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

Original reference

Jean Vuillemin, A Data Structure for Manipulating Priority Queues
Communications of the ACM, 21(4) 309-314. [link]

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 (containing key w/ payload) into a heap
- identify the **min** element in an existing heap
- **extract-min** (identify and delete min) element in an existing heap
- **decrease-key** of an element in an existing heap

*As with these slides, default heap operations in the rest of the slides are defined over a min-heap. One could alternatively define **max**, **extract-max**, **increase-key** operations on a max-heap.

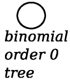
Mergeable heaps

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-key**: decreases the elements key
- **delete**: removes an element from the heap

[†] Again, the slides define these default heap operations over a min-heap. One could alternatively define **max**, **extract-max**, **increase-key** operations on a max-heap.

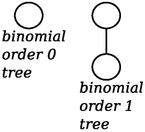
Before Binomial **heap**, let us define a binomial **tree**



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...

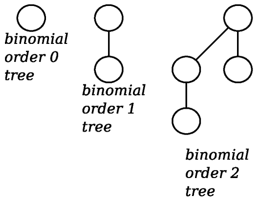
Before Binomial **heap**, let us define a binomial **tree**



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...

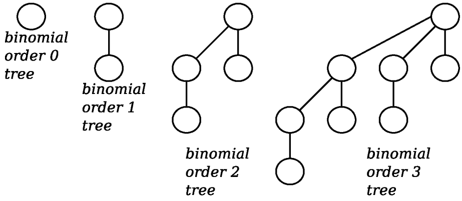
Before Binomial **heap**, let us define a binomial **tree**



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...

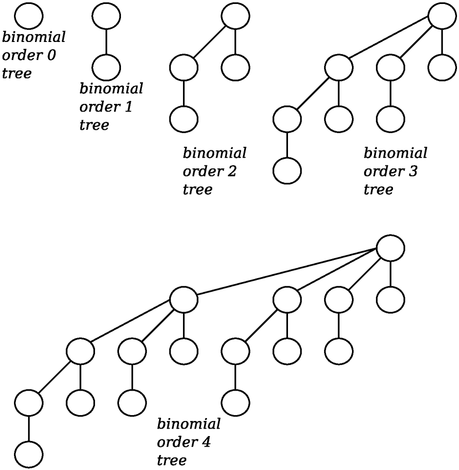
Before Binomial **heap**, let us define a binomial **tree**



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...

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



Binomial trees are defined recursively:

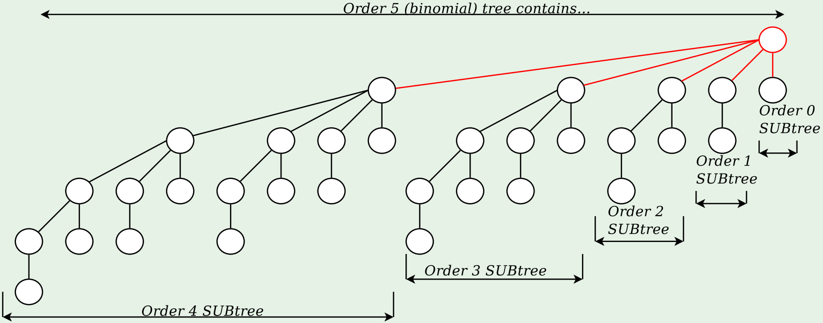
- The binomial tree of order 0 (or B_0 in short) is a single node tree
- The binomial tree of order 1 (B_1) is created from two B_0 trees, by making one B_0 tree the child of the other.
- The binomial tree of order 2 (B_2) is created from two B_1 trees, by making one B_1 tree the child of the other.
- 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: Binomial tree, B_5:

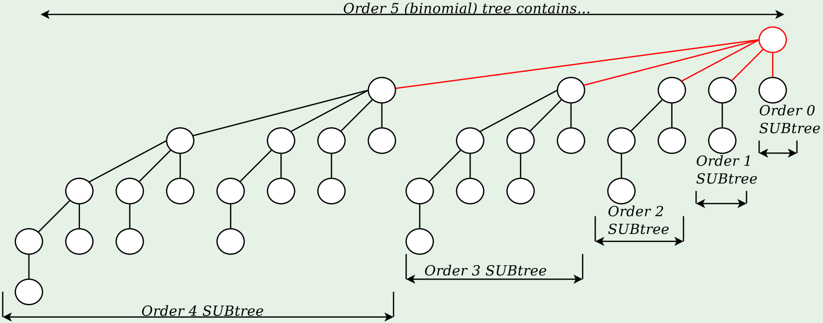


Properties of a Binomial **tree**

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: Binomial tree, B_5:

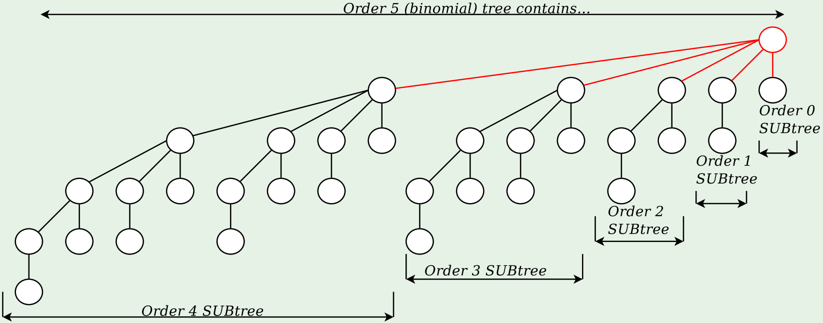


Properties of a Binomial tree

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: Binomial tree, B_5:

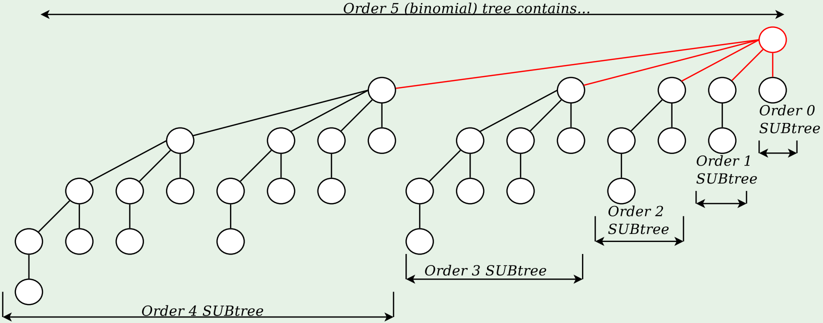


Properties of a Binomial **tree**

Any binomial tree of order k has the following properties:

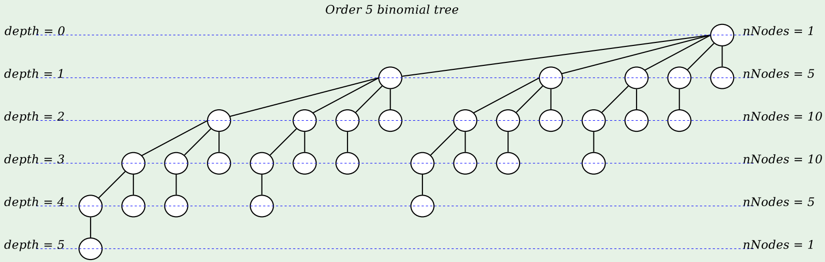
- The number of nodes in any B_k is 2^k.
- The height of any B_k is k.
- The root node of any B_k tree has k subtrees as children.
- Deleting the root node of B_k (with its edges/links) yields k independent lower order binomial trees $B_{k-1}, B_{k-2}, \ldots, B_0$.

Example: Binomial tree, B_5:



Why are these trees called **binomial**?

Example: Binomial tree, 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**?

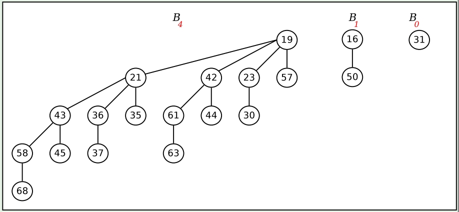
A binomial **heap** is a forest (i.e., a set) of binomial **trees** such that:

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

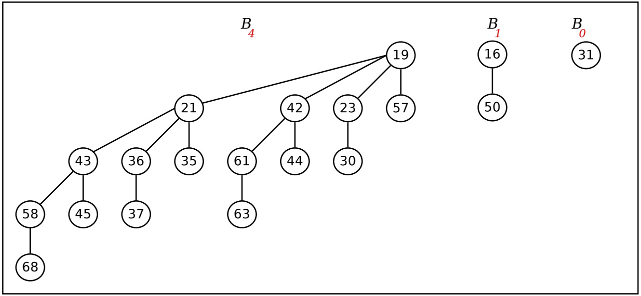
Example

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

- **one** B_4 tree
- **zero** B_3 trees
- **zero** B_2 trees
- **one** B_1 tree
- **one** B_0 tree



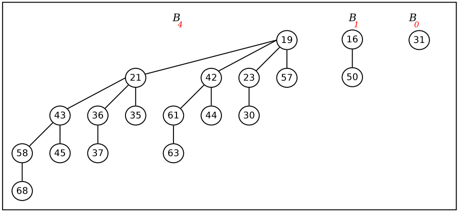
How to find which order trees are there in a Binomial heap?



For the above binomial **heap**:

- $N = 19$.
- (Minimal) Binary representation of 19 gives: $\overset{4}{1}\ \overset{3}{0}\ \overset{2}{0}\ \overset{1}{1}\ \overset{0}{1}$
- The '1's above are at bit positions 0,1 and 4
- Therefore, the binomial **heap** with $N = 19$ contains 3 binomial **trees**: B_0, B_1, B_4 trees.

Binomial heap properties

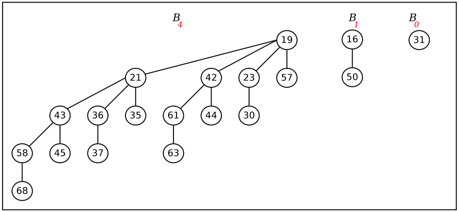


Properties

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

- The '1's in the **minimal binary representation** of N tell us which order/degree binomial **trees** form the binomial **heap** of N elements.
- 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 element with **minimum** key in the entire heap will be one of the root nodes of the **trees** in the collection.

Binomial heap properties

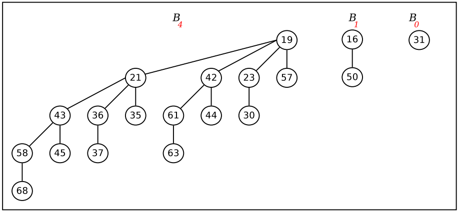


Properties

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

- The '1''s in the **minimal binary representation** of N tell us which order/degree binomial **trees** form the binomial **heap** of N elements.
- 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 element with **minimum** key in the entire heap will be one of the root nodes of the **trees** in the collection.

Binomial heap properties

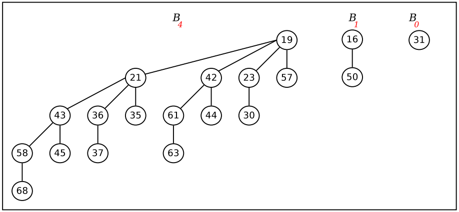


Properties

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

- The '1''s in the **minimal binary representation** of N tell us which order/degree binomial **trees** form the binomial **heap** of N elements.
- 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 element with **minimum** key in the entire heap will be one of the root nodes of the **trees** in the collection.

Binomial heap properties



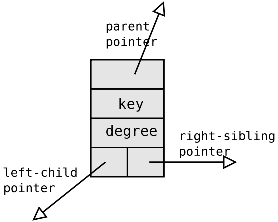
Properties

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

- The '1''s in the **minimal binary representation** of N tell us which order/degree binomial **trees** form the binomial **heap** of N elements.
- 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 element with **minimum** key in the entire heap will be one of the root nodes of the **trees** in the collection.

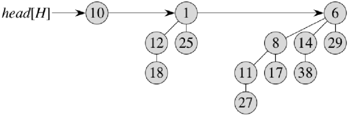
Representing a binomial heap

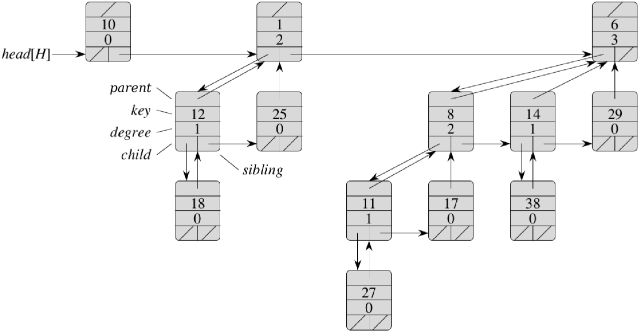
- Unlike **binary** heaps, **binomial** heaps are stored explicitly using a **tree** data structure.
- Each node x:
  - is denoted by a x.**key** (one may also include additional information as x.**payload**).
  - has a pointer x.**parent** to its parent node
  - has a pointer x.**child** to its **leftmost** child node
    - ★ If node x has zero children, then $x.\textbf{child} = nil$
  - has a pointer x.**sibling** to the immediate **sibling** of x to its right.
    - ★ If node x is the rightmost child of its parent, then $x.\textbf{sibling} = nil$
  - stores x.**degree** 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 trees within a binomial heap are organized in a linked list, referred to as the **root list**.

Binomial heap data structure representation – example



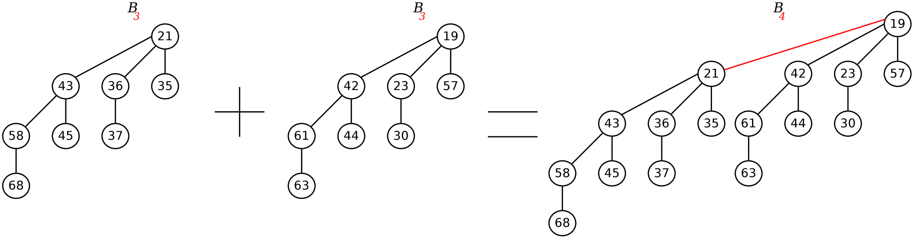


† Image from CLRS: Introduction to Algorithms

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 root with **smaller** key.



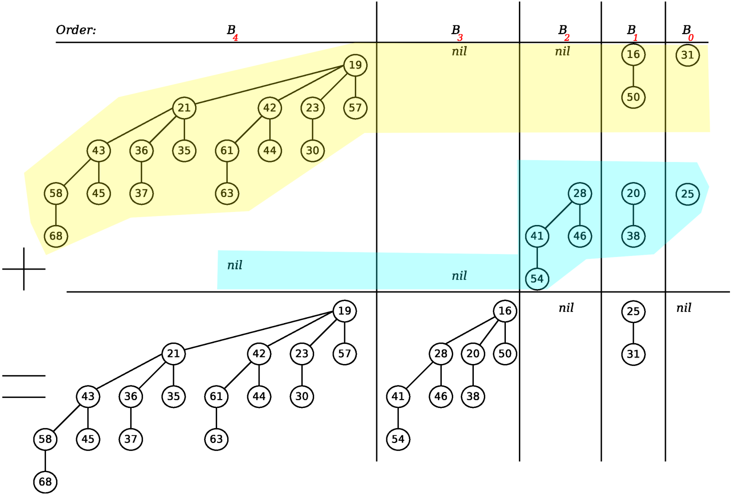
Binomial **heap** operation – **merge** two binomial **heaps** into one

- With merging of two binomial **trees** established (see previous slide), we can now define **merging** of two binomial **heaps** (each containing a collection of trees).
- 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



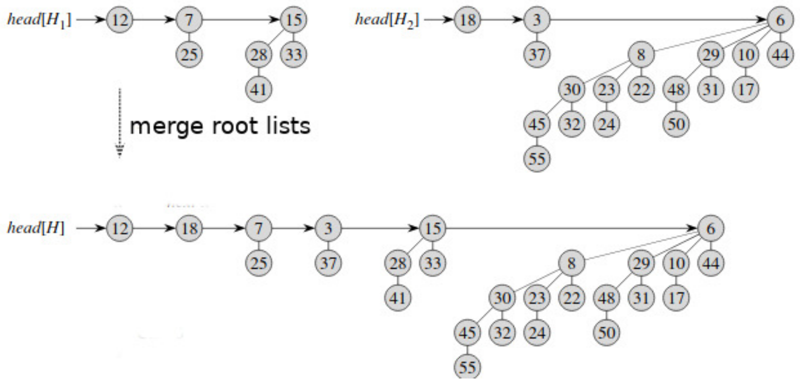
(To be discussed during the lecture)

Implementing **merge**(H_1,H_2) – first phase

Merging any two binomial heaps can be implemented in 2 phases:

First phase

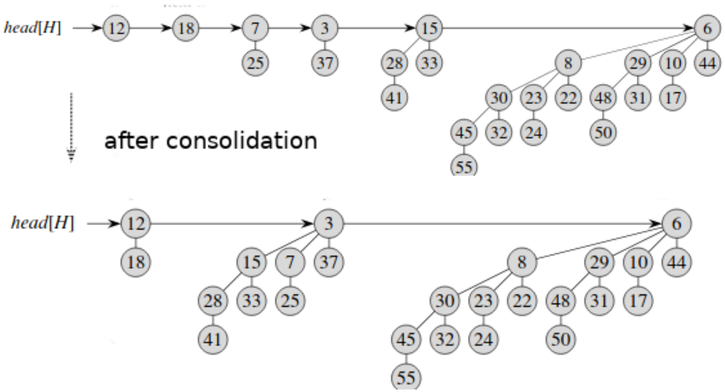
Combine the root level linked-lists (root lists) of H_1 and H_2, such that the degrees are **monotonically increasing**. Note: This potentially **violates** the binomial heap property that requires **at most** one binomial tree of any given degree – see slide #10.



Implementing **merge**(H_1, H_2) – second phase

Second phase

Scan the combined root list (from first phase) and iteratively check the degrees of successive root nodes to **consolidate** them by merging trees (with same degrees) in order to get back a valid binomial heap (with **at most** one tree of any degree).



Complexity of **merge**(H_1, H_2)

Let H_1 contain n_1 elements, and H_2 contain n_2 elements, and $N = n_1 + n_2$

- Running time is $O(\log N)$ worst-case:
  - First phase results in a root list with **at most** $\lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor + 2$ nodes.
  - In second phase, each iteration takes $O(1)$ time, with **at most** $\lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor + 2$ iterations.
  - Since: $\lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor + 2 \leq 2\lfloor \log N \rfloor + 2$
  - Thus, merging two heaps, in worst case, requires $O(\log N)$ effort.

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** node among the nodes in the root level of the heap, and delete it.
- From slide #8, we know that deleting the root node of any B_k tree yields: $B_{k-1}, B_{k-2}, \ldots, B_0$.
- Promote these subtrees to the root level of the existing binomial heap. The result is similar to the result after first phase of merge operation...
- ...with potentially multiple trees of the same degree (violating binomial heap def'n – see slide #10).
- Now consolidate this current state of the tree so that **at most** one binomial tree of any degree will be present. This is same as the second phase of **merge**.

To be discussed with an example during the lecture. Take notes!

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** node among the nodes in the root level of the heap, and delete it.
- From slide #8, we know that deleting the root node of any B_k tree yields: $B_{k-1}, B_{k-2}, \ldots, B_0$.
- Promote these subtrees to the root level of the existing binomial heap. The result is similar to the result after first phase of merge operation...
- ...with potentially multiple trees of the same degree (violating binomial heap def'n – see slide #10).
- Now consolidate this current state of the tree so that **at most** one binomial tree of any degree will be present. This is same as the second phase of **merge**.

To be discussed with an example during the lecture. Take notes!

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** node among the nodes in the root level of the heap, and delete it.
- From slide #8, we know that deleting the root node of any B_k tree yields: $B_{k-1}, B_{k-2}, \ldots, B_0$.
- Promote these subtrees to the root level of the existing binomial heap. The result is similar to the result after first phase of merge operation...
- ...with potentially multiple trees of the same degree (violating binomial heap def'n – see slide #10).
- Now consolidate this current state of the tree so that **at most** one binomial tree of any degree will be present. This is same as the second phase of **merge**.

To be discussed with an example during the lecture. Take notes!

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** node among the nodes in the root level of the heap, and delete it.
- From slide #8, we know that deleting the root node of any B_k tree yields: $B_{k-1}, B_{k-2}, \ldots, B_0$.
- Promote these subtrees to the root level of the existing binomial heap. The result is similar to the result after first phase of merge operation...
- ...with potentially multiple trees of the same degree (violating binomial heap def'n – see slide #10).
- Now consolidate this current state of the tree so that **at most** one binomial tree of any degree will be present. This is same as the second phase of **merge**.

To be discussed with an example during the lecture. Take notes!

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** node among the nodes in the root level of the heap, and delete it.
- From slide #8, we know that deleting the root node of any B_k tree yields: $B_{k-1}, B_{k-2}, \ldots, B_0$.
- Promote these subtrees to the root level of the existing binomial heap. The result is similar to the result after first phase of merge operation...
- ...with potentially multiple trees of the same degree (violating binomial heap def'n – see slide #10).
- Now consolidate this current state of the tree so that **at most** one binomial tree of any degree will be present. This is same as the second phase of **merge**.

To be discussed with an example during the lecture. Take notes!

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 #12)
  - Effort required to promote (to root level) and merge subtrees formed upon deletion of the min element is $O(\log N)$ (see slide #21)
  - Total effort: $O(\log N)$

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 #12)
  - Effort required to promote (to root level) and merge subtrees formed upon deletion of the min element is $O(\log N)$ (see slide #21)
  - Total effort: $O(\log N)$

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 $\#12$)
  - Effort required to promote (to root level) and merge subtrees formed upon deletion of the min element is $O(\log N)$ (see slide $\#21$)
  - Total effort: $O(\log N)$

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 $\#$12)
  - Effort required to promote (to root level) and merge subtrees formed upon deletion of the min element is $O(\log N)$ (see slide $\#$21)
  - Total effort: $O(\log N)$

Binomial **heap** operation – **decrease-key**

We want to decrease key of any node x in a binomial heap containing N elements. ‡

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

To be discussed with an example during the lecture. Take notes!

‡Note: as with binary heaps, binomial (and Fibonacci) heaps are inefficient to **search** for any node x (except the root); For this reason, **decrease-key**(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-key** 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 run-time 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?
- $\vdots$

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

- $T = \frac{N}{2} \times 1 + \frac{N}{4} \times 2 + \frac{N}{8} \times 3 \ldots = 2N$
- This is an instance of 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-key	$O(\log N)$	$O(\log N)$
delete	$O(\log N)$	$O(\log N)$
insert	$O(\log N)$ worst-case $O(1)$ amortized	$O(\log N)$ worst-case $O(1)$ amortized

In the next lecture...

Fibonacci heaps

-=o0o=-
END
-=o0o=-