# Hollow Heaps

Thomas Dueholm Hansen[1] Haim Kaplan[2]
Robert E. Tarjan[3] Uri Zwick[2]

[1] Department of Computer Science,
Aarhus University, Denmark.

[2] School of Computer Science,
Tel Aviv University, Israel.

[3] Department of Computer Science,
Princeton University, USA.

July 7, 2015

# Heap operations

- *make-heap*(): Make an empty heap.
- *insert*$(e, k, h)$: Insert item $e$ with key $k$ in the heap $h$.
- *find-min*$(h)$: Return the item with minimum key in $h$.
- *delete-min*$(h)$: Remove the item with minimum key from $h$.
- *decrease-key*$(e, k, h)$: Decrease the key of item $e$ in $h$ to $k$.
- *delete*$(e, h)$: Remove the item $e$ from $h$.
- *meld*$(h_1, h_2)$: Return a heap containing all items in $h_1$ and $h_2$.

# Previous results

Let $n$ be the number of items in the heap.

- **Binary heaps**: *insert*, *decrease-key*, and *delete-min* each take $O(\log n)$ time in the worst case.
- **Fibonacci heaps**, Fredman and Tarjan (1987): Optimal **amortized** time bounds.

$$\begin{array}{ll} \textit{insert}: & O(1) \\ \textit{decrease-key}: & O(1) \\ \textit{delete-min}: & O(\log n) \\ \textit{meld}: & O(1) \end{array}$$

- Brodal (1996); Brodal, Lagogiannis, and Tarjan (2012): Optimal **worst-case** bounds.
- Numerous other heap variants have been proposed.

# Applications

- **Dijkstra's algorithm** (1959) for solving single source shortest paths problems with non-negative weights.
- Algorithms for finding undirected and directed **minimum spanning trees** (Gabow, Galil, Spencer, and Tarjan; 1986).
- Etc.

# Goal

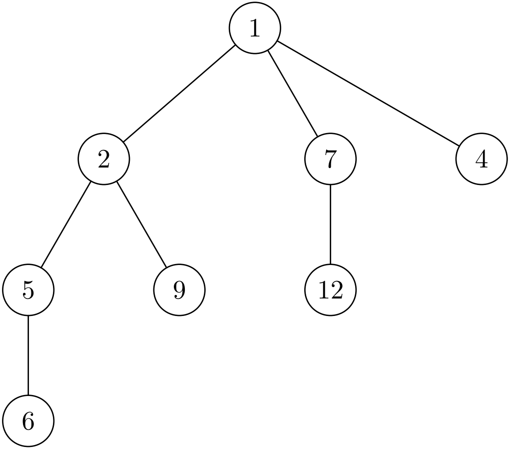
- **Goal**: Find a **simple** and **practical** data structure with optimal worst-case guarantees.
- The design space is rich, and better algorithms may yet be found.
- Fibonacci heaps get enough exposure to motivate further study.

# Goal

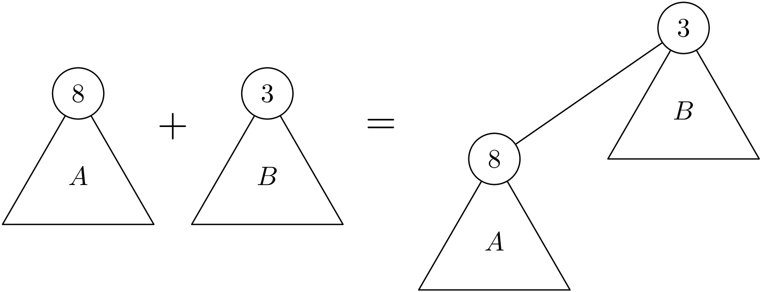
- **Goal**: Find a **simple** and **practical** data structure with optimal worst-case guarantees.
- The design space is rich, and better algorithms may yet be found.
- Fibonacci heaps get enough exposure to motivate further study.
- **Pairing heaps** (Fredman, Sedgewick, Sleator, and Tarjan; 1986) outperform Fibonacci heaps in simplicity and performance.
  - Fredman (1999): *decrease-key* requires $\Omega(\log \log n)$ amortized time.
  - Pettie (2005): $O(2^{2\sqrt{\log \log n}})$ upper bound for *decrease-key*.

# Heap-order

- The key of a node is at least as large as the key of its parent.



# Linking trees while preserving heap-order



# Characteristics of Fibonacci heaps

- A **Fibonacci heap** is a set $S$ of heap-ordered trees.
  - It maintains a pointer to the node with minimum key.

# Characteristics of Fibonacci heaps

- A **Fibonacci heap** is a set $S$ of heap-ordered trees.
  - It maintains a pointer to the node with minimum key.
- Each node has a **rank** that matches its degree, and nodes are possibly **marked**.
- *decrease-key* may trigger **cascading cuts: Mark** the first unmarked ancestor, and cut all ancestors in between.
  - **4 pointers per node**: Parent, child, left sibling, and right sibling.

# Characteristics of Fibonacci heaps

- A **Fibonacci heap** is a set $S$ of heap-ordered trees.
  - It maintains a pointer to the node with minimum key.
- Each node has a **rank** that matches its degree, and nodes are possibly **marked**.
- *decrease-key* may trigger **cascading cuts: Mark** the first unmarked ancestor, and cut all ancestors in between.
  - **4 pointers per node**: Parent, child, left sibling, and right sibling.
- The amortized time per *delete-min* is bounded by the maximum rank.
  - By induction, the number of nodes in a tree of rank $k$ is at least $F_{k+1} = 1 + F_1 + F_2 + \cdots + F_{k-1} = O(1.618^k)$.
  - The maximum rank is at most $\log_\varphi n = O(\log n)$.

# Results

We introduce **Hollow heaps**, a simple alternative to **Fibonacci heaps**. The distinguishing characteristics of hollow heaps are:

- Each heap is a single heap-ordered **directed acyclic graph** (DAG) that represents all comparisons explicitly.

# Results

We introduce **Hollow heaps**, a simple alternative to **Fibonacci heaps**. The distinguishing characteristics of hollow heaps are:

- Each heap is a single heap-ordered **directed acyclic graph** (DAG) that represents all comparisons explicitly.
- There are **no parent pointers**, and *decrease-key* takes $O(1)$ time worst-case and does no cascading cuts.

# Results

We introduce **Hollow heaps**, a simple alternative to **Fibonacci heaps**. The distinguishing characteristics of hollow heaps are:

- Each heap is a single heap-ordered **directed acyclic graph** (DAG) that represents all comparisons explicitly.
- There are **no parent pointers**, and *decrease-key* takes $O(1)$ time worst-case and does no cascading cuts.
- Children are stored in **singly linked lists**.

# Results

We introduce **Hollow heaps**, a simple alternative to **Fibonacci heaps**. The distinguishing characteristics of hollow heaps are:

- Each heap is a single heap-ordered **directed acyclic graph** (DAG) that represents all comparisons explicitly.
- There are **no parent pointers**, and *decrease-key* takes $O(1)$ time worst-case and does no cascading cuts.
- Children are stored in **singly linked lists**.
- The heap contains **hollow nodes**, and *delete-min* takes $O(\log N)$ amortized time, where $N$ is the total number of nodes.

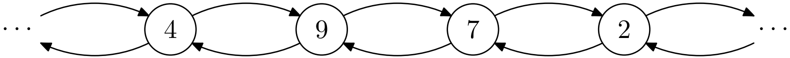
# Results

We introduce **Hollow heaps**, a simple alternative to **Fibonacci heaps**. The distinguishing characteristics of hollow heaps are:

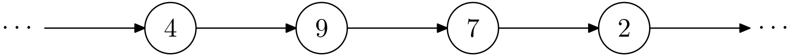
- Each heap is a single heap-ordered **directed acyclic graph** (DAG) that represents all comparisons explicitly.
- There are **no parent pointers**, and *decrease-key* takes $O(1)$ time worst-case and does no cascading cuts.
- Children are stored in **singly linked lists**.
- The heap contains **hollow nodes**, and *delete-min* takes $O(\log N)$ amortized time, where $N$ is the total number of nodes.
- The structure is **exogenous** rather than endogenous: nodes hold items rather than being items.

# Using hollow nodes: deletion in linked lists

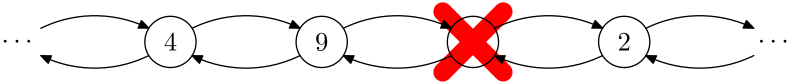
Doubly linked list:



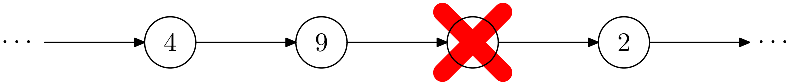

Singly linked list:



Doubly linked list:

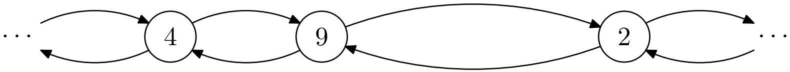



Singly linked list:

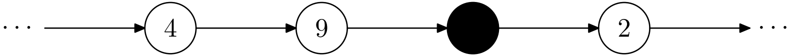


# Using hollow nodes: deletion in linked lists

Doubly linked list:



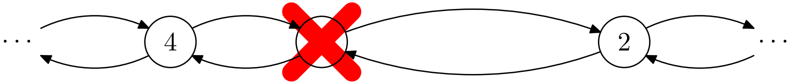

Singly linked list:



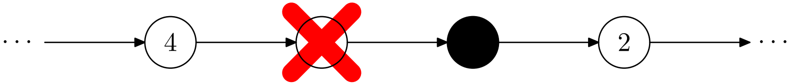

- Deletions in a singly linked list create "hollow" nodes.

# Using hollow nodes: deletion in linked lists

Doubly linked list:



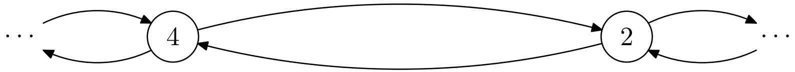

Singly linked list:



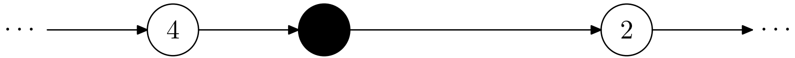

- Deletions in a singly linked list create “hollow” nodes.

# Using hollow nodes: deletion in linked lists

Doubly linked list:




Singly linked list:




- Deletions in a singly linked list create "hollow" nodes.
- They can only be removed from nodes pointing to them.

# Naive hollow heaps

The heap is referenced by its root.

- *meld*$(h_1, h_2)$: Link $h_1$ and $h_2$.
- *insert*$(e, k, h)$: Create a node $x$ with key $k$ and item $e$. Link $x$ and $h$.
- *find-min*$(h)$: Return the item of $h$.
- *decrease-key*$(e, k, h)$: Let $x$ be the node containing $e$. Move $e$ to a new node $y$ with key $k$, making $x$ **hollow**. Link $y$ and $h$.
- *delete-min*$(h)$: Delete $h$ and all its immediate hollow descendants. Link the exposed full nodes.

# Naive hollow heaps

The heap is referenced by its root.

- *meld*$(h_1, h_2)$: Link $h_1$ and $h_2$.
- *insert*$(e, k, h)$: Create a node $x$ with key $k$ and item $e$. Link $x$ and $h$.
- *find-min*$(h)$: Return the item of $h$.
- *decrease-key*$(e, k, h)$: Let $x$ be the node containing $e$. Move $e$ to a new node $y$ with key $k$, making $x$ **hollow**. Link $y$ and $h$.
- *delete-min*$(h)$: Delete $h$ and all its immediate hollow descendants. Link the exposed full nodes.

**Note:** This version is inefficient.

# Making hollow heaps efficient

- Assign an integer **rank** to every node.
- Initially nodes have rank 0.
- **Ranked link:** Link two nodes of the same rank and increase the rank of the winner by 1.
- **Unranked link:** Link two nodes without changing their ranks.

# Making hollow heaps efficient

- Assign an integer **rank** to every node.
- Initially nodes have rank 0.
- **Ranked link:** Link two nodes of the same rank and increase the rank of the winner by 1.
- **Unranked link:** Link two nodes without changing their ranks.

- We allow hollow nodes to have up to two parents.
- **Heap order:** The key of a node is at least as large as the keys of both its parents.

# Hollow heaps

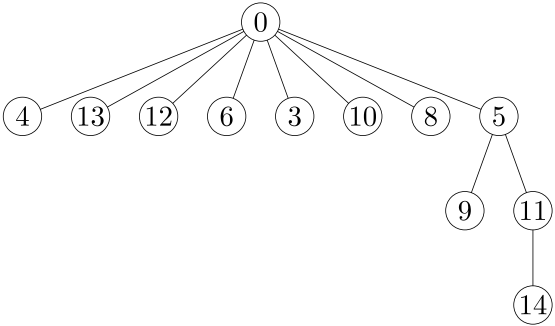
- *meld*$(h_1, h_2)$: Link $h_1$ and $h_2$.
- *insert*$(e, k, h)$: Create a node $x$ with key $k$, item $e$, and rank $x.rank = 0$. Link $x$ and $h$.
- *find-min*$(h)$: Return the item of $h$.
- *decrease-key*$(e, k, h)$:
  - Let $x$ be the node containing $e$.
  - Move $e$ to a new node $y$ with key $k$ and rank $y.rank = \max\{0, x.rank - 2\}$.
  - Make $x$ **hollow** and make $y$ the **second parent** of $x$.
  - Link $y$ and $h$.

# Hollow heaps

- *meld*$(h_1, h_2)$: Link $h_1$ and $h_2$.
- *insert*$(e, k, h)$: Create a node $x$ with key $k$, item $e$, and rank $x.rank = 0$. Link $x$ and $h$.
- *find-min*$(h)$: Return the item of $h$.
- *decrease-key*$(e, k, h)$:
  - Let $x$ be the node containing $e$.
  - Move $e$ to a new node $y$ with key $k$ and rank $y.rank = \max\{0, x.rank - 2\}$.
  - Make $x$ **hollow** and make $y$ the **second parent** of $x$.
  - Link $y$ and $h$.
- *delete-min*$(h)$:
  - Delete $h$ and all its immediate hollow descendants **that only have one parent**.
  - While possible, link the exposed full nodes with **ranked links**.
  - Link the remaining nodes with **unranked links**.

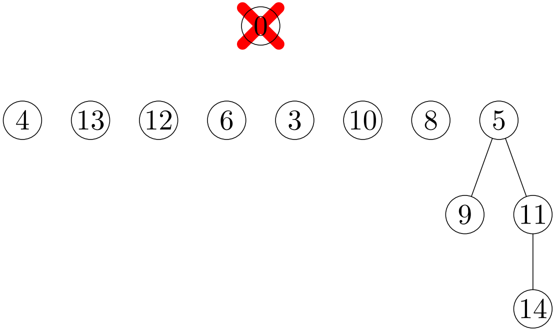
# Example

- The heap after successive insertions of items with keys 14, 11, 5, 9, 0, 8, 10, 3, 6, 12, 13, 4.



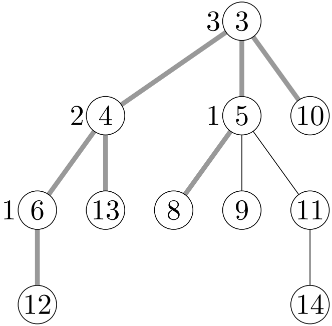
# Example

- The heap after successive insertions of items with keys 14, 11, 5, 9, 0, 8, 10, 3, 6, 12, 13, 4.



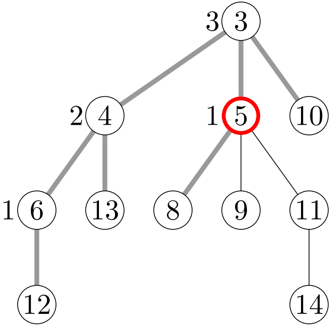

- **Next:** *delete-min*

- After *delete-min*.



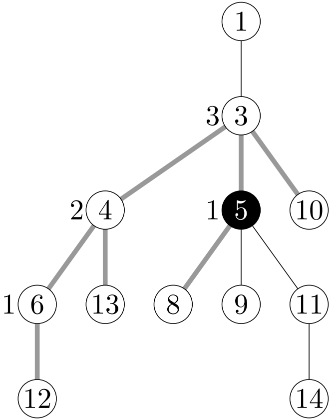
# Example

- After *delete-min*.



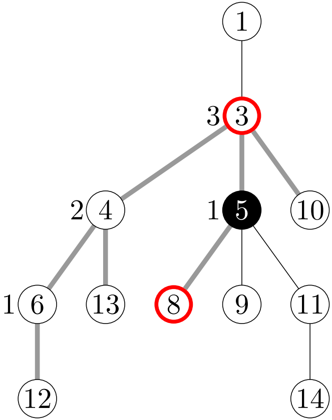

- **Next:** Decrease 5 to 1.

- After decreasing 5 to 1.



# Example

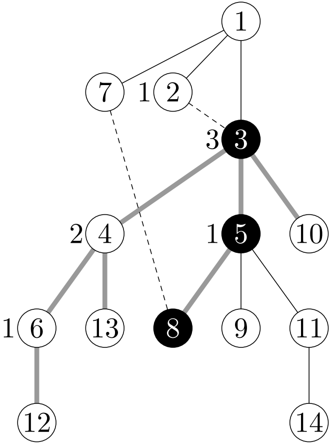
- After decreasing 5 to 1.




- **Next:** Decrease 3 to 2 and 8 to 7.

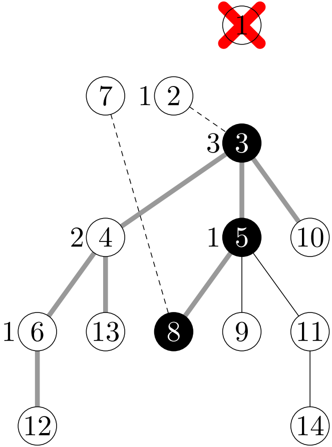
# Example

- After decreasing 3 to 2 and 8 to 7.



# Example

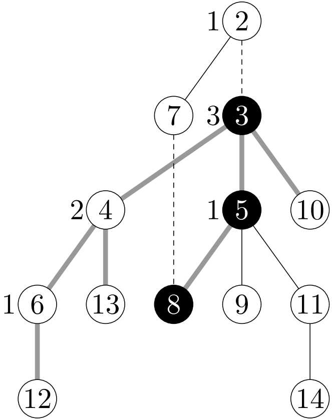
- After decreasing 3 to 2 and 8 to 7.



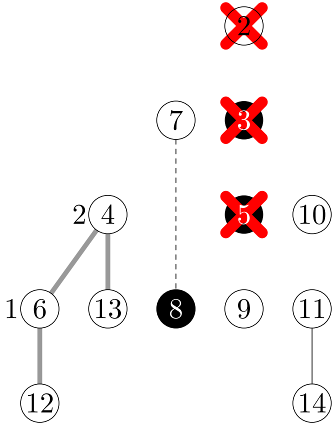

- **Next:** *delete-min*

# Example

- After *delete-min*.

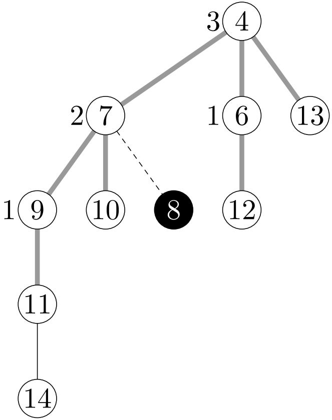


- After *delete-min*.




- **Next:** *delete-min*

- After *delete-min*.



# Results

- Let $N$ be the number of nodes.
- We prove these **amortized** time bounds for hollow heaps:

| | |
|---|---|
| *insert* : | $O(1)$ |
| *decrease-key* : | $O(1)$ |
| *delete-min* : | $O(\log N)$ |
| *meld* : | $O(1)$ |

- $N$ is at most the number of items $n$ plus the number of performed *decrease-key* operations.
- In many applications $N = O(poly(n))$, and otherwise the bound can be improved by occasionally **rebuilding** the heap.

# Eager hollow heaps

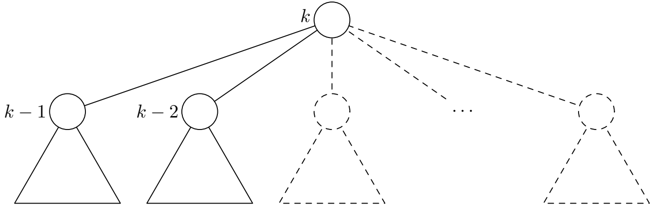
- **Eager implementation:** No second parents. Nodes are moved immediately by *decrease-key*.
- *decrease-key*$(e, k, h)$:
  - Let $x$ be the node containing $e$.
  - Move $e$ to a new node $y$ with key $k$ and rank $y.rank = \max\{0, x.rank - 2\}$, making $x$ **hollow**.
  - Move all but the **two highest ranked children** of $x$ to $y$.
  - Link $y$ and $h$.
- *delete-min*$(h)$:
  - Delete $h$ and all its immediate hollow descendants.
  - While possible, link the exposed full nodes with **ranked links**.
  - Link the remaining nodes with **unranked links**.

# Bounding the maximum rank

- A **full** node of rank $k$ has $k$ ranked children with ranks $0, \dots, k-1$.
- A **hollow** node of rank $k$ has 2 ranked children of ranks $k-1$ and $k-2$ (or fewer if $k \leq 1$).

# Bounding the maximum rank

- A **full** node of rank $k$ has $k$ ranked children with ranks $0, \ldots, k-1$.
- A **hollow** node of rank $k$ has 2 ranked children of ranks $k-1$ and $k-2$ (or fewer if $k \leq 1$).
- By induction, the number of nodes in a tree of rank $k$ is at least $F_{k+3} - 1 = 1 + (F_{k+2} - 1) + (F_{k+1} - 1)$.




- The maximum rank is at most $\log_\varphi N = O(\log N)$.

# Analysis

- **Potential argument**: All **unranked children** and **children of hollow nodes** have 1 unit of potential each.
- The potential pays for **linking** and for **deleting hollow nodes**.
- *insert*: Create 1 new unranked child $(+1)$.
- *decrease-key*: Create 1 new unranked child and 2 new children of a hollow node $(+3)$.
- *delete-min*: Create at most $O(\log N)$ new unranked children.

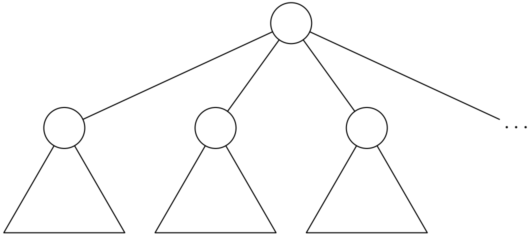
# Analysis

- **Potential argument**: All **unranked children** and **children of hollow nodes** have 1 unit of potential each.
- The potential pays for **linking** and for **deleting hollow nodes**.
- *insert*: Create 1 new unranked child $(+1)$.
- *decrease-key*: Create 1 new unranked child and 2 new children of a hollow node $(+3)$.
- *delete-min*: Create at most $O(\log N)$ new unranked children.

- The analysis of **lazy hollow heaps** is essentially the same.
- In this case nodes are assigned **virtual children** corresponding to actual children in the eager version.

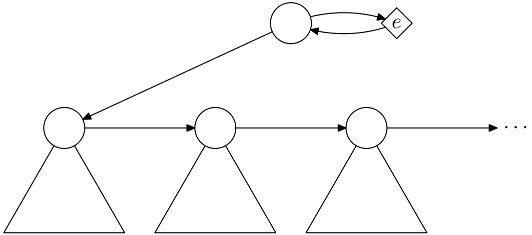
# Lazy hollow heap, implementation

- **3 pointers per node**: Child, right sibling, and second parent (can be replaced by a single bit).
- **2 additional pointers to and from items**: Nodes must hold items rather than be items; the structure is **exogenous** rather than **endogenous**.



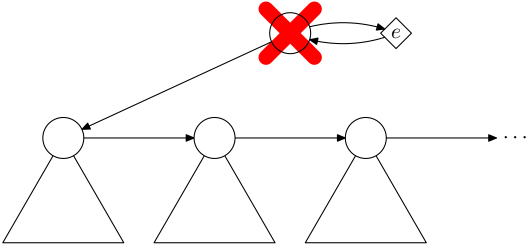
# Lazy hollow heap, implementation

- **3 pointers per node**: Child, right sibling, and second parent (can be replaced by a single bit).
- **2 additional pointers to and from items**: Nodes must hold items rather than be items; the structure is **exogenous** rather than **endogenous**.

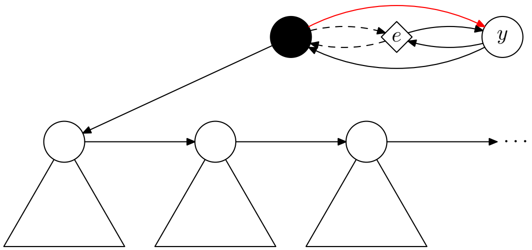


# Lazy hollow heap, implementation

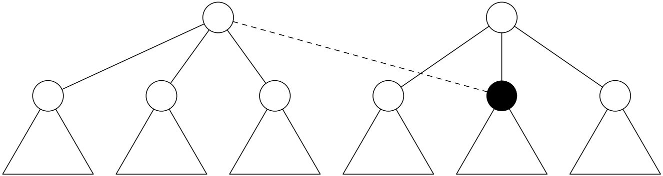
- **3 pointers per node**: Child, right sibling, and second parent (can be replaced by a single bit).
- **2 additional pointers to and from items**: Nodes must hold items rather than be items; the structure is **exogenous** rather than **endogenous**.



- **3 pointers per node**: Child, right sibling, and second parent (can be replaced by a single bit).
- **2 additional pointers to and from items**: Nodes must hold items rather than be items; the structure is **exogenous** rather than **endogenous**.

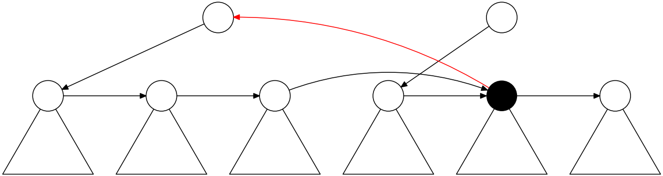


# Lazy hollow heap, implementation



- **Invariant:** A hollow node is the first child of its second parent.

# Lazy hollow heap, implementation



- **Invariant:** A hollow node is the first child of its second parent.
- When traversing a list of children, we reach the end when either:
  - There is no next sibling.
  - The second parent pointer points back to the parent whose children we are traversing.
- Second parent pointers also show whether nodes have 1 or 2 parents.

# Future work

- An **experimental study** of hollow heaps.
- Non-cascading Fibonacci heap without hollow nodes?
- **Pairing heaps** (Fredman, Sedgewick, Sleator, and Tarjan; 1986) outperform Fibonacci heaps in practice, although *decrease-key* requires $\Omega(\log \log n)$ time.
- What can be said about **hollow pairing heaps**?
- Array-based heaps outperform pointer-based heaps in practice, but the worst-case bounds are non-optimal. Can we get the best of both worlds?

# Future work

- An **experimental study** of hollow heaps.
- Non-cascading Fibonacci heap without hollow nodes?
- **Pairing heaps** (Fredman, Sedgewick, Sleator, and Tarjan; 1986) outperform Fibonacci heaps in practice, although *decrease-key* requires $\Omega(\log \log n)$ time.
- What can be said about **hollow pairing heaps**?
- Array-based heaps outperform pointer-based heaps in practice, but the worst-case bounds are non-optimal. Can we get the best of both worlds?

**Thank you for listening!**

# Initial experiments

- Larkin, Sen, and Tarjan (2014) made an **experimental study** of various heaps.
- We used their framework to make some initial tests of hollow heaps with randomly generated benchmarks.
- 2,000,000 insertions and deletions, and 5,000,000 *decrease-key* operations.
  - Fibonacci heap: 2488270 ms
  - Pairing heap: 1796165 ms
  - Lazy hollow heap: 1940643 ms