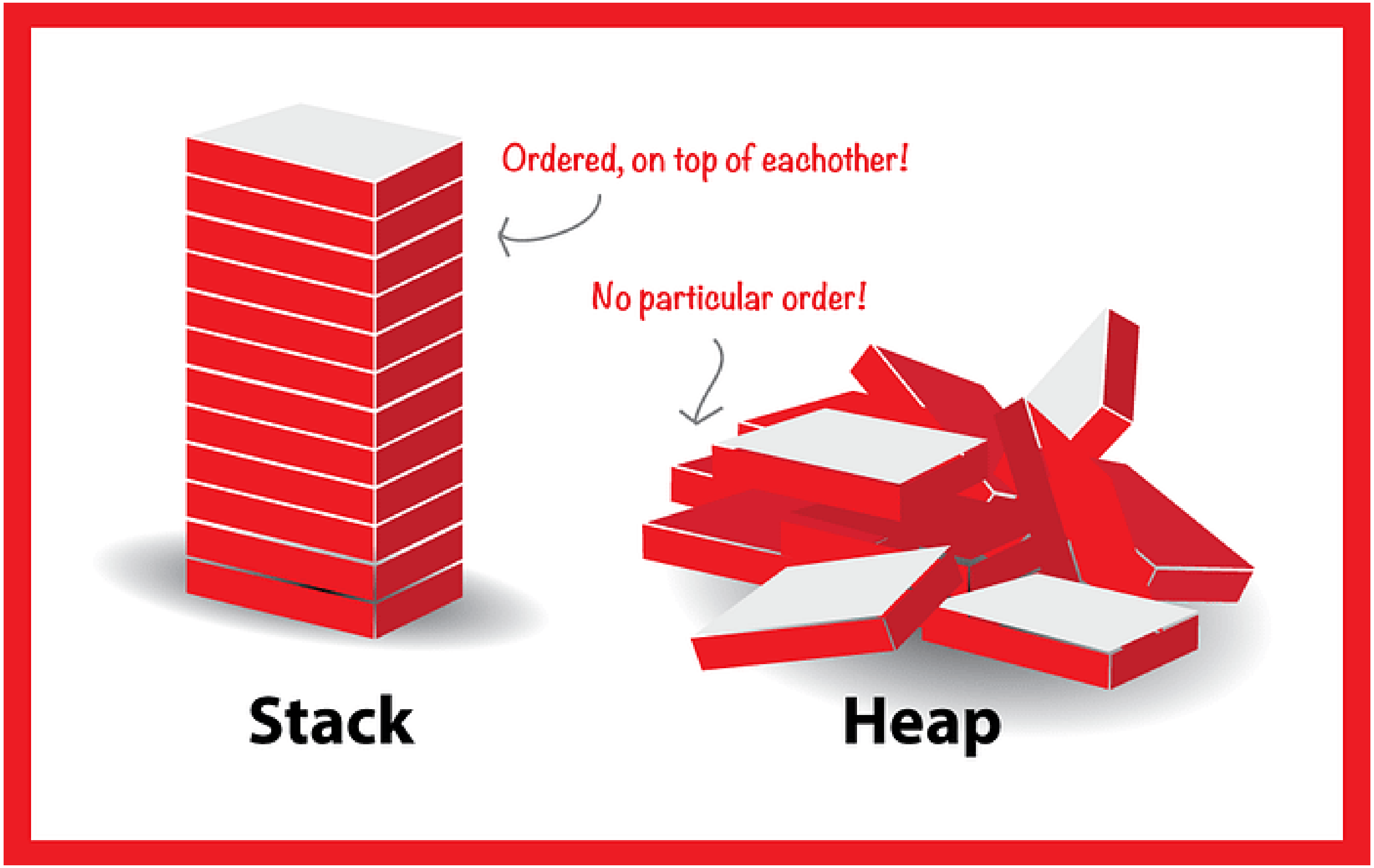
Lecture 4
Heap and Heap-Sort

Subject Lecturer: Kevin K.F. YUEN, PhD.

Acknowledgement: Slides were offered from Prof. Ken Yiu.
Some parts have been revised and indicated.

Outline

- What is a *heap*?
- How do we maintain the *heap property*?
- How do we use a heap to *sort an array*?
- How do we *insert*, *delete*, *update* items in a heap?
- What are the variants of heaps?



Source: google photo

Heap: Applications

- Heap (or called Max-Heap)
  - A **data structure** that supports fast retrieval of the **maximum** value
  - Also called a *priority queue*, used for managing a set of items based on their "priority"

- Applications
  - Task/resource management (based on priority)
  - Sorting
  - Selection (e.g., finding the *k*-th largest value in an array)
  - Graph problems

Heap

Array Implementation

Instance variables	*Meaning*
A	An array of items
length	Length of array A
size	Actual size of the heap

Note that n=*size*

Operation	*Complexity*	*Meaning*
Max-Heapify(A, i)	O(log n)	Maintain the heap property
Build-Max-Heap(A)	O(n)	Build a max-heap from array
Get-Max(A)	O(1)	Get the maximum item
Extract-Max(A)	O(log n)	Remove the maximum item
Update-Key(A, i, k)	O(log n)	Update at position i by item k
Insert-Key(A, k)	O(log n)	Insert an item k

Most of the operations use "Max-Heapify"

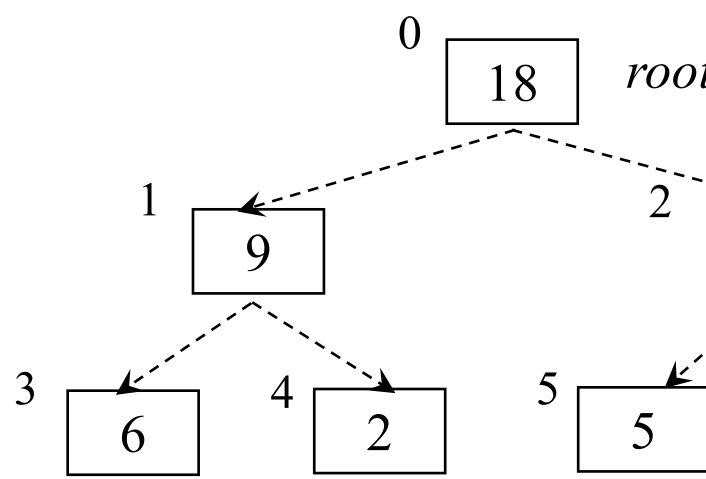
Heap Structure

- It is stored as an **array** $A[0..length-1]$
  - No need to store any pointer ☺
  - Length: the length of array A
  - Size: the actual number of items

0	1	2	3	4	5	6
18	9	12	6	2	5	/

Length = 7, (Heap) Size = 6

- It can be viewed as a *hidden* binary tree
  - Nodes are filled from top to bottom
  - At the same level, nodes are filled from left to right



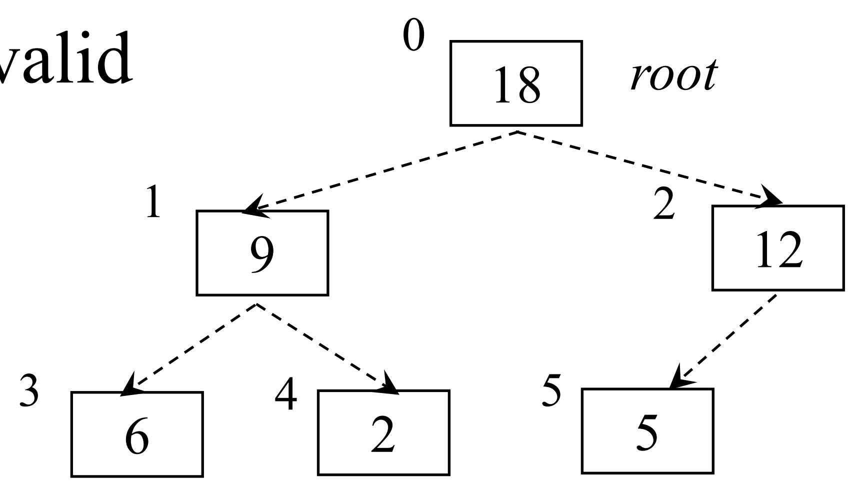
- Given node i, we can find:
  - Parent(i) $= \lfloor (i-1)/2 \rfloor$
  - Left(i) $= 2i + 1$
  - Right(i) $= 2i + 2$

[Exercise] find the parent, left child, and right child of node i=2

- Substitute i=2 in the following functions
  - $\text{Parent}(i) = \lfloor (i-1)/2 \rfloor$
  - $\text{Left}(i) = 2i + 1$
  - $\text{Right}(i) = 2i + 2$

0	1	2	3	4	5	6
18	9	12	6	2	5	/

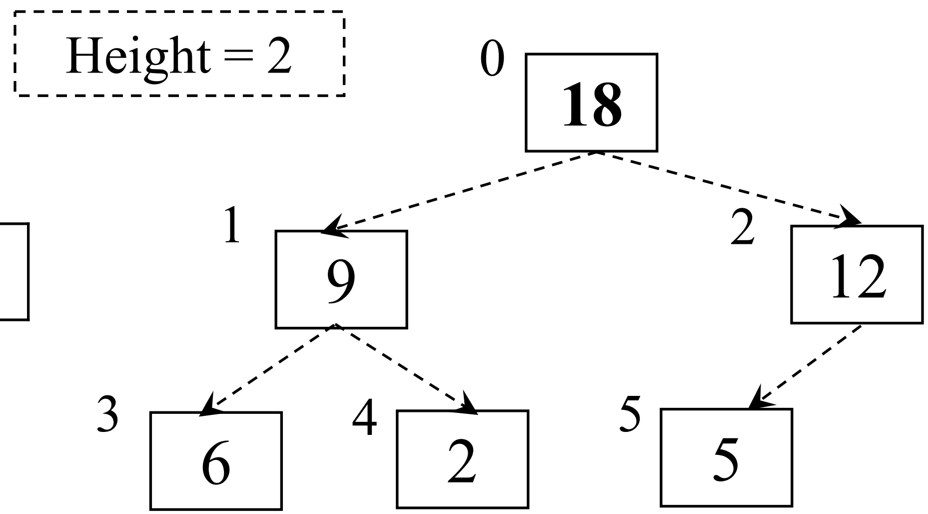
- Which of these positions are valid (i.e., less than *size*=6)?



Heap Property

- **Heap property**: $A[\text{Parent}(i)] \geq A[i]$
  - The root (i=0) stores the largest item
- Height $h = \lfloor \log_2 n \rfloor$, the largest distance of the path from the root to any leaf node
  - where n is number of items in the heap

0	1	2	3	4	5
18	9	12	6	2	5



[Exercise] Check the heap property for each array

"Array A"

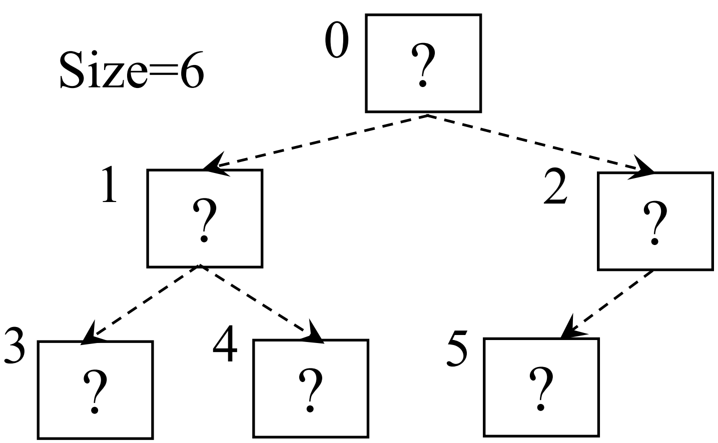
0	1	2	3	4	5
12	77	92	63	39	54

"Array B"

0	1	2	3	4	5
92	77	12	63	39	54

"Array C"

0	1	2	3	4	5
92	77	54	63	39	12



Heap: comparison for other types of objects

- In the previous examples, integers are stored in a heap
- Can other types of objects (e.g., strings) be stored in a heap?
  - Yes, provided that we can define the **comparator** (i.e., comparison rule) between two objects
- Example application: task management by the task priority
  - Each *task object* has two attributes: taskID, taskPriority
  - Define the comparator for two tasks *x* and *y* as:
    - `-1    if ( x.taskPriority < y.taskPriority )`
    - `0     if ( x.taskPriority = y.taskPriority )`
    - `1     if ( x.taskPriority > y.taskPriority )`

0	1	2	3	4	5
(A,18)	(B,9)	(C,12)	(D,6)	(E,2)	(F,5)

Java: Priority Queue<E>

- PriorityQueue<E>
  - <E> is the type of item stored
  - We need a “Comparator” to compare two (type-<E>) objects
  - It behaves as a Min-heap, but not Max-heap
- Reference:
- https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/PriorityQueue.html

Operation	***Meaning***
`boolean add(E e)`	Inserts the specified item into this priority queue
`E peek()`	Retrieves, but does not remove, the head of this priority queue
`E poll()`	Removes the head of this priority queue
`int size()`	Returns the number of items
`void clear()`	Removes all items

Code example: a priority queue for strings by increasing length

```java
import java.util.*;

class CompareSTR implements Comparator<String> {
    public int compare(String s1, String s2) {
        if (s1.length() < s2.length())
            return -1;
        else if(s1.length() > s2.length())
            return 1;
        else
            return 0;
    }
}

public class Test {
    public static void main(String[] args) {
        PriorityQueue<String> pq = new PriorityQueue<String>(new CompareSTR());
        pq.add("!!!!!!!!!!!!!!!!!!!!!!");
        pq.add("@@@@@@@@@@@@@");
        pq.add("##################");
        pq.add("$$$$$$$$$$");
        pq.add("%%%%%%%%%%%%%%%%");
        while (!pq.isEmpty())
            System.out.println(pq.poll());
    }
}
```

```
$$$$$$$$$$
@@@@@@@@@@@@@
%%%%%%%%%%%%%%%%
##################
!!!!!!!!!!!!!!!!!!!!!!
```

Outline

- What is a *heap*?
- How do we maintain the *heap property*?
- How do we use a heap to *sort an array*?
- How do we *insert*, *delete*, *update* items in a heap?
- What are the variants of heaps?

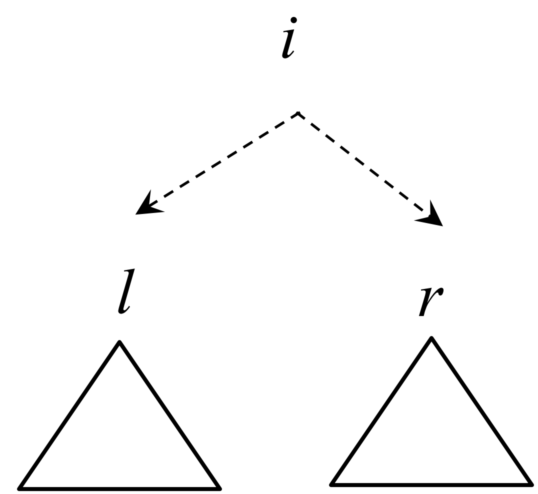
Maintaining the Heap Property

Pre-condition: before calling Max-Heapify, we require that "the trees rooted at Left(i) and Right(i) are max-heaps"

Max-Heapify (A, i)

$l \leftarrow$ Left(i)
$r \leftarrow$ Right(i)
$largest \leftarrow i$
if $l < A.size$ and $A[l] > A[largest]$
    $largest \leftarrow l$
if $r < A.size$ and $A[r] > A[largest]$
    $largest \leftarrow r$
if $largest \neq i$
    swap $A[i]$ with $A[largest]$
    Max-Heapify(A, $largest$)

Idea: pass a small item from the top to the bottom (by using recursion)

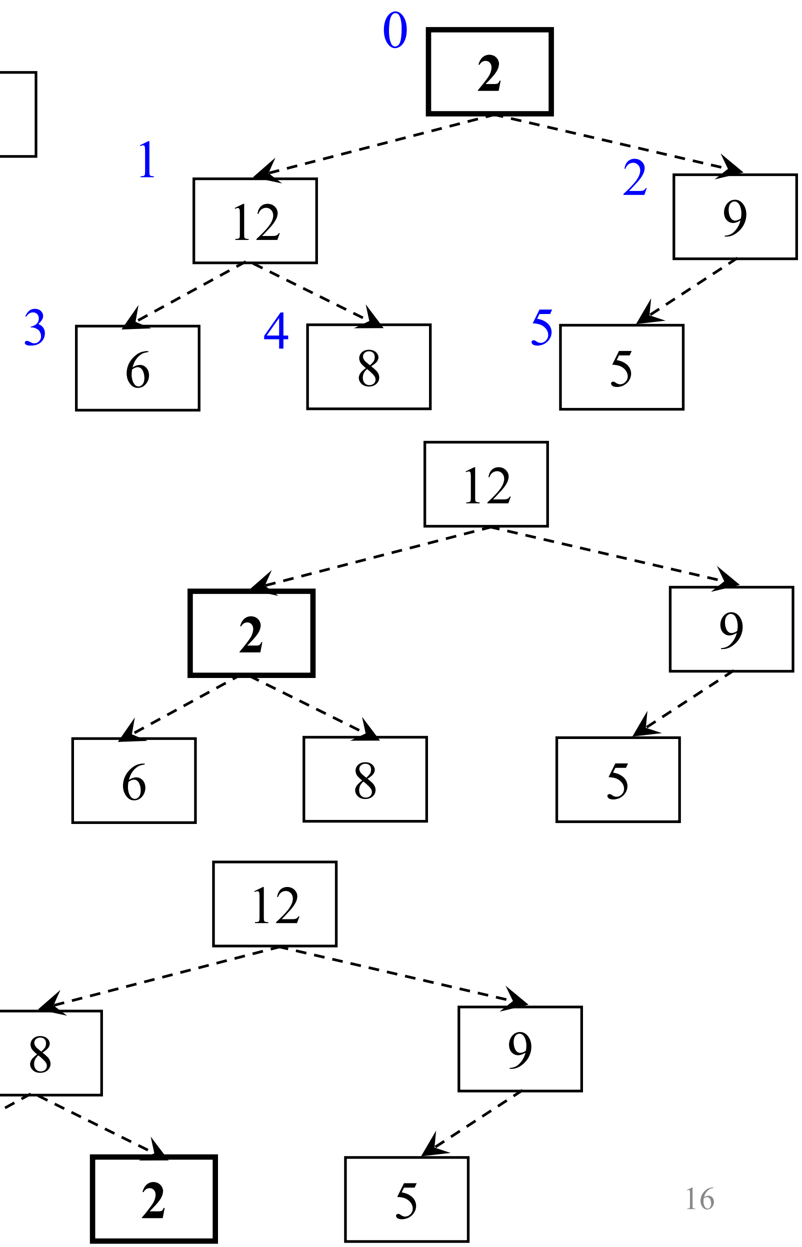


Maintaining the Heap Property

Example: Max-Heapify(A, 0)

Max-Heapify (A, i)

1. $l \leftarrow$ Left(i)
2. $r \leftarrow$ Right(i)
3. $largest \leftarrow i$
4. if $l < A.size$ and $A[l] > A[largest]$
5. $\quad largest \leftarrow l$
6. if $r < A.size$ and $A[r] > A[largest]$
7. $\quad largest \leftarrow r$
8. if $largest \neq i$
9. $\quad$ swap $A[i]$ with $A[largest]$
10. $\quad$ Max-Heapify(A, $largest$)



Max-Heapify: Running Time

Max-Heapify (A, i)

1. $l \leftarrow$ Left(i)
2. $r \leftarrow$ Right(i)
3. $largest \leftarrow i$
4. if $l < A.size$ and $A[l] > A[largest]$
5. $\quad largest \leftarrow l$
6. if $r < A.size$ and $A[r] > A[largest]$
7. $\quad largest \leftarrow r$
8. if $largest \neq i$
9. $\quad$ swap $A[i]$ with $A[largest]$
10. $\quad$ Max-Heapify(A, $largest$)

- $T(n) = O(\log n)$
  - where n is $A.size$
- We skip the detailed analysis
  - beyond the scope of this course

Outline

- What is a *heap*?
- How do we maintain the *heap property*?
- How do we use a heap to *sort an array*?
- How do we *insert*, *delete*, *update* items in a heap?
- What are the variants of heaps?

Heap Sort

- Sorting problem
  - *Input*: an array of n items
  - *Output*: an array of n items in ascending order
- The idea of *heap sort*
  - 1. Build a heap from an array
  - 2. Repeatedly move the largest item in the heap to the end of the array
- How do we implement these steps?

Building a Heap from Array

Is this method is correct?

Any wasted work in this method?

~~Build-1 (A)~~

~~1. $n \leftarrow A.length$~~

~~2. for $i \leftarrow 0$ to $n-1$~~

~~3. Max-Heapify (A, i)~~

~~Build-2 (A)~~

~~1. $n \leftarrow A.length$~~

~~2. for $i \leftarrow n-1$ downto 0~~

~~3. Max-Heapify (A, i)~~

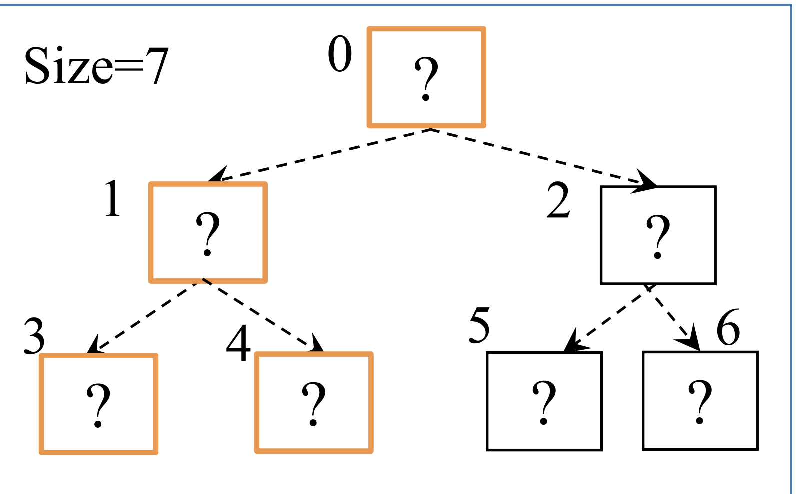
Pre-condition (of Max-Heapify):
The trees rooted at Left(i) and Right(i) are required to be max-heaps !

Build-3 (A)

1. $n \leftarrow A.length$
2. for $i \leftarrow \lfloor n/2 \rfloor - 1$ downto 0
3. Max-Heapify (A, i)

0	1	2	3	4	5	6
11	22	33	44	55	66	77

Building a Heap from Array



In the beginning:

0	1	2	3	4	5	6
11	22	33	44	55	66	77

i=2, after line 3:

0	1	2	3	4	5	6
11	22	77	44	55	66	33

i=1, after line 3:

0	1	2	3	4	5	6
11	55	77	44	22	66	33

i=0, during the execution of line 3:

0	1	2	3	4	5	6
77	55	11	44	22	66	33

i=0, after line 3:

0	1	2	3	4	5	6
77	55	66	44	22	11	33

Build-Max-Heap (A)

1. $n \leftarrow A.length$
2. for $i \leftarrow \lfloor n/2 \rfloor - 1$ downto 0
3. Max-Heapify (A, i)

Build-Max-Heap: Correctness

0	1	2	3	4	5	6
11	22	33	44	55	66	77

Build-Max-Heap (A)

1. $n \leftarrow$ A.*length*
2. for $i \leftarrow \lfloor n/2 \rfloor - 1$ downto 0
3. Max-Heapify (A, i)

Loop invariant

At the start of iteration of loop i, each node $i+1$, $i+2$, …, $n-1$ is the root of a max-heap

Initialization: We have: $i = \lfloor n/2 \rfloor - 1$ before 1st iteration. Each node $\lfloor n/2 \rfloor$, $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, …, $n-1$ is a leaf.

Maintenance: Any children of the current node i is greater than i. The pre-condition of Max-Heapify is true. After the call, the heap property holds at the subtree rooted at i. Decrementing i maintains the invariant.

Termination: We have: i=0 at termination. Node 0 is the root of the whole max-heap.

Heapsort: Example

Idea: extract the maximum item from the heap repeatedly

Heapsort (*A*)

1. **Build-Max-Heap** (*A*)
2. for *i* ← *A*.length–1 downto 1
3. swap *A*[0] with *A*[*i*]
4. *A.size* ← *A.size* – 1
5. Max-Heapify (*A*, 0)

After line 1:

0	1	2	3	4	5	6
77	55	66	44	22	11	33

i=6, after line 3:

0	1	2	3	4	5	6
33	55	66	44	22	11	**77**

i=6, after line 5:

0	1	2	3	4	5	6
66	55	33	44	22	11	**77**

i=5, after line 3:

0	1	2	3	4	5	6
11	55	33	44	22	**66**	**77**

i=5, after line 5:

0	1	2	3	4	5	6
55	44	33	11	22	**66**	**77**

Continue with these steps to get a sorted array ……

Heapsort: Running Time

Heapsort (A)

1. **Build-Max-Heap** (A)
2. for $i \leftarrow A$.length–1 downto 1
3. swap $A[0]$ with $A[i]$
4. $A.size \leftarrow A.size - 1$
5. Max-Heapify (A, 0)

- Build-Max-Heap: O(n) time
- Total n – 1 calls of Max-Heapify
  - Max-Heapify: O(log n) time
- Total time:

$= O(n) + (n - 1) * O(\log n)$

$= O(n + (n - 1) * \log n)$

$= O(n + n \log n)$

$= O(n \log n)$

Outline

- What is a *heap*?
- How do we maintain the *heap property*?
- How do we use a heap to *sort an array*?
- How do we *insert*, *delete*, *update* items in a heap?
  - All these operations use the "**Max-Heapify**" operation
- What are the variants of heaps?

Heap: Get-Max(A)

Get-Max(A)

1. return A[0]

0	1	2	3	4	5	6
77	55	66	44	22	11	/

- Time: O(1)

Heap: Extract-Max(A)

Extract-Max(A)

1. assert A.size ≥ 1
2. *max* ← A[0]
3. A[0] ← A[A.size–1]
4. A.size ← A.size – 1
5. Max-Heapify (A, 0)
6. return *max*

0	1	2	3	4	5	6
77	55	66	44	22	11	/

0	1	2	3	4	5	6
11	55	66	44	22	/	/

0	1	2	3	4	5	6
66	55	11	44	22	/	/

- Time: O(log n)

Pre-condition (of Max-Heapify):
The trees rooted at Left(i) and Right(i) are max-heaps !

Update-Key(A,i,k): Decrease

- Consider the case that the new key k is less than $A[i]$
- Should we go up or go down the tree?
- Time complexity: O(log n)
- Example: Update-Key(A, 1, 33)

Update-Key (A, i, k)

1. if $A[i] > k$
2. $A[i] \leftarrow k$
3. Max-Heapify (A, i)
4. else
5. ……

0	1	2	3	4	5	6
77	55	66	44	22	11	/

0	1	2	3	4	5	6
77	**33**	66	44	22	11	/

0	1	2	3	4	5	6
77	**44**	66	**33**	22	11	/

Update-Key(A,i,k): Increase

- Consider the case that the new key k is greater than $A[i]$
- Should we go up or go down the tree?
- Time complexity: O(log n)
- Example: Update-Key(A, 1, 88)

```
Update-Key (A, i, k)
if A[i] > k
    ……
else
    A[i] ← k
    while i ≥ 0 and A[Parent(i)]<A[i]
        swap A[i] with A[Parent(i)]
        i ← Parent(i)
```

0	1	2	3	4	5	6
77	55	66	44	22	11	/

0	1	2	3	4	5	6
77	**88**	66	44	22	11	/

0	1	2	3	4	5	6
88	**77**	66	44	22	11	/

Insert-Key(A,k)

Insert-Key(A, k)

1. A.size ← A.size + 1
2. A[A.size–1] ← –∞
3. Update-Key(A, A.size–1, k)

- Example: Insert-Key(A, 70)
- Time: O(log n)

0	1	2	3	4	5	6
77	55	66	44	22	11	/

0	1	2	3	4	5	6
77	55	66	44	22	11	**70**

0	1	2	3	4	5	6
77	55	**70**	44	22	11	**66**

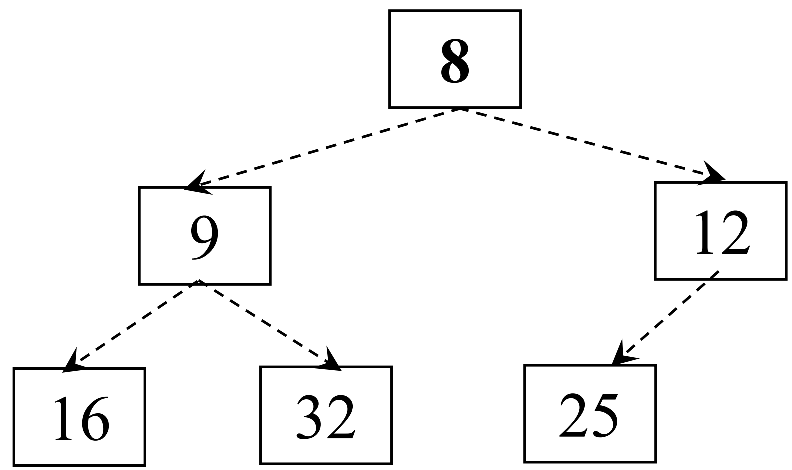
Outline

- What is a *heap*?
- How do we maintain the *heap property*?
- How do we use a heap to *sort an array*?
- How do we *insert*, *delete*, *update* items in a heap?
- What are the variants of heaps?

Min-Heap

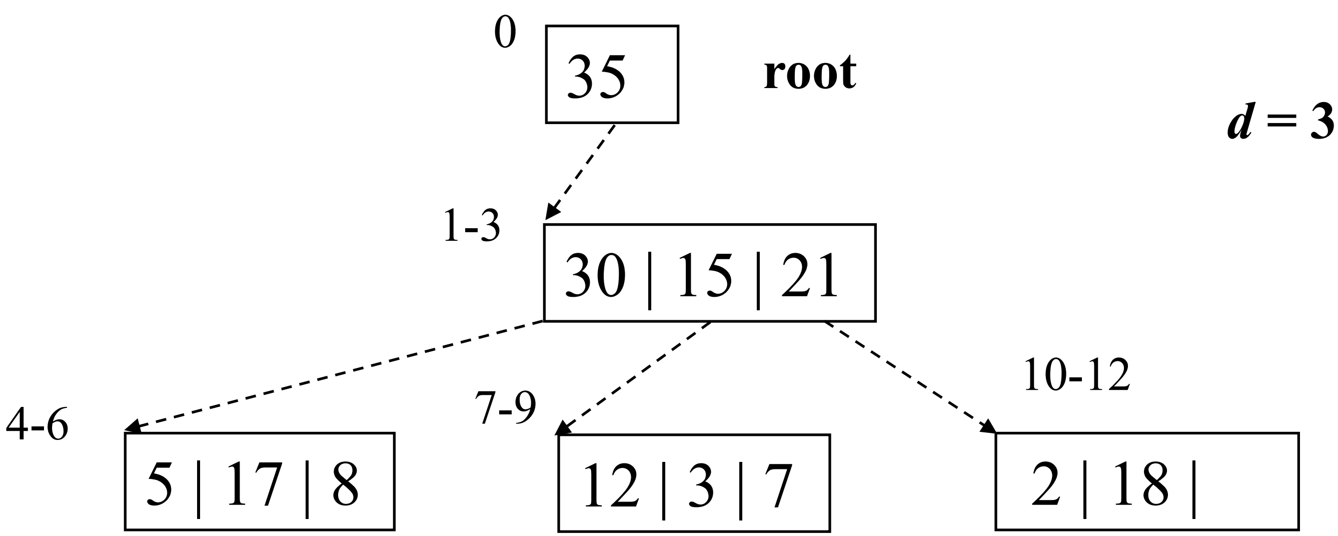
- *Min-heap* is a data structure that supports fast retrieval of the **minimum** value
  - Min-heap property: $A[\text{Parent}(i)] \leq A[i]$
  - The root (i=0) contains the smallest item
- Algorithms for max-heap can be easily modified to work on min-heap
  - How?

0	1	2	3	4	5
8	9	12	16	32	25



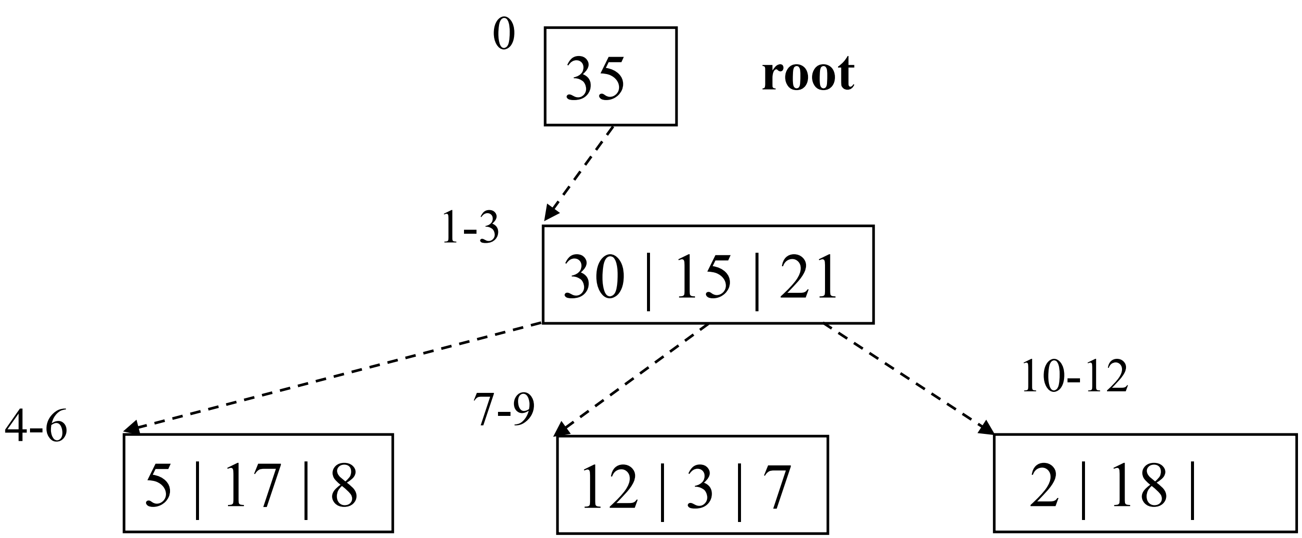
d-ary Heap

- *d*-ary heap is a generalization of binary heap
  - In *d*-ary heap, each node stores *d* keys
- How do we define the parent and children of a key?
  - Let *i* be the absolute position of a key in the array
  - Access parent: $\text{Parent}(i) = \lfloor (i-1)/ d \rfloor$
  - Access *j*-th child: $\text{Child}_j(i) = d\, i + j$



d-ary Heap Property

- *d*-ary heap property: $A[\text{Parent}(i)] \geq A[i]$
- Height $h = \lfloor \log_d n \rfloor$
  - Smaller height than binary heap
- Insert operations: $O(\log_d n)$ *faster*
- Extract-max operations: $O(d \log_d n)$ *slower*



$d = 3$

Some Questions about Heaps

Question 1

- Given two heaps A and B, how to print all their items in the descending order?

Question 2

- Given an array A, how to find the k-minimum value quickly by using a heap?

About Week 5 (2 Oct 2024)

- 1 Oct 2024 is the public holiday. No Class on Tuesday.
- 6:30 pm – 9:20 pm, 2 Oct 2024: the hybrid lab class (Online +lab rooms)
- Both Tuesday and Wednesday groups should join online or in person.
- Lab rooms are at PQ604ABC and PQ603. First come, first served. (because #Student > #lab computers).
- Online link of Microsoft Teams will be sent within one day before the class. Please use your PolyU Connect account to login MS Teams.
- Recorded video will be provided after the class, if you are not available to join.
- You are encouraged to use your own computer. So, you can easily continue the unfinished work after the class.
- If you use lab computers, please save the file in J: or the other proper space. Reboot will reset everything.
- If your laptop OS is macOS and you are new to the programming area, please use lab computer in Windows System.
- If you use your own laptop, please refer to previous guide to install JDK 21 and IntelliJ (community Edition 2024) before the class.

Information about Quiz 1

- Date/Time: a 45-minute time slot
  - Quiz 1 for Tuesday class: about 8:00pm, 8 Oct 2024,
  - Quiz 1 for Wednesday class: about 8:00pm, 9 Oct 2024.
  - No resit quiz will be made.
- Scope: Lectures 1 – 5
  - Data structures: stack, queue, linked list, tree, heap
  - Sorting algorithms
- Format: short questions on 3 hard-copy pages
  - E.g., draw the running steps of an algorithm
  - E.g., apply a data structure to solve a problem
  - Closed book

Summary

- The *heap structure* and *heap property*
- *Sort an array* by using a heap
- Heap operations
  - *Extract-Max*, *Update-Key*, *Insert-Key*
- How do we solve problems by using heap?
- Please read Chapter 9 in the book "*Data Structures and Algorithms in Java*", 6th Edition