**Prepared by:** [Arun Konagurthu]

# FIT3155: Advanced Algorithms and Data Structures

## Week 5: **Binomial heap and its amortized analysis**

Faculty of Information Technology, Monash University

# What is covered in this lecture?

Binomial heap and its amortized analysis

# Source material and recommended reading

- Weiss, Data Structures and Algorithm Analysis (Chapters 6.8, 11.1, 11.2)
- Cormen et al., Introduction to Algorithms (Chapter 19) [link]

# Priority queues (implemented using heaps)

Recall from FIT2004 that the heap data structure was used in several applications:

- Heap sort
- Dijkstra's shortest path algorithm
- Prim's algorithm

Recall also that this data structure supports the following operations*:

- **`insert`** a new element (key/priority+payload) into a heap
- identify the **`min`** element in an existing heap
- **`extract-min`** (identify and delete min) element in an existing heap
- **`decrease-priority`** of an element in an existing heap

*As with these slides, default heap operations are defined over a min-heap. One could alternatively define **`max`**, **`extract-max`**, **`increase-priority`** operations on a max-heap.

# Mergeable heaps

Today (binomial heap) and next start of next lecture (Fibonacci heap), we will learn about mergeable heaps that support (at least) the following operations:

**insert**: inserts a new element into the existing heap

**min**: finds the min element in the heap

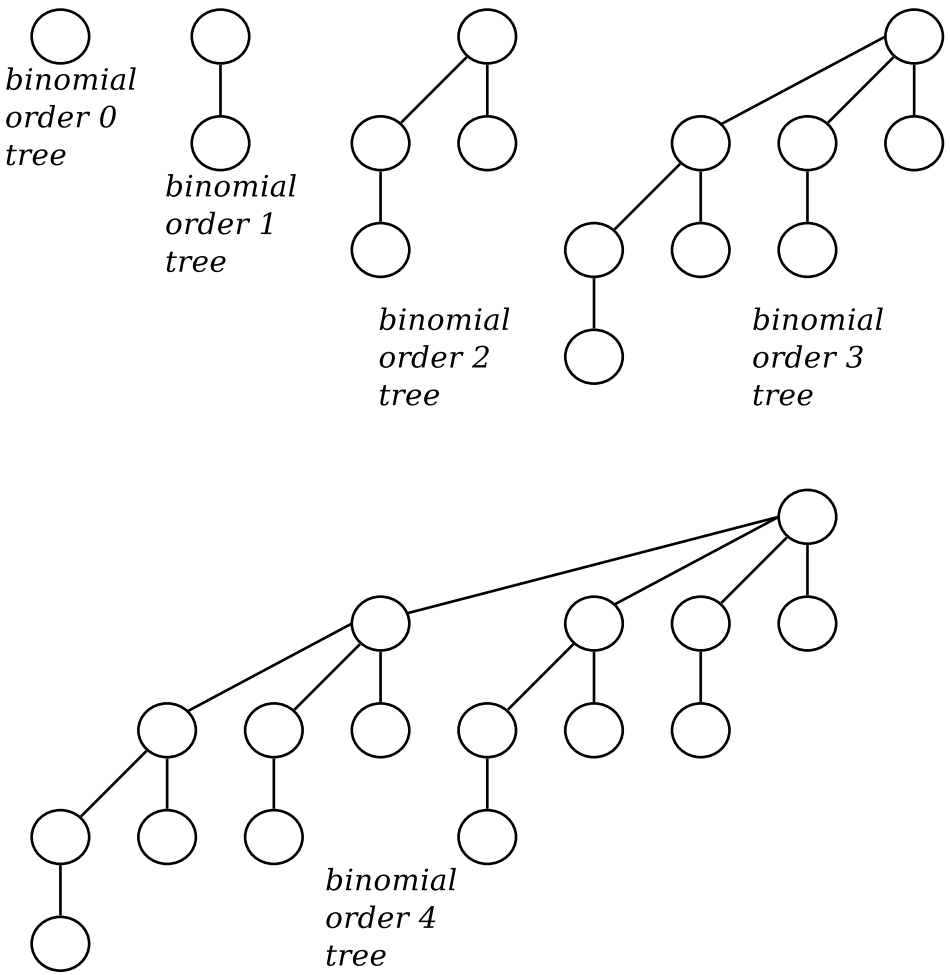
**extract-min**: finds and deletes the min element in the heap

**merge**: merges two heaps into one

**decrease-priority**: decreases the elements key/priority

**delete**: removes an element from the heap

# Before Binomial **heap**, let us define a binomial **tree**




Binomial trees are defined recursively:

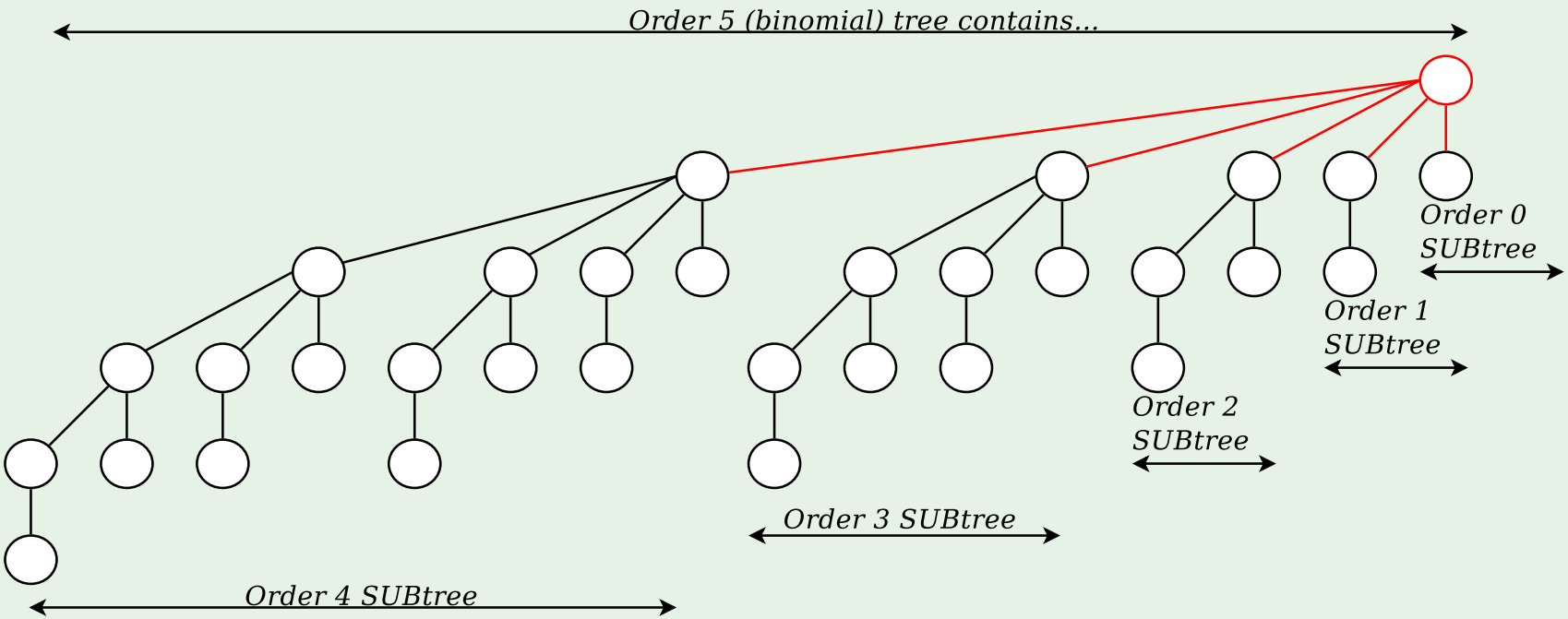
- The binomial tree of order 0 (or $B_0$ in short) is a single node tree
- The binomial tree of order 1 ($B_1$) is created from two $B_0$ trees, by making one $B_0$ tree the child of the other.
- The binomial tree of order 2 ($B_2$) is created from two $B_1$ trees, by making one $B_1$ tree the child of the other.
- and so on...

# Properties of a Binomial **tree**

Any binomial tree of order k has the following properties:

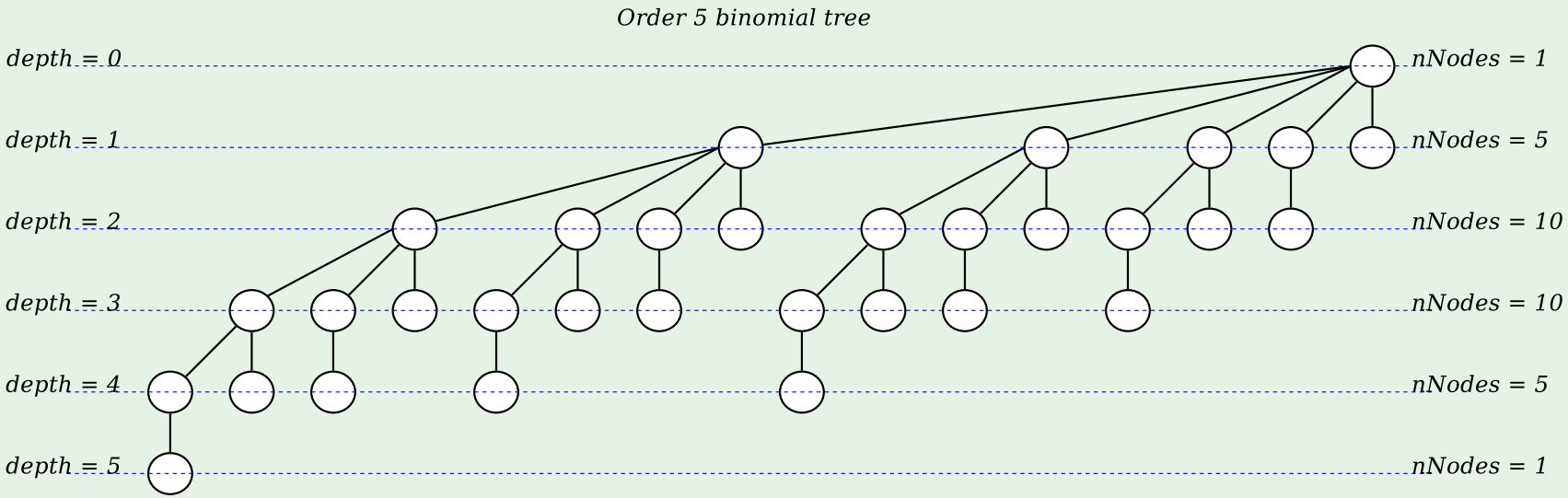
- The number of nodes in any $B_k$ is $2^k$.
- The height of any $B_k$ is $k$.
- The root node of any $B_k$ tree has $k$ subtrees as children.
- Deleting the root node of $B_k$ (with its edges/links) yields $k$ independent lower order binomial trees $B_{k-1}, B_{k-2}, \ldots, B_0$.

## Example



# Why are these trees called **binomial**?

## Example: $B_5$




## Main property

A main property of any $B_k$ tree is that the **number of nodes** at any given depth $d$ is given by the **binomial coefficient** $\binom{k}{d}$, that is "$k$-choose-$d$"

# What is a binomial **heap**?

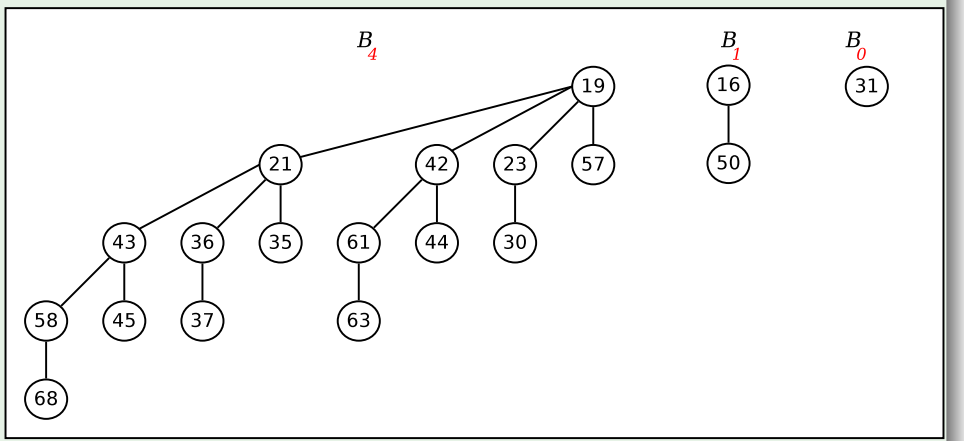
A binomial **heap** is a collection/set of binomial **trees** such that:

- each binomial tree in the set satisfies the heap property – i.e., each tree-node's key/priority is $\leq$ its children's keys/priorities.
- There is **at most** one (i.e. either 0 or 1) binomial tree of any given order in that set.

## Example

On the right is a binomial **heap** that contains a collection/set of binomial **trees**:

- one $B_4$ tree
- zero $B_3$ tree
- zero$B_2$ tree
- one $B_1$ tree
- one $B_0$ tree



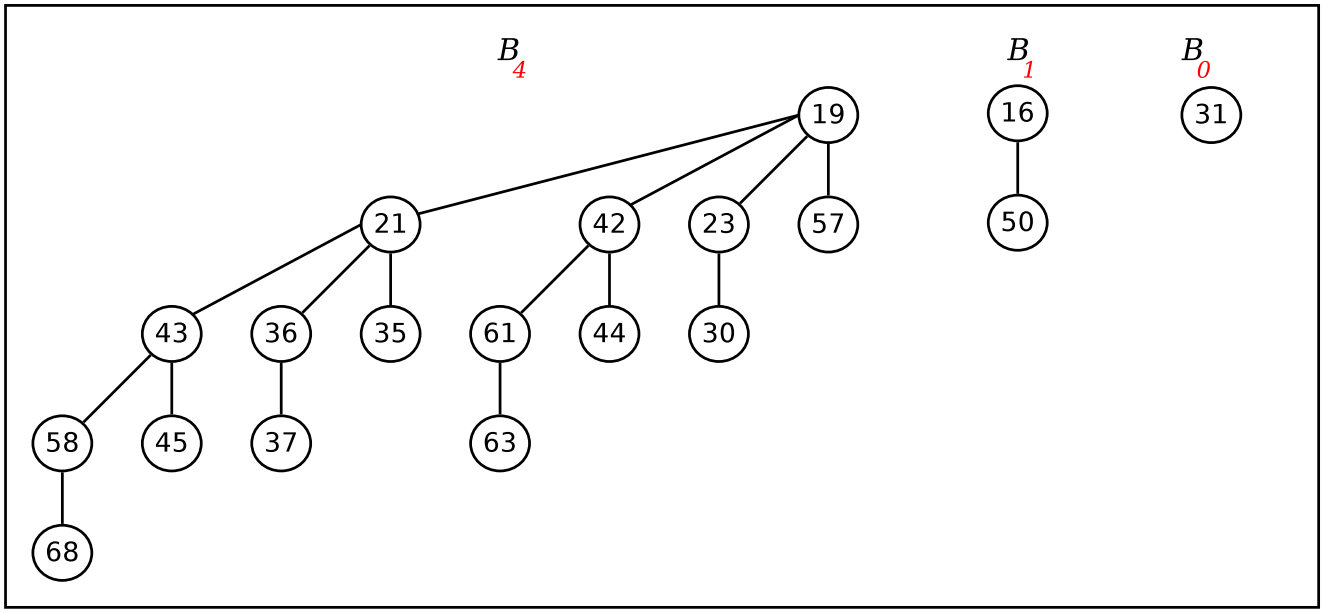
# Binomial heap properties

## Properties

For any binomial **heap** containing $N$ elements, the following properties hold:

- There are at most $\lfloor \log_2 N \rfloor + 1$ binomial **trees**
- The height of each binomial **tree** is $\leq \lfloor \log_2 N \rfloor$
- The '1's in the **binary representation** of $N$ tell us which order binomial **trees** are present in the collection forming this binomial **heap** of $N$ elements.
- the element with **minimum** key is one of the of root nodes of the **trees** in the collection.

# Binomial heap properties – Example




## Example

For the above binomial **heap**:

- $N = 19$.
- Number of trees is 3
- binary representation of 19 is: $\overset{4}{1}\ \overset{3}{0}\ \overset{2}{0}\ \overset{1}{1}\ \overset{0}{1}$ (therefore contains $B_4, B_1, B_0$)

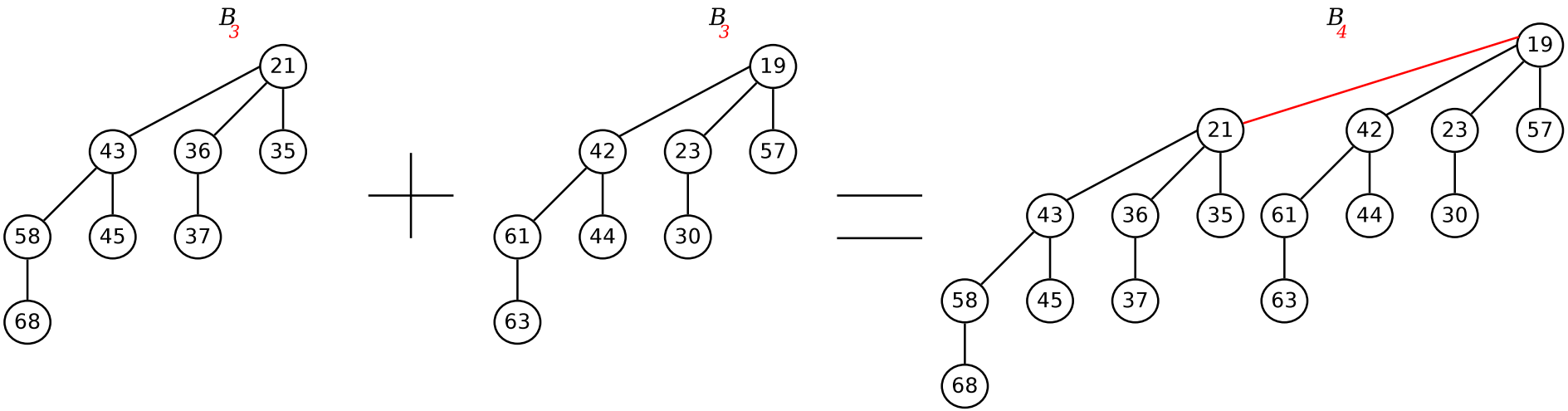
# Representing a binomial heap

- Unlike **binary** heaps, **binomial** heaps are stored explicitly using a **tree** data structure.
- Each node $x$:
  - is denoted by a **key**,
  - has associated **payload** information
  - has a pointer **parent**$[x]$ to its parent node
  - has a pointer **child**$[x]$ to its **leftmost** child node
    - ⋆ If node $x$ has zero children, then **child**$[x] = nil$
  - has a pointer **sibling**$[x]$ to the immediate **sibling** of $x$ to its right.
    - ⋆ If node $x$ is the rightmost child of its parent, then **sibling**$[x] = nil$
  - stores **degree**$[x]$ which is the number of children of $x$ (i.e., same as the **order** of the binomial tree rooted at $x$)
- Finally, the roots of the binomial tress within a binomial heap are organized in a linked list, referred to as the **root list**.

operations on a binomial heap

# Merging two binomial **trees** into one

- First, merging two binomial **trees**, each of the **same** order (say) $k$ results in an order $k+1$ binomial tree, where:
  - the two roots are linked, such that...
  - ...the root containing the **larger** key becomes the **child** of the smaller root.



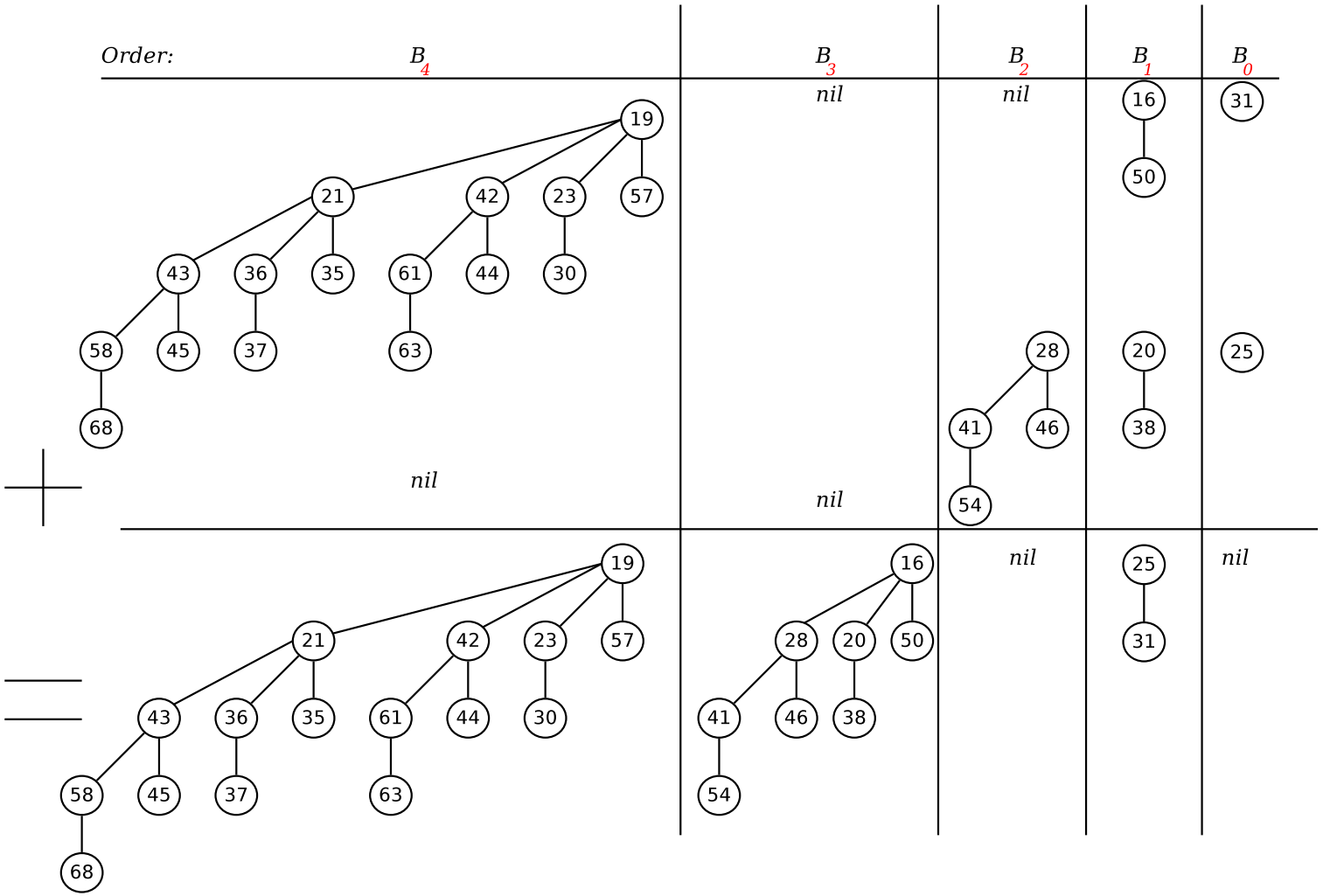
# Binomial **heap** operation – **merge** two binomial **heaps** into one

- With merging of two binomial **trees** established (see previous slide), we can now define **merging** of two binomial **heaps**.
- Heaps are merged in a way that is reminiscent of how we add two numbers in binary:

## Example: addition of $19 + 7 = 26$ in binary

| Order: | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|
| carry: | | 1 | 1 | 1 | |
| | 1 | 0 | 0 | 1 | 1 |
| $+$ | 0 | 0 | 1 | 1 | 1 |
| Result: | 1 | 1 | 0 | 1 | 0 |

# Example of merging 2 binomial heaps containing 19 and 7 elements each




(To be discussed during the lecture)

# Running time of **merge** operation between 2 binomial heaps

- Running time is $O(\log N)$ worst-case – why?
  - time is bounded by maximum number of possible merges between trees of the same order within the heaps.
  - the number of trees in each heap containing $N$ elements is bounded by $\lfloor \log N \rfloor + 1$
  - merging two heaps in worst case requires $2 \times (\lfloor \log N \rfloor + 1)$ tree merges

# Binomial **heap** operation – **extract-min**

We use this to identify and delete the minimum element among all **root nodes** of the trees in the heap.

- Identify the **min** root node among the trees in the heap.
- From slide #8, we know that deleting the root node of any $B_k$ tree yields: $B_{k-1}, B_{k-2}, \ldots, B_0$.
- If we promote these subtrees to the root level of the existing binomial heap...
- ...this might create multiple trees of the same order (violating the definition of a binomial heap – see slide #10).
- So, progressively **merge** the binomial trees of the same order (starting from 0) until the binomial heap definition is satisfied.

(Example will be handled during the lecture)

# Running time of **extract-min** operation

- Running time is $O(\log N)$ worst-case – why?
- Effort required to find the **min** is $O(\log N)$. (see slide #11)
- Effort required to promote subtrees formed upon deletion to root level is $O(\log N)$ – the number of these subtrees is bounded by $\lfloor \log N \rfloor$.
- Effort required to merge multiple trees into a binomial heap is also $O(\log N)$. (see slide #18)
- Total effort: $O(\log N)$

# Binomial **heap** operation – **decrease-priority**

We want to decrease priority of any node $x$ in a binomial heap containing $N$ elements. [†]

- decrease priority of node $x$.
- if min-heap property is violated (i.e. $x < parent[x]$), bubble up node $x$.
- Running time (worst-case): $O(\log N)$ – depth of the binomial tree in which $x$ resides is bounded above by $\lfloor \log N \rfloor$

(Example will be handled during the lecture)

[†]Note: as with binary heaps, binomial (and Fibonacci) heaps are inefficient to **search** for any node $x$ (except the root); For this reason, **decrease-priority**$(x)$ assumes a pointer to $x$ as part of its input.

# Binomial **heap** operation – **delete**

We want to delete any node $x$ in a binomial heap containing $N$ elements. ‡

- run **decrease-priority** by setting $x$ to $-\infty$.
- run **extract-min**.
- Running time (worst-case): $O(\log n)$.

‡Note: as with binary heaps, binomial (and Fibonacci) heaps are inefficient to **search** for any node $x$ (except the root); For this reason, **delete**$(x)$ assumes a pointer to $x$ as part of its input.

# Binomial **heap** operation – **insert**

We want to insert a new element $x$ into an existing binomial heap $H_1$

- Make a new binomial heap $H_2$ with $x$ as its only element.
- run **merge**$(H_1, H_2)$.
- At face value, the runtime per single **insert** takes $O(\log N)$ effort.

# Amortized analysis of **insert** operation

Consider the problem of building a **binomial** heap of $N$ elements:

- From FIT2004, we know that at least a **binary** heap of $N$ elements can be built in $O(N)$ time.
- What about a **binomial** heap then?

**claim**

A **binomial** heap of $N$ elements can be built by $N$ successive inserts in $O(N)$-time.

# Amortized analysis of **`insert`** operation ...continued(2)

- Time required for inserting **each** element $x$ into a heap $H_1$ (starting from an empty heap) involves:
  - time to create a new binomial heap $H_2$ containing only $1$ element $x$ – which requires constant effort, **`plus`**
  - time to merge $H_2$ into $H_1$. It isn't fully clear yet how many merges (between same-order binomial trees) will be required in each insert operation.
- Total over $N$ insertions requires:
  - $O(N)$ **`plus`**
  - total merging time.

# Amortized analysis of **`insert`** operation ...continued(3)

It is easy to see (by beholding how the numbers starting from 0 change when 1 is added each time):

- the first insertion into an empty $H_1$ heap requires zero merges. Why?
- the second insertion involves exactly one merge between two $B_0$ binomial trees, yielding a heap containing one $B_1$ tree.
- the third insertion involves zero merges
  - $H_1$ before insertion contains 2 elements (contained in 1 $B_1$ tree).
  - merging the new inserted element into $H_1$ adds only a new $B_0$ tree to the existing $B_1$ tree. Therefore no merges.
- the fourth insertion involves exactly two merges – why?.
- the fifth insertion involves zero merges – why?
- the sixth insertion involves zero merges – why?
- $\vdots$

# Amortized analysis of **insert** operation ...continued(3)

When inserting $N$ elements, if the binary representation of number elements in $H_1$ before each insertion ends in

- .....**0**, the effort takes only 1 unit of time.
- .....**0**1, the effort takes only 2 units of time.
- .....**0**11, the effort takes only 3 units of time.
- ....**0**111, the effort takes only 4 units of time.
- ...**0**1111, the effort takes only 5 units of time.
- $\vdots$

## Total time over $N$ insertions

- $T = \frac{N}{2} \times 1 + \frac{N}{4} \times 2 + \frac{N}{8} \times 3 \ldots \leq 2N$
- Such series is called an Arithmetico-Geometric series.

Thus total time is bounded by $O(N)$, implying that each **insert** into a binomial heap is $O(1)$ amortized!

# Summary

| Operation | Binary heap | Binomial heap |
|---|---|---|
| **make-new-heap** | $O(1)$ | $O(1)$ |
| **min** | $O(1)$ | $O(\log N)$ |
| **extract-min** | $O(\log N)$ | $O(\log N)$ |
| **merge** | $O(N)$ | $O(\log N)$ |
| **decrease-priority** | $O(\log N)$ | $O(\log N)$ |
| **delete** | $O(\log N)$ | $O(\log N)$ |
| **insert** | $O(\log N)$ worst-case<br>$O(1)$ amortized | $O(\log N)$ worst-case<br>$O(1)$ amortized |

**In the next lecture...**

(1) Fibonacci heaps and amortized analysis (2) B-Trees

-=o0o=-

END

-=o0o=-