

Balanced Binary Search Trees: Motivation

- After *insertions* into a BST, the **worst-case RT** of a *search* occurs when the *height h* is at its **maximum: $O(n)$** :
 - e.g., Entries were inserted in an decreasing order of their keys
(100, 75, 68, 60, 50, 1)
⇒ **One-path, left-slanted** BST
 - e.g., Entries were inserted in an increasing order of their keys
(1, 50, 60, 68, 75, 100)
⇒ **One-path, right-slanted** BST
 - e.g., Last entry's key is in-between keys of the previous two entries
(1, 100, 50, 75, 60, 68)
⇒ **One-path, side-alternating** BST
- To avoid the worst-case RT (\therefore a **ill-balanced tree**), we need to take actions **as soon as** the tree becomes **unbalanced**.

3 of 33

Priority Queues, Heaps, and Heap Sort

Learning Outcomes of this Lecture

This module is designed to help you understand:

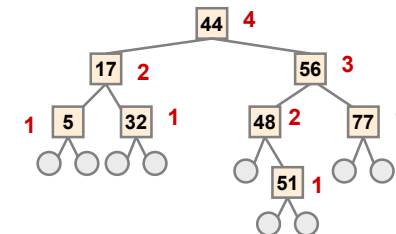
- When the **Worst-Case RT** of a **BST Search** Occurs
- Height-Balance** Property
- The **Priority Queue (PQ)** ADT
- Time Complexities of **List-Based PQ**
- The **Heap** Data Structure (Properties & Operations)
- Heap Sort**
- Time Complexities of **Heap-Based PQ**
- Heap** Construction Methods: Top-Down vs. Bottom-Up
- Array**-Based Representation of a **Heap**

2 of 33

Balanced Binary Search Trees: Definition

- Given a node p , the **height** of the subtree rooted at p is:

$$\text{height}(p) = \begin{cases} 0 & \text{if } p \text{ is external} \\ 1 + \text{MAX}(\{\text{height}(c) \mid \text{parent}(c) = p\}) & \text{if } p \text{ is internal} \end{cases}$$
- A **balanced** BST T satisfies the **height-balance property**:
For every **internal node** n , **heights** of n 's **child nodes** differ ≤ 1 .



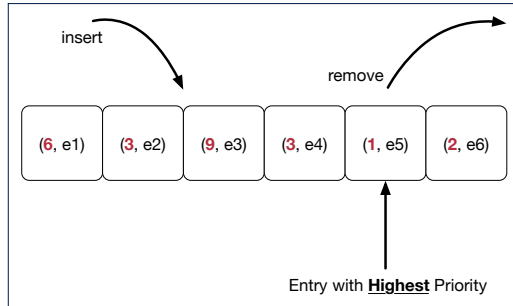
- Q: Is the above tree a **balanced BST**? ✓
- Q: Will the tree remain **balanced** after inserting 55? ✗
- Q: Will the tree remain **balanced** after inserting 63? ✓

4 of 33

What is a Priority Queue?



- A **Priority Queue (PQ)** stores a collection of *entries*.



- Each *entry* is a pair: an *element* and its *key*.
- The *key* of each *entry* denotes its *element's* "priority".
- Keys* in a Priority Queue (PQ) are **not** used for uniquely identifying an entry.

- In a PQ, the next entry to remove has the "**highest**" priority.
 - e.g., In the stand-by queue of a fully-booked flight, **frequent flyers** get the higher priority to replace any cancelled seats.
 - e.g., A network router, faced with insufficient bandwidth, may only handle **real-time tasks** (e.g., streaming) with highest priorities.

5 of 33

Two List-Based Implementations of a PQ



Consider two strategies for implementing a **PQ**, where we maintain:

- A list **always sorted** in a non-descending order [\approx INSERTIONSORT]
- An **unsorted** list [\approx SELECTIONSORT]

PQ Method	List Method	
	SORTED LIST	UNSORTED LIST
size	list.size $O(1)$	
isEmpty	list.isEmpty $O(1)$	
min	list.first $O(1)$	search min $O(n)$
insert	insert to "right" spot $O(n)$	insert to front $O(1)$
removeMin	list.removeFirst $O(1)$	search min and remove $O(n)$

7 of 33

The Priority Queue (PQ) ADT



- min**
 - [precondition: PQ is not empty]
 - [postcondition: return entry with **highest priority** in PQ]
- size**
 - [precondition: none]
 - [postcondition: return number of entries inserted to PQ]
- isEmpty**
 - [precondition: none]
 - [postcondition: return whether there is no entry in PQ]
- insert(k, v)**
 - [precondition: PQ is not full]
 - [postcondition: insert the input entry into PQ]
- removeMin**
 - [precondition: PQ is not empty]
 - [postcondition: remove and return a **min** entry in PQ]

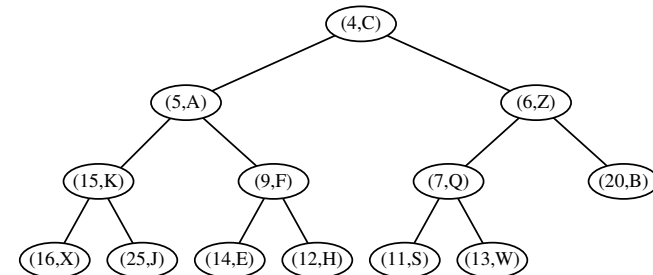
6 of 33

Heaps



A **heap** is a **binary tree** which:

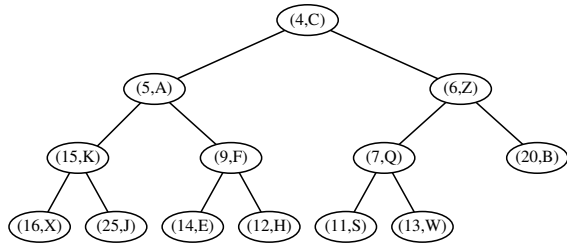
- Stores in each node an *entry* (i.e., *key* and *value*).



- Satisfies a **relational** property of stored **keys**
- Satisfies a **structural** property of tree **organization**

8 of 33

Heap Property 1: Relational

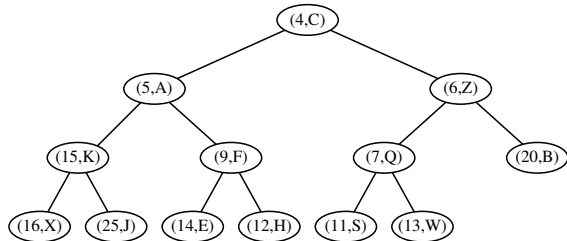


Keys in a **heap** satisfy the **Heap-Order Property** :

- Every node n (other than the root) is s.t. $key(n) \geq key(\text{parent}(n))$
 \Rightarrow Keys in a **root-to-leaf path** are sorted in a **non-descending order**.
 e.g., Keys in entry path $\langle (4, C), (5, A), (9, F), (14, E) \rangle$ are sorted.
 \Rightarrow The **minimal key** is stored in the **root**.
 e.g., Root $(4, C)$ stores the minimal key 4.
- Keys of nodes from **different subtrees** are **not** constrained at all.
 e.g., For node $(5, A)$, key of its **LST's** root (15) is **not minimal** for its **RST**.

9 of 33

Heap Property 2: Structural



A **heap** with **height h** satisfies the **Complete BT Property** :

- Nodes with **depth $\leq h - 2$** has **two** child nodes.
- Nodes with **depth $h - 1$** may have **zero, one, or two** child nodes.
- Nodes with **depth h** are filled from **left to right**.

Q. When the # of nodes is n , what is h ?

$$\lceil \log_2 n \rceil$$

Q. # of nodes from Level 0 through Level $h - 1$?

$$2^h - 1$$

Q. # of nodes at Level h ?

$$n - (2^h - 1)$$

Q. **Minimum** # of nodes of a complete BT?

$$2^h$$

Q. **Maximum** # of nodes of a complete BT?

$$2^{h+1} - 1$$

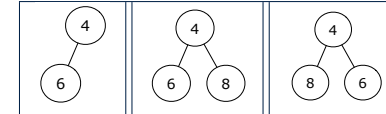
10 of 33

Heaps: More Examples

- The **smallest heap** is just an empty binary tree.
- The **smallest non-empty heap** is a **one-node heap**.
e.g.,



- Two-node and Three-node Heaps:**



- These are **not two-node heaps**:



11 of 33

Heap Operations

- There are **three main operations** for a **heap** :
 - Extract the Entry with Minimal Key:**
Return the stored entry of the **root**. [$O(1)$]
 - Insert a New Entry:**
A single **root-to-leaf path** is affected. [$O(h)$ or $O(\log n)$]
 - Delete the Entry with Minimal Key:**
A single **root-to-leaf path** is affected. [$O(h)$ or $O(\log n)$]
- After performing each operation,
both **relational** and **structural** properties must be maintained.

12 of 33

Updating a Heap: Insertion



To insert a new entry (k, v) into a heap with *height* h :

1. Insert (k, v) , possibly **temporarily** breaking the *relational property*.

- 1.1 Create a new entry $e = (k, v)$.
- 1.2 Create a new **right-most** node n at *Level* h .
- 1.3 Store entry e in node n .

After steps 1.1 and 1.2, the *structural property* is maintained.

2. Restore the **heap-order property (HOP)** using **Up-Heap Bubbling**:

- 2.1 Let $c = n$.
- 2.2 While **HOP** is not restored and c is not the root:
 - 2.2.1 Let p be c 's parent.
 - 2.2.2 If $key(p) \leq key(c)$, then **HOP** is restored.
 Else, swap nodes c and p . [“upwards” along n 's *ancestor path*]

Running Time?

- o All sub-steps in 1, as well as steps 2.1, 2.2.1, and 2.2.2 take $O(1)$.
- o Step 2.2 may be executed up to $O(h)$ (or $O(\log n)$) times.

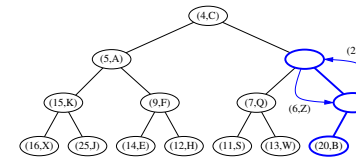
[$O(\log n)$]

13 of 33

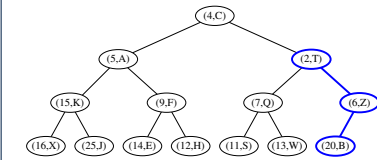
Updating a Heap: Insertion Example (1.2)



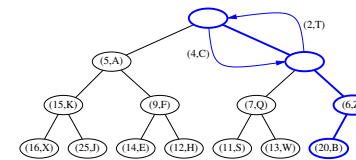
(4) **HOP** violated $\because 2 < 6 \therefore$ Swap.



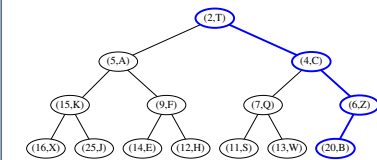
(5) After swap, entry $(2, T)$ prompted up.



(6) **HOP** violated $\because 2 < 4 \therefore$ Swap.



(7) Entry $(2, T)$ becomes root \therefore Done.

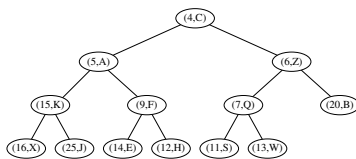


15 of 33

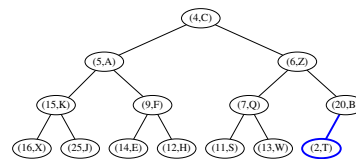
Updating a Heap: Insertion Example (1.1)



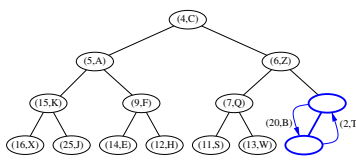
(0) A heap with height 3.



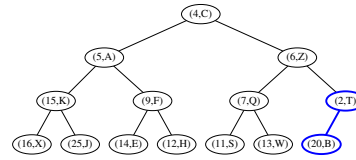
(1) Insert a new entry $(2, T)$ as the **right-most** node at Level 3. Perform **up-heap bubbling** from here.



(2) **HOP** violated $\because 2 < 20 \therefore$ Swap.



(3) After swap, entry $(2, T)$ prompted up.



14 of 33

Updating a Heap: Deletion



To delete the **root** (with the *minimal* key) from a heap with *height* h :

1. Delete the **root**, possibly **temporarily** breaking **HOP**.

- 1.1 Let the **right-most** node at *Level* h be n .
- 1.2 Replace the **root's** entry by n 's entry.
- 1.3 Delete n .

After steps 1.1 – 1.3, the *structural property* is maintained.

2. Restore **HOP** using **Down-Heap Bubbling**:

- 2.1 Let p be the **root**.
- 2.2 While **HOP** is not restored and p is not external:
 - 2.2.1 IF p has no **right child**, let c be p 's **left child**.
 Else, let c be p 's child with a **smaller key value**.
 - 2.2.2 If $key(p) \leq key(c)$, then **HOP** is restored.
 Else, swap nodes p and c . [“downwards” along a *root-to-leaf path*]

Running Time?

- o All sub-steps in 1, as well as steps 2.1, 2.2.1, and 2.2.2 take $O(1)$.
- o Step 2.2 may be executed up to $O(h)$ (or $O(\log n)$) times.

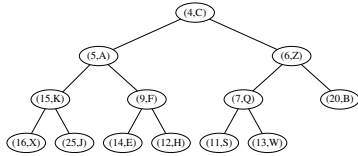
[$O(\log n)$]

16 of 33

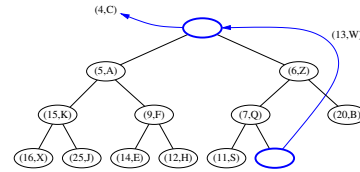
Updating a Heap: Deletion Example (1.1)



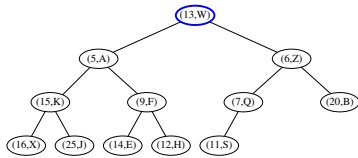
(0) Start with a heap with height 3.



(1) Replace root with (13, W) and delete **right-most** node from Level 3.

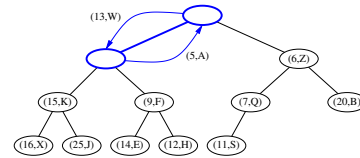


(2) (13, W) becomes the root. Perform **down-heap bubbling** from here.



17 of 33

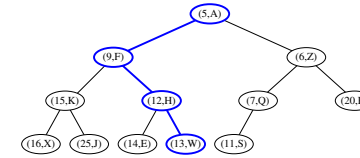
(3) Child with smaller key is (5, A). **HOP violated** ∴ 13 > 5 ∴ Swap.



Updating a Heap: Deletion Example (1.3)



(8) After swap, entry (13, W) becomes an external node ∴ Done.

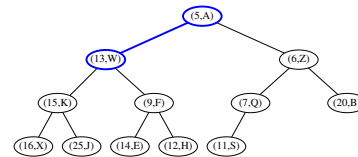


19 of 33

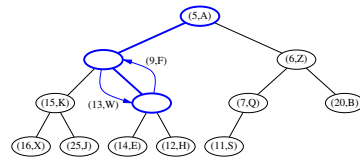
Updating a Heap: Deletion Example (1.2)



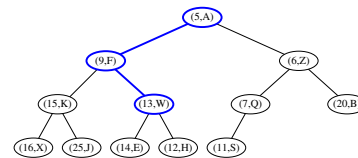
(4) After swap, entry (13, W) demoted down.



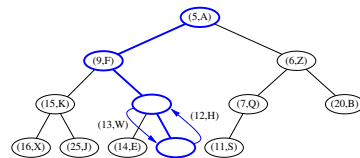
(5) Child with smaller key is (9, F). **HOP violated** ∴ 13 > 9 ∴ Swap.



(6) After swap, entry (13, W) demoted down.



(7) Child with smaller key is (12, H). **HOP violated** ∴ 13 > 12 ∴ Swap.



18 of 33

Heap-Based Implementation of a PQ



PQ Method	Heap Operation	RT
min	root	$O(1)$
insert	insert then up-heap bubbling	$O(\log n)$
removeMin	delete then down-heap bubbling	$O(\log n)$

20 of 33

Top-Down Heap Construction: List of Entries is Not Known in Advance

Problem: Build a heap out of N entries, supplied one at a time.

- Initialize an **empty heap h** . [$O(1)$]
 - As each new entry $e = (k, v)$ is supplied, **insert e** into h .
 - Each insertion triggers an **up-heap bubbling** step, which takes $O(\log n)$ time. [$n = 0, 1, 2, \dots, N - 1$]
 - There are N insertions.
- \therefore Running time is $O(N \cdot \log N)$

21 of 33

Bottom-Up Heap Construction: List of Entries is Known in Advance

Problem: Build a heap out of N entries, supplied all at once.

- **Assume:** The resulting heap will be **completely filled** at all levels.
 $\Rightarrow N = 2^{h+1} - 1$ for some **height $h \geq 1$** [$h = (\log(N + 1)) - 1$]
- Perform the following steps called **Bottom-Up Heap Construction**:
 - Step 1:** Treat the first $\frac{N+1}{2^1}$ list entries as heap roots.
 $\therefore \frac{N+1}{2^1}$ heaps with height 0 and size $2^1 - 1$ constructed.
 - Step 2:** Treat the next $\frac{N+1}{2^2}$ list entries as heap roots.
 - Each **root** sets two heaps from **Step 1** as its **LST** and **RST**.
 - Perform **down-heap bubbling** to restore **HOP** if necessary. $\therefore \frac{N+1}{2^2}$ heaps, each with height 1 and size $2^2 - 1$, constructed.
 - ...
 - Step $h+1$:** Treat next $\frac{N+1}{2^{h+1}} = \frac{(2^{h+1}-1)+1}{2^{h+1}} = 1$ list entry as heap root.
 - Each **root** sets two heaps from **Step h** as its **LST** and **RST**.
 - Perform **down-heap bubbling** to restore **HOP** if necessary. $\therefore \frac{N+1}{2^{h+1}} = 1$ heap, each with height h and size $2^{h+1} - 1$, constructed.

22 of 33

Bottom-Up Heap Construction: Example (1.1)

- Build a heap from the following list of 15 keys:

$\langle 16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14 \rangle$

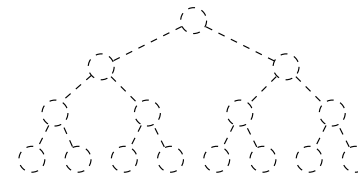
- The resulting heap has:
 - **Size N** is 15
 - **Height h** is $(\log(15 + 1)) - 1 = 3$
- According to the **bottom-up heap construction** technique, we will need to perform $h + 1 = 4$ steps, utilizing 4 sublists:

$\langle \underbrace{16, 15, 4, 12, 6, 7, 23, 20}_{\frac{15+1}{2^1} = 8}, \underbrace{25, 9, 11, 17}_{\frac{15+1}{2^2} = 4}, \underbrace{5, 8}_{\frac{15+1}{2^3} = 2}, \underbrace{14}_{\frac{15+1}{2^4} = 1} \rangle$

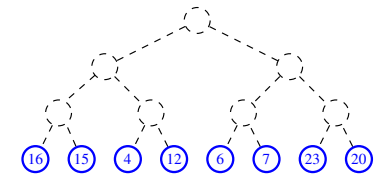
23 of 33

Bottom-Up Heap Construction: Example (1.2)

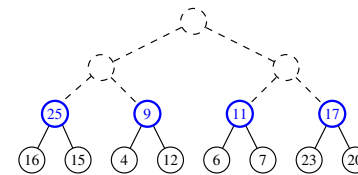
We know in advance to build a heap with height 3 and size $2^{3+1} - 1 = 15$



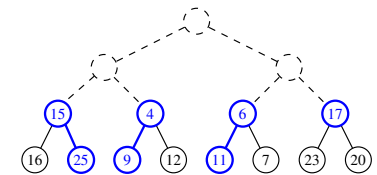
(Step 1) Treat first $\frac{15+1}{2^1}$ entries as roots.
 \therefore 8 one-node heaps.



(Step 2) Treat next $\frac{15+1}{2^2}$ entries as roots.
 Set LST and RST of each root.



(Step 2 cont.) Down-heap bubbling.
 \therefore 4 three-node heaps.

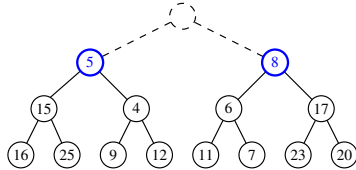


24 of 33

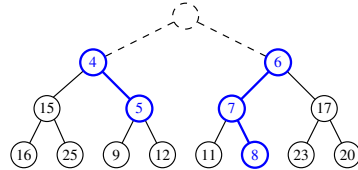
Bottom-Up Heap Construction: Example (1.3)



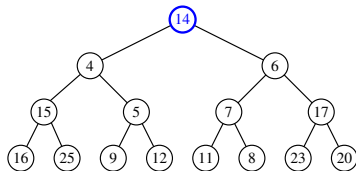
(Step 3) Treat next $\frac{15+1}{2^3}$ entries as roots.
Set LST and RST of each root.



(Step 3 cont.) Down-heap bubbling.
 \therefore 2 three-node heaps.

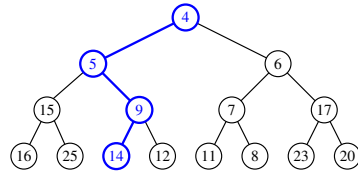


(Step 4) Treat next $\frac{15+1}{2^4}$ entry as roots.
Set LST and RST of each root.



25 of 33

(Step 4 cont.) Down-heap bubbling.
 \therefore 1 fifteen-node heap.



RT of Bottom-Up Heap Construction



- Intuitively, the majority of the intermediate roots from which we perform **down-heap bubbling** are of very **small height values**:
 - The first $\frac{n+1}{2}$ 1-node heaps with **height 0** require **no** down-heap bubbling. [About 50% of the list entries processed]
 - Next $\frac{n+1}{4}$ 3-node heaps with **height 1** require down-heap bubbling. [Another 25% of the list entries processed]
 - Next $\frac{n+1}{8}$ 7-node heaps with **height 2** require down-heap bubbling. [Another 12.5% of the list entries processed]
 - ...
 - Next two $\frac{N-1}{2}$ -node heaps with **height (h - 1)** require down-heap
 - Final one **N**-node heaps with **height h** requires down-heap bubbling.
- Running Time of the **Bottom-Up Heap Construction** takes only **$O(n)$** .

26 of 33

The Heap Sort Algorithm



Sorting Problem:

Given a list of **n** numbers $\langle a_1, a_2, \dots, a_n \rangle$:

Precondition: NONE

Postcondition: A permutation of the input list $\langle a'_1, a'_2, \dots, a'_n \rangle$ sorted in a non-descending order (i.e., $a'_1 \leq a'_2 \leq \dots \leq a'_n$)

The **Heap Sort** algorithm consists of **two** phases:

- Construct** a **heap** of size **N** out of the input array.
 - Approach 1: Top-Down "Continuous-Insertions" [$O(N \cdot \log N)$]
 - Approach 2: Bottom-Up Heap Construction [$O(N)$]
- Delete** **N** entries from the heap.
 - Each deletion takes **$O(\log N)$** time.
 - 1st deletion extracts the **minimum**, 2nd deletion the 2nd **minimum**, ...
 - \Rightarrow Extracted **minimums** from **N** deletions form a **sorted** sequence.

\therefore Running time of the **Heap Sort** algorithm is **$O(N \cdot \log N)$** .

27 of 33

The Heap Sort Algorithm: Exercise



Sort the following array of integers

$\langle 16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14 \rangle$

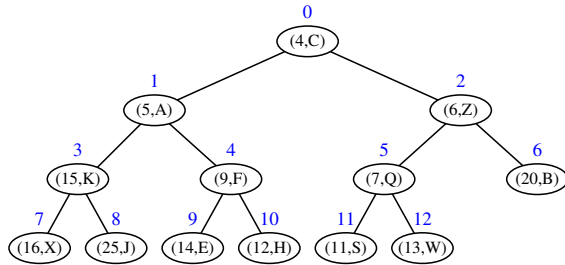
into a **non-descending** order using the **Heap Sort Algorithm**.

Demonstrate:

- Both **top-down** and **bottom-up** heap constructions in Phase 1
- Extractions of minimums in Phase 2

28 of 33

Array-Based Representation of a CBT (1)

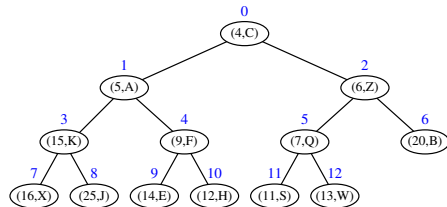


$$index(x) = \begin{cases} 0 & \text{if } x \text{ is the root} \\ 2 \cdot index(\text{parent}(x)) + 1 & \text{if } x \text{ is a left child} \\ 2 \cdot index(\text{parent}(x)) + 2 & \text{if } x \text{ is a right child} \end{cases}$$

(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

29 of 33

Array-Based Representation of a CBT (2)



(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

- Q1:** Where are nodes at **Levels 0 .. h - 1** stored in the array?
Indices $0 \dots (2^h - 2) \equiv 0 \dots (2^{\lceil \log_2 N \rceil} - 2)$ [e.g., Indices $0 \dots 2^3 - 2$]
- Q2:** Where are nodes at **Level h** stored in the array?
Indices $2^h - 1 \dots (N - 1) \equiv 2^{\lceil \log_2 N \rceil} - 1 \dots (N - 1)$ [e.g., Indices $7 \dots 12$]
- Q3:** How do we determine if a non-root node x is a **left or right child**?
IF $index(x) \% 2 == 1$ THEN left ELSE right
- Q4:** Given a non-root node x , how do we determine the **index of x 's parent**?
IF $index(x) \% 2 == 1$ THEN $\frac{index(x)-1}{2}$ ELSE $\frac{index(x)-2}{2}$

30 of 33

Index (1)



Learning Outcomes of this Lecture

Balanced Binary Search Trees: Motivation

Balanced Binary Search Trees: Definition

What is a Priority Queue?

The Priority Queue (PQ) ADT

Two List-Based Implementations of a PQ

Heaps

Heap Property 1: Relational

Heap Property 2: Structural

Heaps: More Examples

Heap Operations

31 of 33

Index (2)



Updating a Heap: Insertion

Updating a Heap: Insertion Example (1.1)

Updating a Heap: Insertion Example (1.2)

Updating a Heap: Deletion

Updating a Heap: Deletion Example (1.1)

Updating a Heap: Deletion Example (1.2)

Updating a Heap: Deletion Example (1.3)

Heap-Based Implementation of a PQ

Top-Down Heap Construction:

List of Entries is Not Known in Advance

Bottom-Up Heap Construction:

List of Entries is Known in Advance

32 of 33



Index (3)

[Bottom-up Heap Construction: Example \(1.1\)](#)

[Bottom-up Heap Construction: Example \(1.2\)](#)

[Bottom-up Heap Construction: Example \(1.3\)](#)

[RT of Bottom-up Heap Construction](#)

[The Heap Sort Algorithm](#)

[The Heap Sort Algorithm: Exercise](#)

[Array-Based Representation of a CBT \(1\)](#)

[Array-Based Representation of a CBT \(2\)](#)