CS 225 – Data Structures

ZJUI – Spring 2022

Lecture 5: Priority Queues

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

5 Priority Queues

5.1 Binary Heaps and Heapsort

An **almost complete tree** is a binary tree, in which all leaves have depth d or $d-1$ and there is at most one non-leaf node of depth $d-1$ with only one child node and all left siblings of this node have two children, and all right siblings have no children

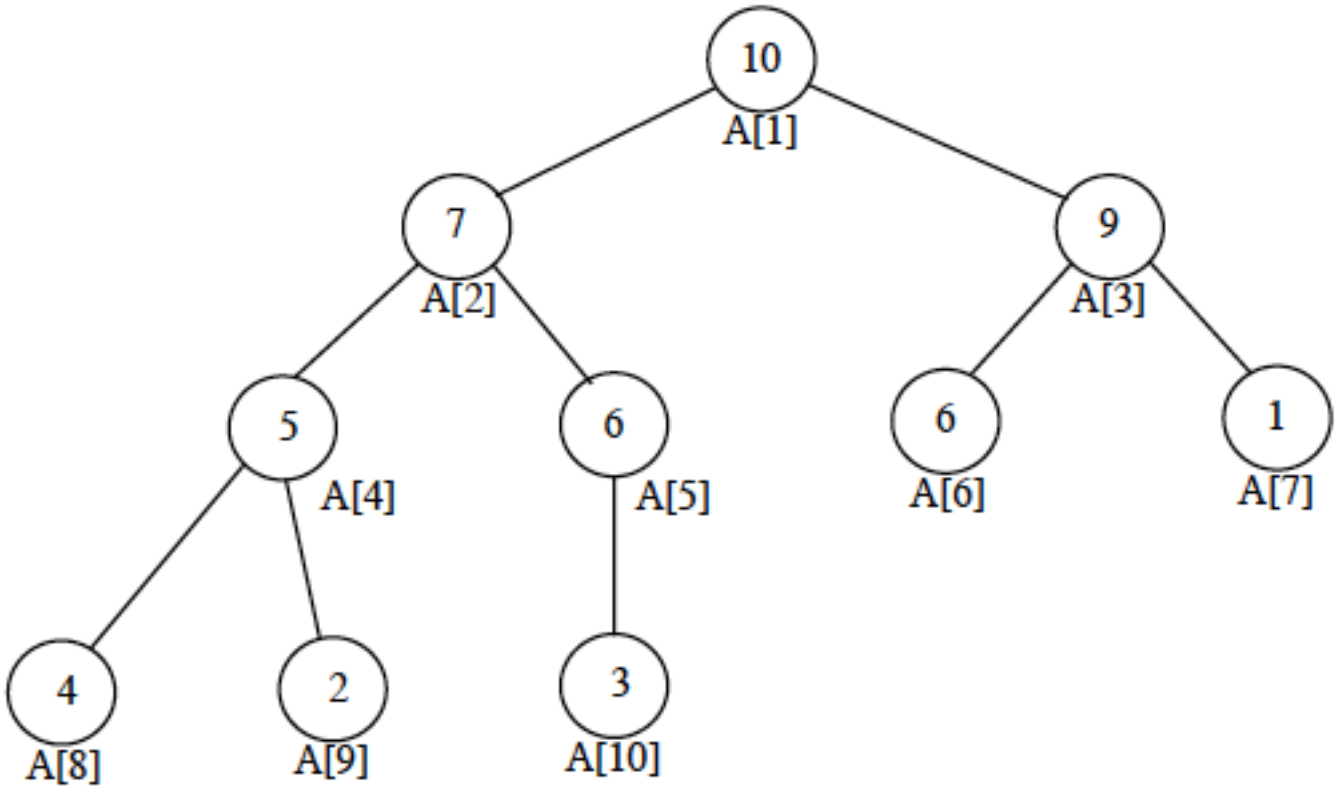
A **max-heap** is a node-labelled almost complete tree such that for each nodes v and each of its children v' we have $\ell(v) \geq \ell(v')$

Analogously, a **min-heap** is a node-labelled almost complete tree such that for each nodes v and each of its children v' we have $\ell(v) \leq \ell(v')$

Heaps with n nodes are represented by arrays $A[1, \ldots, n]$ such that $A[1]$ contains the label of the root, and when $A[i]$ contains $\ell(v)$, then $A[2i]$ and $A[2i+1]$ contain the labels of the children of v (provided the indices are $\leq n$)

Example

The following shows a max-heap and its representing array



$$A = [10, 7, 9, 5, 6, 6, 1, 4, 2, 3]$$

Building a Max-Heap

Start from an arbitrary array $A[1, \ldots, n]$ (representing an almost complete node-labelled binary tree), then use an operation ***sift-down*** on $A[\lfloor n/2 \rfloor], \ldots, A[1]$ (i.e. on all non-leaf nodes)

sift-down swaps an element $A[i]$ with the larger one of $A[2i]$, $A[2i+1]$, provided this element is $\geq A[i]$

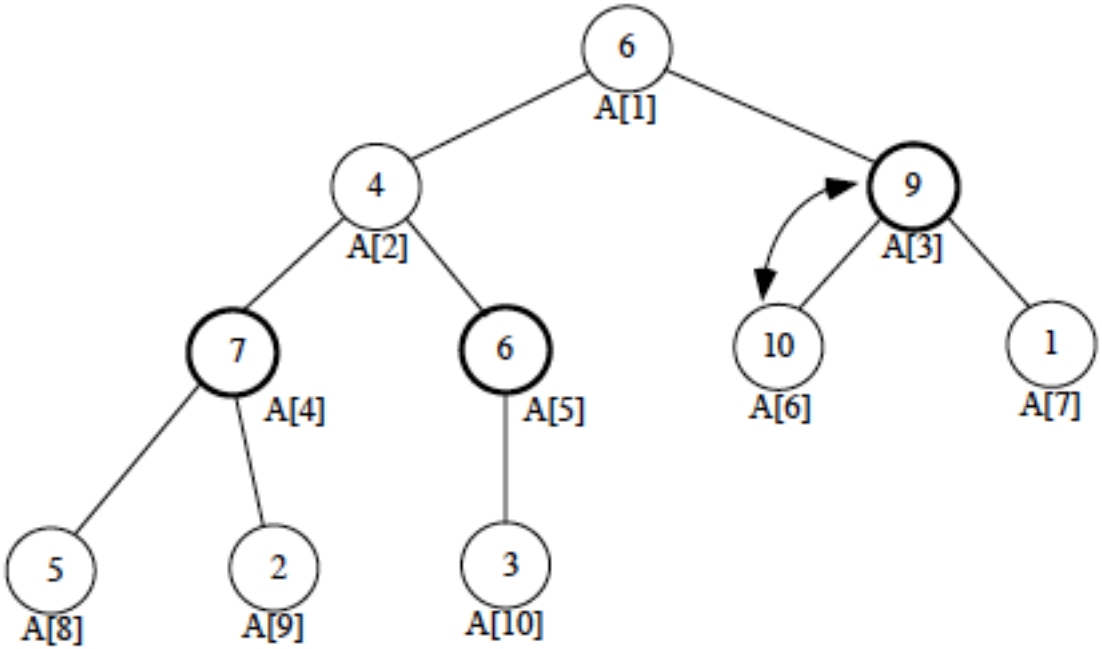
This is iterated for the child, the label of which was swapped

As non-leaves are processed in a bottom-up way, we can easily show by induction that the result is a max-heap

The building of a min-heap is done analogously

Example

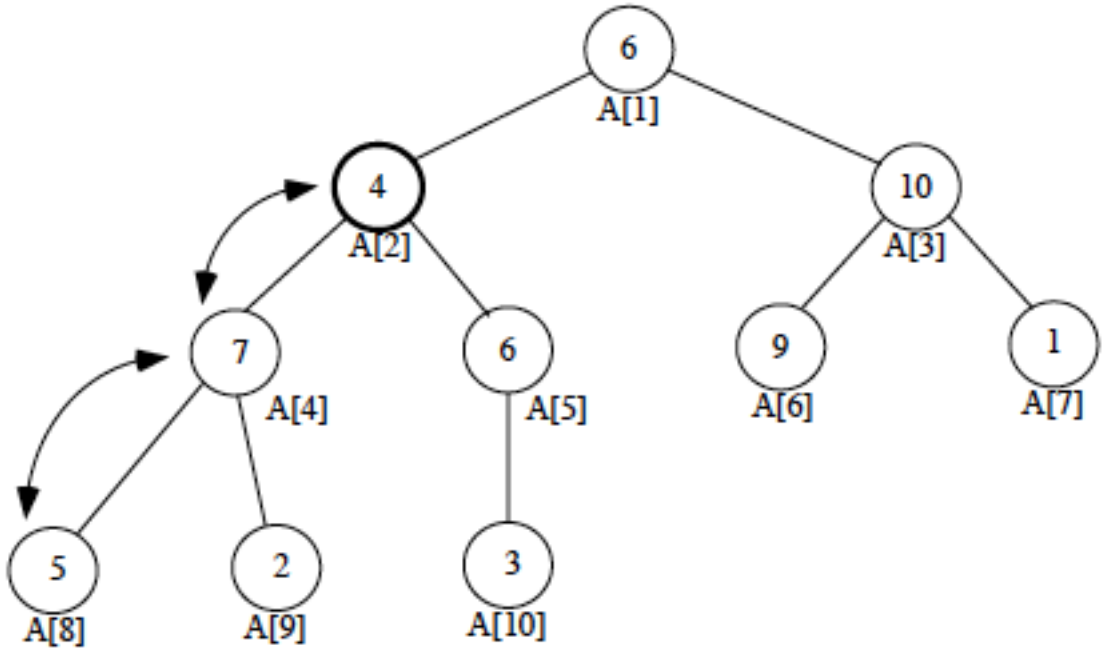
We illustrate the building of a max-heap for the array $A = [6, 4, 9, 7, 6, 10, 1, 5, 2, 3]$



The ***sift-down*** for $A[5] = 6$ and $A[4] = 7$ makes no change, the ***sift-down*** for $A[3] = 9$ swaps the 9 with the 10

Example / cont.

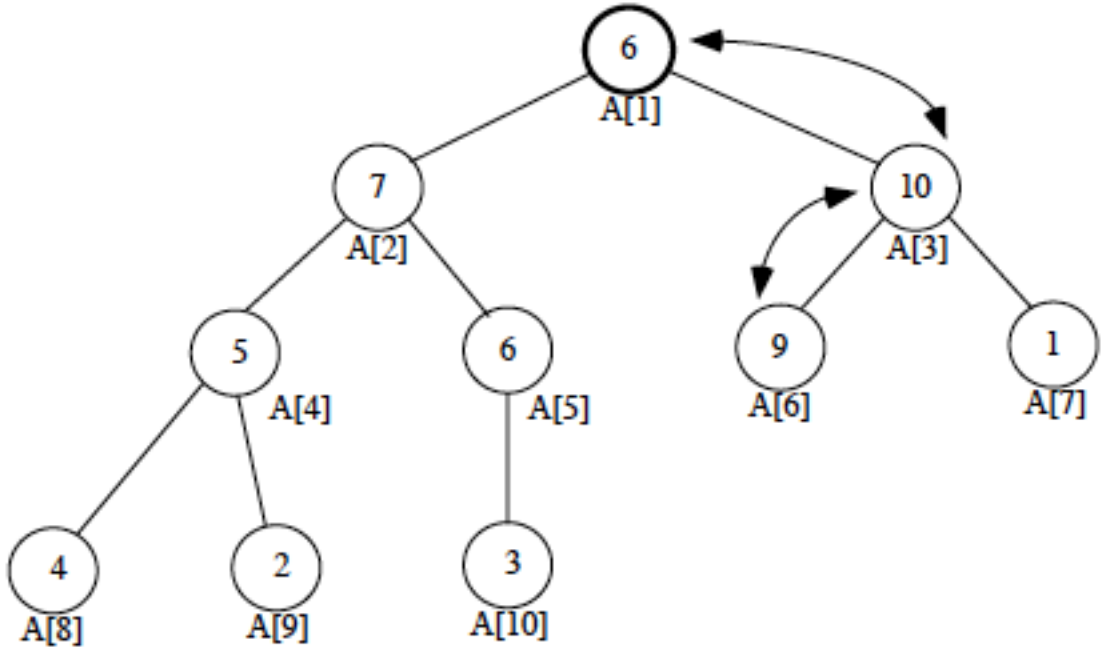
Then we have the array $A = [6, 4, 10, 7, 6, 9, 1, 5, 2, 3]$



The ***sift-down*** for $A[2] = 4$ requires first to swap 4 with 7, then 4 with 5

Example / cont.

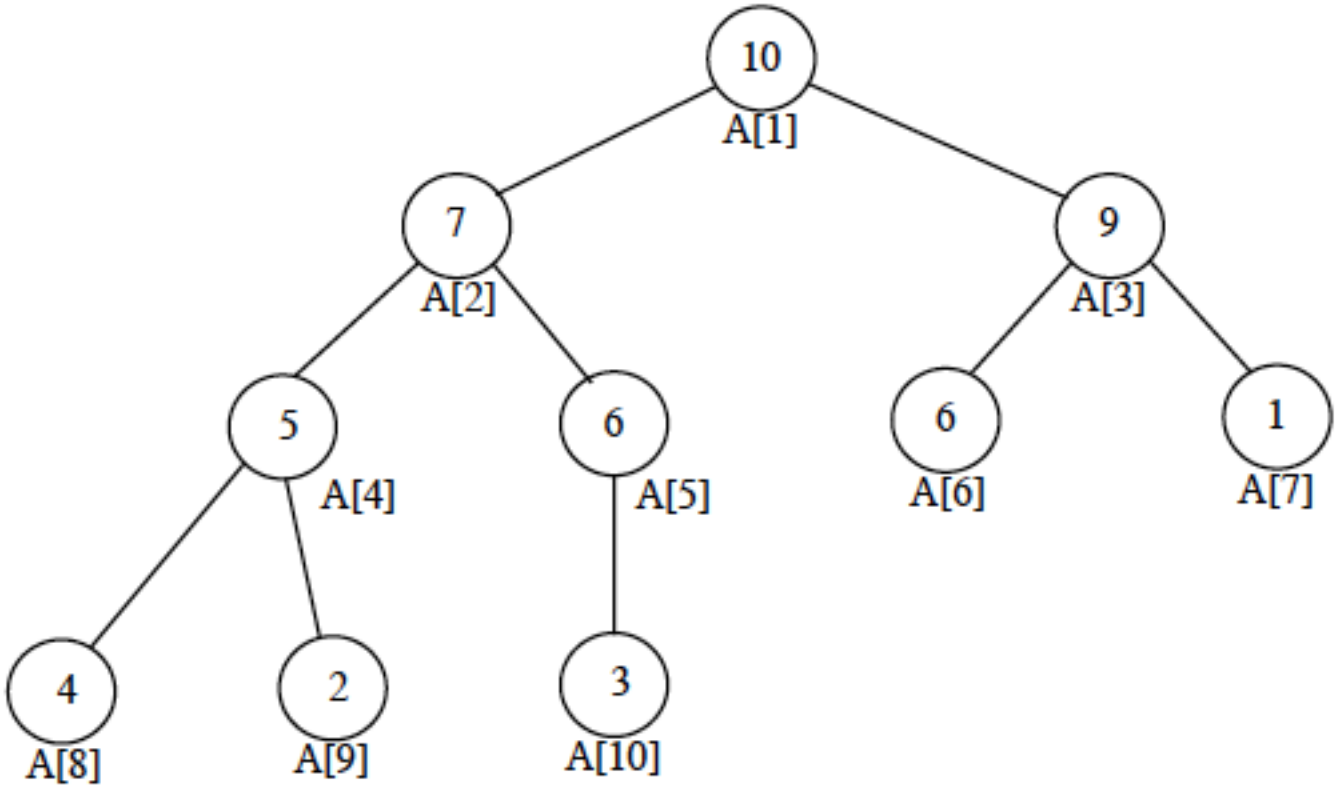
Then we have the array $A = [6, 7, 10, 5, 6, 9, 1, 4, 2, 3]$



Finally, the ***sift-down*** for $A[1] = 6$ requires first to swap 6 with 10, then 6 with 9

Example / cont.

The final result is the max-heap represented by $A = [10, 7, 9, 5, 6, 6, 1, 4, 2, 3]$



$$A = [10, 7, 9, 5, 6, 6, 1, 4, 2, 3]$$

Insertion into a Heap

Suppose to be given a max-heap represented by an array $A[1, \ldots, n]$ and a new element x to be inserted into the heap

A naive way of insertion is to extend the array to $A[1, \ldots, n+1]$ with $A[n+1] = x$ and to apply the operation ***build_heap*** discussed above

However, this would examine many nodes of the tree that are not affected by the insertion

Therefore, apply an operation ***sift-up*** instead: swap the new element with its parent, if the heap-property is violated, then continue with grandparent, grand-grandparent, etc.

Example. If in our previous example we insert $A[11] = 8$, we have to swap 8 first with the parent $A[5] = 6$, then with the grand-parent $A[2] = 7$, so the final result will be $A = [10, 8, 9, 5, 7, 6, 1, 4, 2, 3, 6]$

Deletion from a Heap

Now consider an operation *delete_max* that removes the largest element from a max-heap

Clearly, the maximum element is $A[1]$, so the problem reduces to restoring the max-heap property after the deletion of $A[1]$

For this move $A[n]$ into the first position and apply ***sift-down*** to restore the heap property

Example. When applied to our previous example, we delete $A[1] = 10$, then we move $A[10] = 3$ into the first position, which gives the array $A = [3, 7, 9, 5, 6, 6, 1, 4, 2]$

Then swap $A[1] = 3$ with $A[3] = 9$ and further swap $A[3] = 3$ with $A[6] = 6$ to restore the max-heap property

The result is the heap represented by $A = [9, 7, 6, 5, 6, 3, 1, 4, 2]$

Heapsort

Heapsort is an efficient in-place sorting algorithm that exploits a max-heap

Given an array A, first turn it into a max-heap as described above

Then let i run from n down to 2 each time swapping $A[1]$ with $A[i]$ followed by a ***sift-down*** of $A[1]$ within the restricted array $A[1, \ldots, i-1]$

As a heap is almost balanced, ***sift-down*** has a time complexity in $\Theta(\log n)$

Consequently, building a heap and heapsort have time-complexity in $\Theta(n \log n)$ (in the worst and the average case)

Example

Take the array $A = [6, 4, 9, 7, 6, 10, 1, 5, 2, 3]$, which (as we have seen) is turned into the max-heap $A = [10, 7, 9, 5, 6, 6, 1, 4, 2, 3]$

Then the first steps of heapsort produce the following changes to the array

On the left we have the array after swapping the first element, on the right the array after sift-down

$$A = [\mathbf{3}, 7, 9, 5, 6, 6, 1, 4, 2, \mathbf{10}] \qquad A = [\mathbf{9}, 7, 6, 5, 6, 3, 1, 4, 2, \mathbf{10}]$$
$$A = [\mathbf{2}, 7, 6, 5, 6, 3, 1, 4, \mathbf{9}, \mathbf{10}] \qquad A = [7, 6, 6, 5, 2, 3, 1, 4, \mathbf{9}, \mathbf{10}]$$
$$A = [\mathbf{4}, 6, 6, 5, 2, 3, 1, \mathbf{7}, \mathbf{9}, \mathbf{10}] \qquad A = [6, 5, 6, 4, 2, 3, 1, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$
$$A = [\mathbf{1}, 5, 6, 4, 2, 3, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}] \qquad A = [6, 5, 3, 4, 2, 1, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$

Example / cont.

We continue the swapping with follow-on sift-down until we reach $A = [1, 2, 3, 4, 5, 6, 6, 7, 9, 10]$, i.e. we have accomplished the sorting

$$A = [1, 5, 3, 4, 2, \mathbf{6}, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$
$$A = [2, 4, 3, 1, \mathbf{5}, \mathbf{6}, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$
$$A = [1, 4, 3, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$
$$A = [1, 2, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$
$$A = [1, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$

$$A = [5, 4, 3, 1, 2, \mathbf{6}, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$
$$A = [4, 2, 3, 1, \mathbf{5}, \mathbf{6}, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$
$$A = [3, 2, 1, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$
$$A = [2, 1, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$
$$A = [1, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{6}, \mathbf{7}, \mathbf{9}, \mathbf{10}]$$

Correctness of Heapsort

If we have a node-labelled binary tree, in which only the root violates the max-heap property, then sift-down of the root creates a max-heap

This is because sift-down swaps in each steps one element with the larger of its children, and thus, only the subtree rooted at that child may violate the max-heap property after the swapping

When heapsort swaps $A[1]$ with $A[i]$, the element $A[i]$ is the largest in $A[1, \ldots, i]$

After the swap only $A[1]$ violates the max-heap property in $A[1, \ldots, i-1]$, which is restored by the sift-down

As always the largest not yet considered element is swapped to the end of the sublist, the final result is ordered

5.2 Adressable Priority Queues

Note that the building of heaps and the insertion of new elements applies in the same way to min-heaps

We first emphasised max-heaps, because of their usage in heapsort—the origin of heaps

When considering heaps, these are actually representations of priority queues—i.e. we insert elements arbitrarily (as above), but always retrieve (and delete) the element with highest priority

Thus, in a min-heap we are usually interested in retrieving (and deleting) the minimum element (corresponding to the element in a queue with highest priority), while insertion is done as described above

Analogously, in a max-heap we would be interested in retrieving (and deleting) the maximum element as described above

Application Examples

Assume a company that assigns orders to subcontractors—always assign an order to subcontractor who (in the current business year) has so far received the minimum number of orders

For this a min-heap with operations *build_heap*, *insert*, *retrieve_min* and *delete_min* will be sufficient—when an order is assigned to the contractor with highest priority (i.e. minimum number of orders), the contractor (the minimum) is deleted from the heap and re-inserted with an updated number of orders

A slightly more complicated example arises, if the subcontractor with the smallest number of unfinished orders is selected

In this case a contractor has to be a accessed in order to update the number of unfinished orders each time an order is finished

This leads us to **addressable priority queues**

Adressable Priority Queues: Operations

For an **addressable priority queue** we need an aditional **handle** for each element in the queue (in addition to the **key** value defining the priority)

An *insert* (k) into an addressable priority queue is the same as an insert into a min- or max-heap, but returns a handle

We provide an operation *delete*(h), which deletes the element with handle h from the queue

An operation *decrease*(h, k) updates the key value of the element with handle h to k—clearly k must be smaller then the current key value

The *decrease* operation combines a deletion with a re-insertion as required in our example

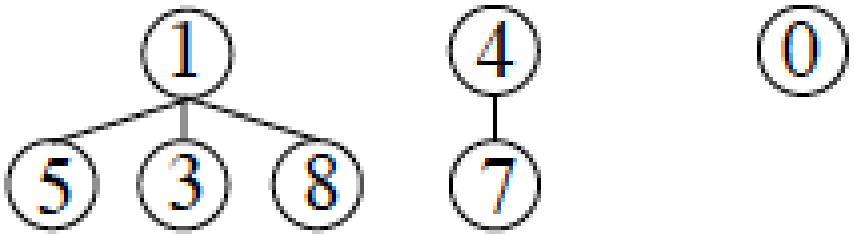
For the implementation of these operations binary heaps are not flexible enough

Heap-Ordered Forests

Instead of a single binary min-heap use a forest, where each tree is **heap-ordered**, that is the key value of any node is $\geq$ the key value of its parent

Apart from this there is no other restriction on the trees with respect to balance or order

Example



In addition use a min-pointer to the root of the tree containing the element with minimum key value

Operations on Heap-Ordered Forests

We require just three basic operations on heap-ordered forests:

- **add** a new tree to the forest and update the min-pointer
- **combine** two trees making the root of the tree with smaller key value to the parent of the root of the other tree
- **cut** out a subtree and turn it into a tree of the forest

A **rebalance** operation takes a sequence of trees and combines the first and second, third and fourth, etc. trees using the **combine** operation

Realisation of Adressable Priority Queue Operations

The operation *insert* (k) simple adds a new tree with a single node k to the forest using **add**

The operation *delete_min* removes the root of the tree indicated by the min-pointer; all trees rooted at the children become new trees of the forest, to which **rebalance** is applied

The operation $decrease(h, k)$ changes the value of the tree node indicated by h to k; as this may destroy the heap-order, use **cut** to cut out the subtree rooted at h

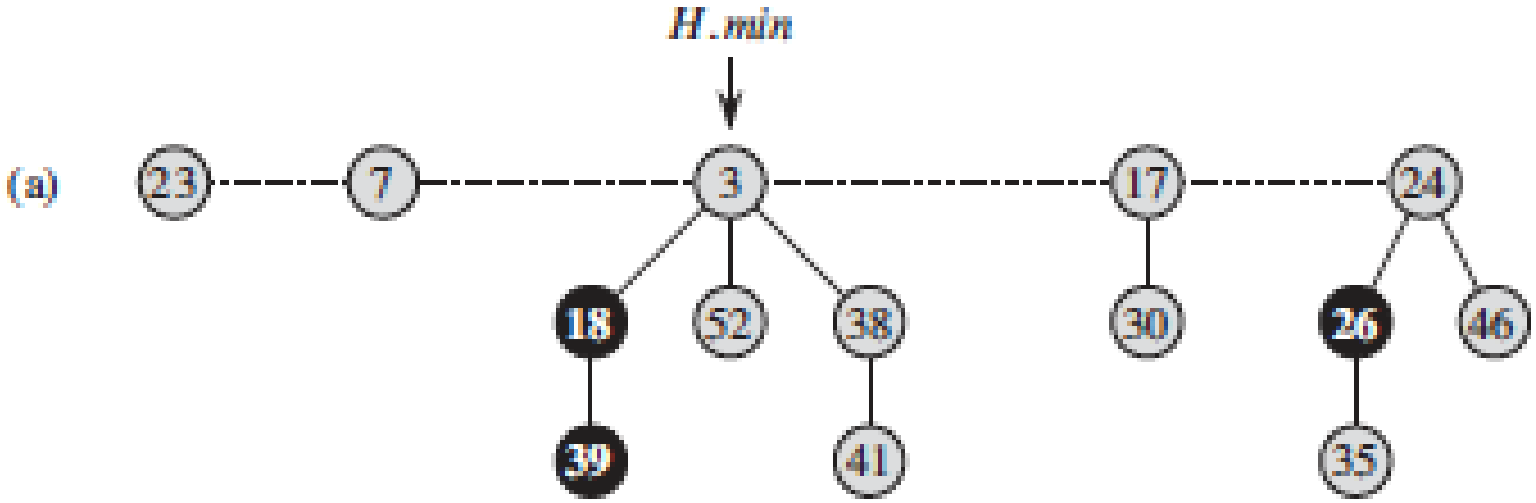
For the operation $delete(h)$ decrease first the value of the node indicated by h to become the minimum, then apply *delete_min*

Finally, we also can implement a *merge* operation, which simply builds the union of two forests and updates the min-pointer

We will look into **Fibonacci heaps** as an efficient way to implement adressable priority queues with these operations

5.3 Fibonacci Heaps

A ***Fibonacci heap*** is a forest of heap-ordered trees



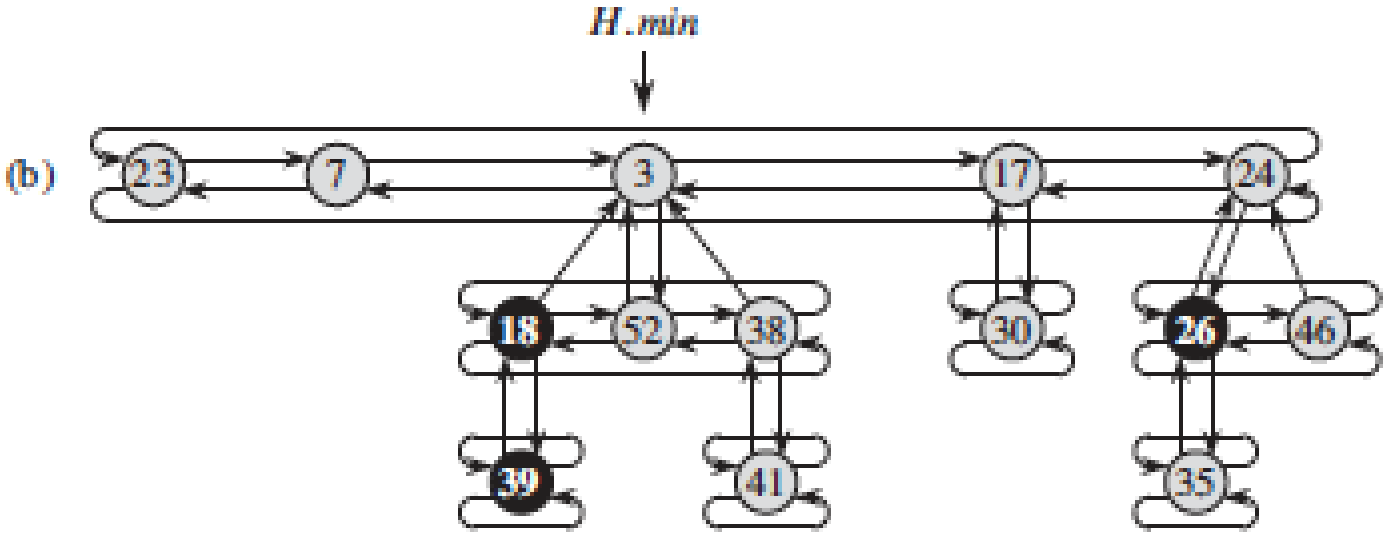
We will concentrate on Fibonacci min-heaps—Fibonacci max-heaps are handled analogously

Representation of Fibonacci Heaps

The roots of the trees in a Fibonacci heap are represented as a circular doubly-linked list (without dummy node), the **root list** of the Fibonacci heap

The counter $H.n$ is used to store the number of nodes in a Fibonacci heap H

There is a pointer $H.min$ to the root with the minimum key value—this root is called the **minimum node** of the Fibonacci heap



Representation of Fibonacci Heaps / cont.

In each tree of a Fibonacci heap all children of the same node are represented in a circular, doubly-linked list using pointers $x.left$ and $x.right$—the **child list** of a node

The number of children is stored in $x.degree$

Each node x in such a tree (except the root) has a pointer $x.parent$ to its parent node

Each non-leaf node x in such a tree has a pointer $x.child$ to one of its children—which one does not matter

Nodes x in a tree in a Fibonacci heap can be marked—use $x.mark = True$

Marking of Nodes / Potential

A node x is **marked**, if it has lost a child since the last time x has been made the child of another node

A newly created node is always unmarked

A node that becomes the child of another node gets unmarked

Marking is used for more sophisticated rebalancing in connection with the *decrease* operation

In a Fibonacci heap H let $t(H)$ denote the **number of trees**, and let $m(H)$ denote the **number of marked nodes**

The ***potential*** of the Fibonacci heap H is defined as $\Phi(H) = t(H) + 2m(H)$—it is used for amortisation analysis

Fibonacci Heap Operations

Operations on Fibonacci heaps are those we introduced for addressable priority queues realised by heap-ordered forests—these operations are implemented in a "lazy" way

To start we build a new Fibonacci heap H by creating an empty root list with a null pointer $H.min$—then $H.n = 0$ and the potential is $\Phi(H) = 0$

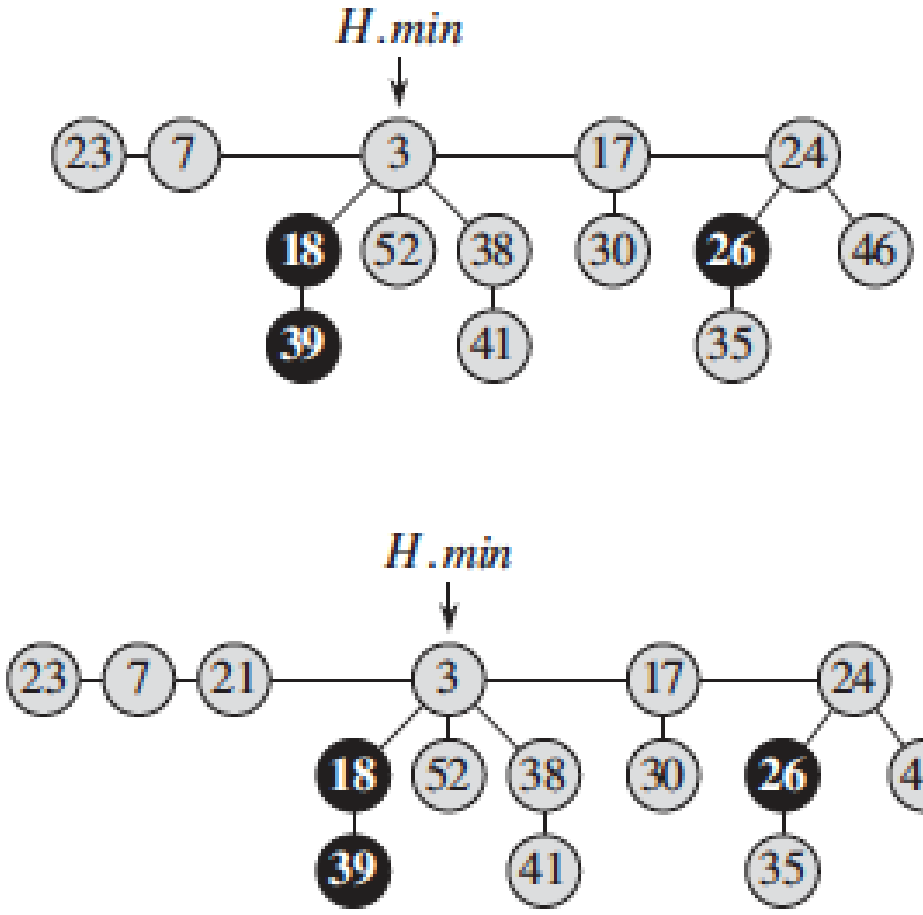
Insertion of a new node just adds a trivial tree (with just this one node); so the root list is enlarged, $H.n$ is increased, and a new min-pointer $H.min$ is found by comparing the key value of the new node with the current value of $H.min$

The real work starts with the *delete_min* operation: when the node identified by $H.min$ is deleted, the root list (and the children) have to be scanned to determine the new minimum and adjust $H.min$

Combining this search with a consolidation of the Fibonacci heap—we will see that we can ensure that after the execution of *delete_min* the size of the root list of a Fibonacci heap with n nodes is at most $D(n) + 1$, where $D(n)$ is the maximum degree of any node in the forest

Example: Insert

We insert a new node with key value 21 into a Fibonacci heap:



Insertion: Amortisation Analysis

The insertion of a new node into a Fibonacci heap H produces a new Fibonacci heap H'

Clearly, the operation takes constant time

Then we have $t(H') = t(H) + 1$ and $m(H') = m(H)$, as there is one new tree, but the number of marked nodes remains unchanged

Then clearly the increase in potential is $\Phi(H') - \Phi(H) = 1$

In order to enable amortisation analysis, we use the potential function Φ, so an insertion makes a contribution of 1 (or more general any constant c) to the potential

Consequently, the amortised time complexity of *insert* is in $O(1)$

Finding the Minimum and Union

As we have a pointer $H.min$ for a Fibonacci heap H, finding the node with the minimum key value and returning this key is done in constant time

As the Fibonacci heap is not altered by this operation, the contribution to the potential is 0, and the amortised time complexity is in $O(1)$

Building the union of two Fibonacci heaps H_1 and H_2 requires just the concatenation of the two root lists; the new min-pointer $H.min$ will be either $H_1.min$ or $H_2.min$

As we use a representation of the root list as a doubly linked list, the time complexity of *union* is in $O(1)$

Clearly, the new resulting Fibonacci heap H has potential $\Phi(H) = \Phi(H_1) + \Phi(H_2)$, i.e. the increase of the potential—the contribution of the *union* operation— is 0

So the amortised time complexity of *union* is in $O(1)$

Extracting the Minimum

For the *delete_min* operation we obtain the minimum directly using the pointer $H.min$, which is done in constant time

Then the child list of the eliminated node has to replace the eliminated node, which requires constant time for doubly-linked lists; we also decrement $H.n$

In order to determine the new minimum node, a naive way would be to scan the new root list, which would take linear time

Instead of such a costly search we let $H.min$ point to the right sibling of the eliminated node, if this sibling exists, or otherwise let it become a null pointer

In case $H.min \neq$ `None` start an operation *consolidate* to reorganise the Fibonacci heap—this will be the most complicated operation

The *consolidate* Operation

The *consolidate* operation iterates through the nodes w in the root list combining trees, if their roots have the same degree, until degrees of roots are distinct

For this we use an array $A[0, \ldots, D(n)]$ such that for $A[i] = y$ we have that y is in the root list with degree i—initially, all entries of the array are null pointers

This requires to have an estimate for $D(n)$, which we will handle later

When two trees are combined, the root x with smaller key value is made parent of the other root y, x is removed from the root list and is unmarked

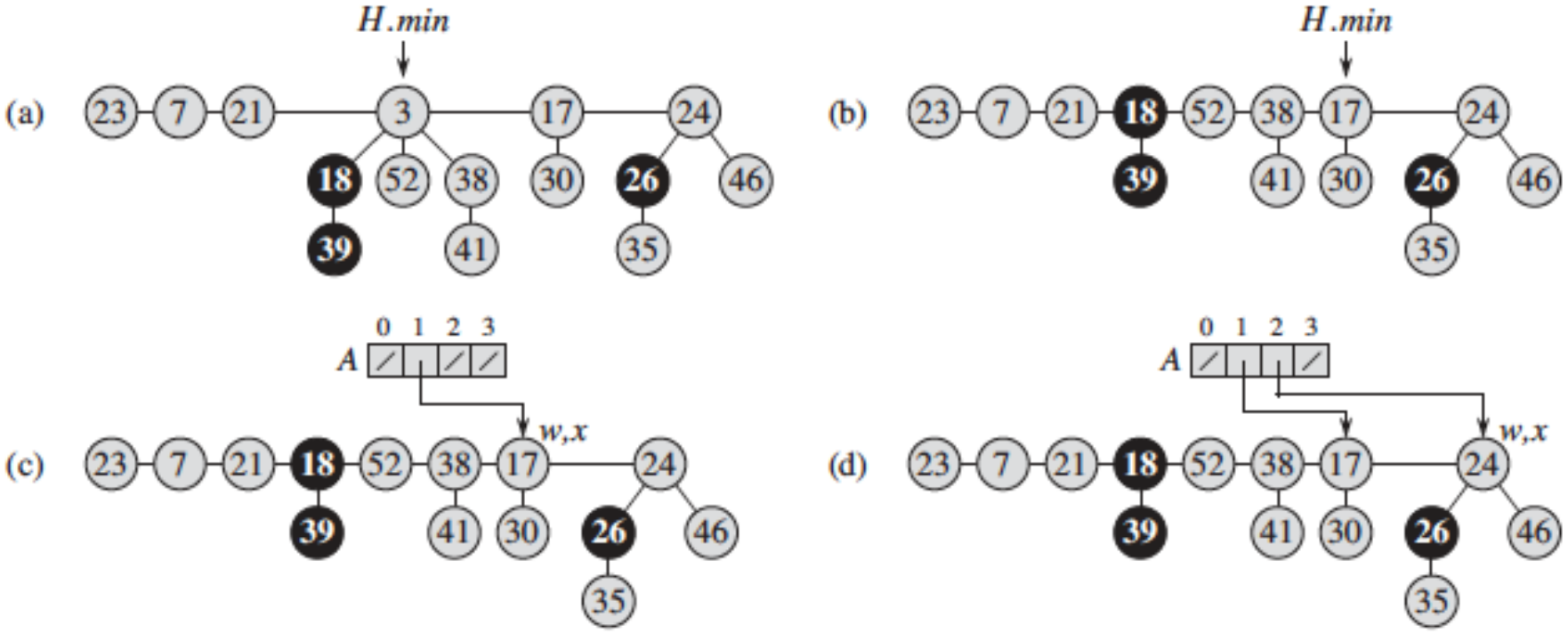
We start with $x = w$, but if x is made the child of a node y, then x is set to the new root y

In this way also the degree $x.degree$ is incremented

We omit further details, and instead look at a detailed example

Example

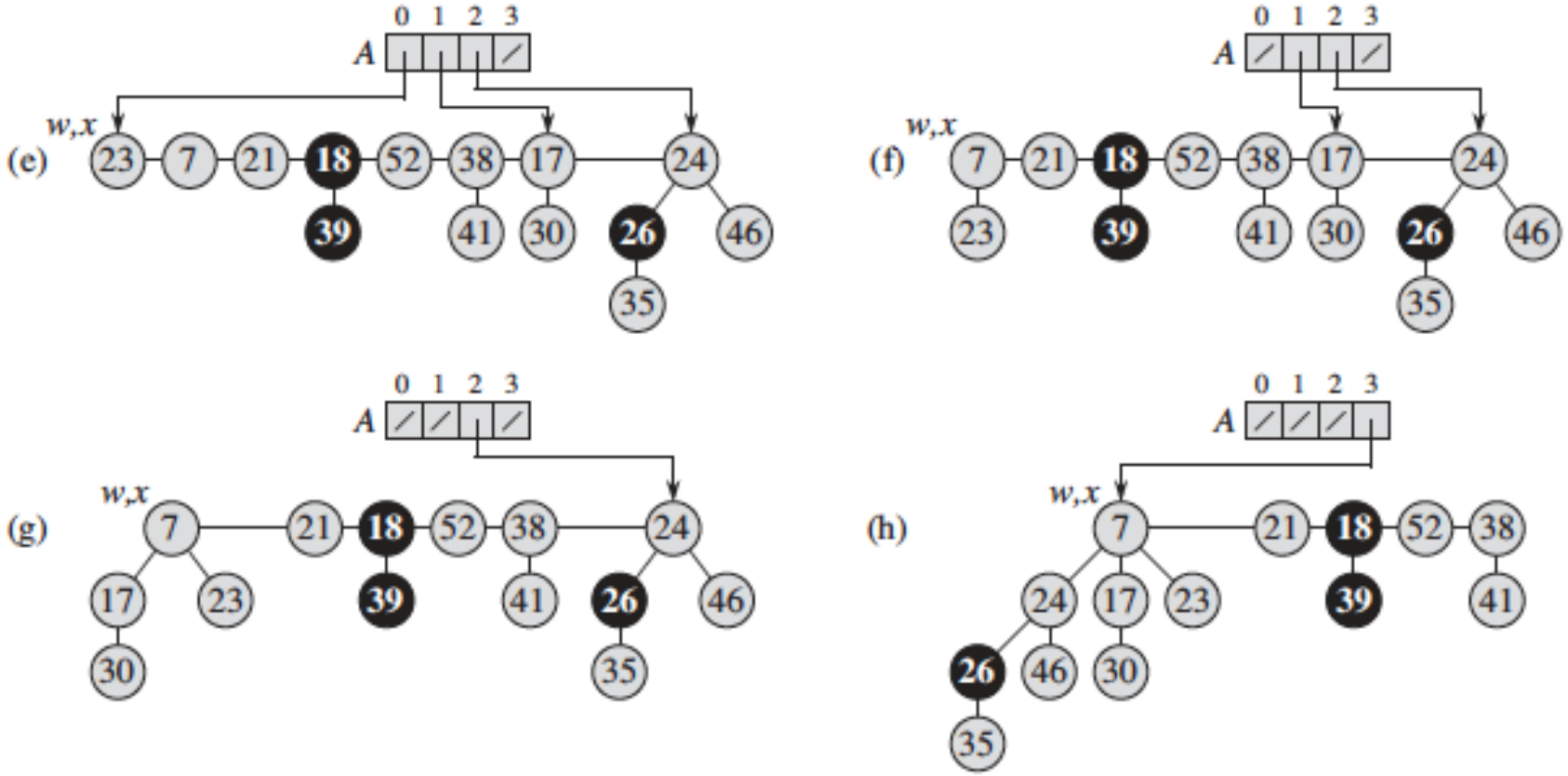
First remove the node with the minimum 3 (a), add the children to the root list, and make the right sibling with 17 the new min-node (b)



Start with the min-node and always follow the right pointer: first only $A[1] = 17$ (c), then $A[2] = 24$ (d) and $A[0] = 23$ (e) are set

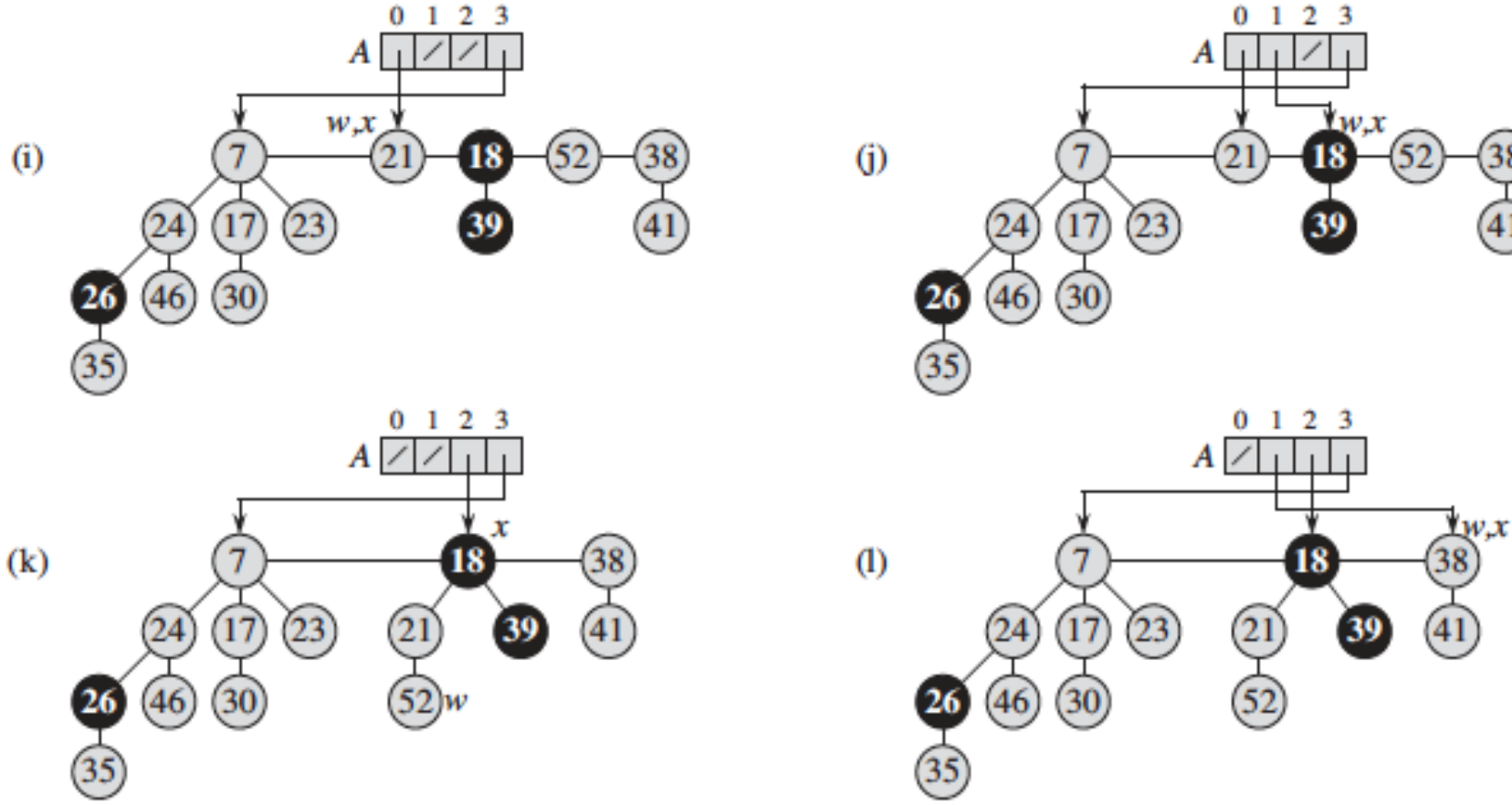
Example / cont.

The first time trees are combined occurs with 7 and 23 (f), then further with 17 (g) and 24 (h)



Example / cont.

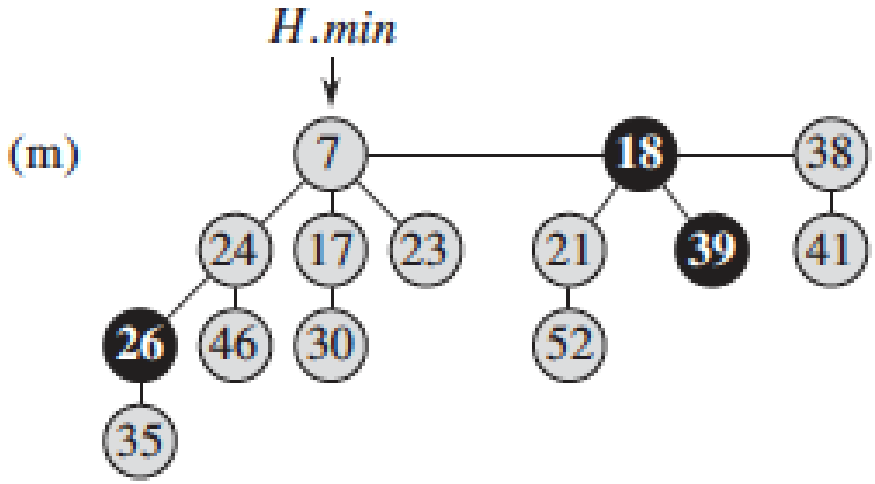
For 21 and 18 there is nothing to do (i), then combine first 21 with 52 (j), then further 21 with 18 (k)



After this all nodes in the root list have different degrees

Example / cont.

Finally, run through the new root list to asssign $H.min$ to point to the correct node



Note that in the consolidation no node becomes marked, so $m(H)$ can only decrease

Amortised Complexity of Minimum Extraction

We have already seen that deleting the min-node and readjusting the Fibonacci heap has a time complexity in $O(1)$

Then the root list at the start of *consolidate* has at most the length $D(n)+t(H)-1$

In each iteration step two trees are combined, which takes constant time

As the number of iterations is bounded by the length of the root list, the complexity of minimum extraction including the consolidation is in $O(D(n) + t(H))$

As degrees are always between 0 and $D(n)$ and finally there are only nodes in the root list with different degrees, there will be at most $D(n) + 1$ trees remaining

Amortised Complexity / cont.

So the potential before *delete_min* is $\Phi(H) = t(H) + 2m(H)$

The potential after *delete_min* is $\Phi(H') \leq D(n) + 1 + 2m(H)$

Thus, the amortised complexity of *delete_min* becomes at most

$$\begin{aligned} & c(D(n) + t(H)) + c(\Phi(H') - \Phi(H)) \\ = \; & c(D(n) + t(H)) + c(D(n) + 1 - t(H)) \\ = \; & 2cD(n) + c \end{aligned}$$

which is in $O(D(n))$

We will show later that $D(n) \leq \log_\Phi n$ holds with $\Phi = \frac{1}{2}(1 + \sqrt{5}) = 1.61803\ldots$ (the **golden ratio**), so we actually have amortised complexity in $O(\log n)$

Deletion and Decreasing Keys

For the operation $decrease(h, k)$ we can (if the heap order is violated) cut out the subtree rooted at the node given by the handle h, and insert its root together with the new key value k into the root list—if necessary, we have to adjust $H.min$

When doing so the parent of the cut-out subtree will be marked, if it was unmarked

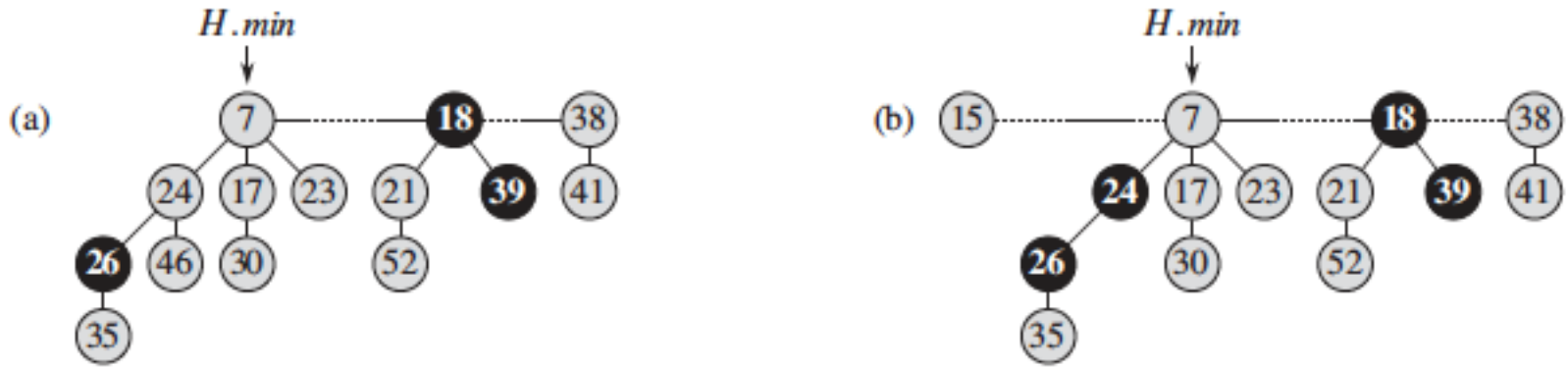
If the parent was already marked, it is also cut out, unmarked and inserted into the root list—this is cascaded upwards as long as parents are marked, and terminates at a root

For $delete(h)$ for an arbitrary node first decrease the key value of h to a value smaller than the current minimum, e.g. execute $decrease(h, -\infty)$

Then apply $delete\_min$

Example: Cascaded Decrease of Keys

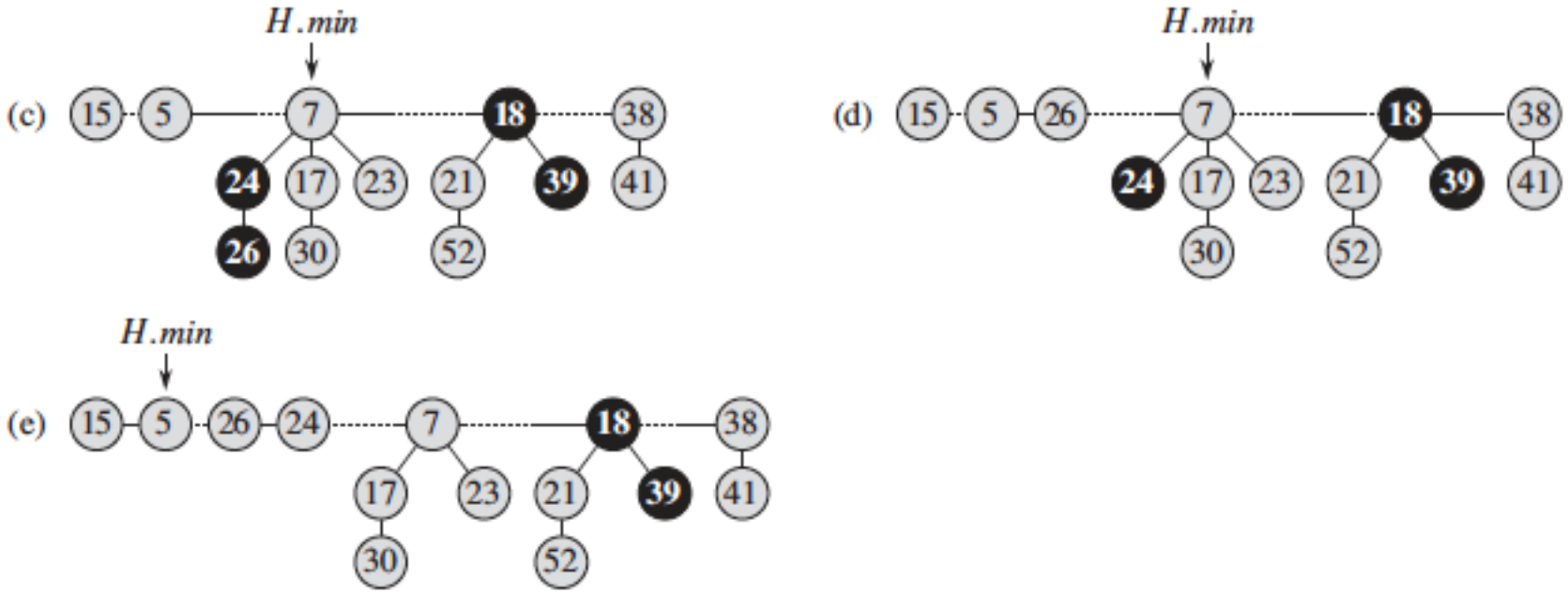
Decrease the key value of the node with current key 46 to 15



(a) shows the Fibonacci heap before the decrease, and (b) shows the Fibonacci heap after the decrease

Example / cont.

Decrease the key value of the node with current key 35 to 5; then the marked parent 26 and the marked grandparent 24 are moved to the root list



(b) shows the Fibonacci heap before the decrease, (c) after the decrease, and (d) and (e) the results after the two cascaded cuts

Amortised Complexity of Decrease

Start by determining the actual cost of a *decrease* operation; this is determined by the number c of cascading cuts

The decrease of the key value as well as each cut including the insertion into the root list take time in $O(1)$, so the time complexity of the *decrease* operation is in $O(c)$

Next consider the change in the potential: each of the cuts adds a new tree to the root list, and (except for the first cut) turns a marked node into an unmarked one, i.e. $t(H)$ is increased by c, and $m(H)$ is decreased by $c - 1$

In addition the last cut may be accompanied with a node becoming marked, so the difference in potential is at most

$$(t(H) + c) + 2(m(H) - c + 2) - (t(H) + 2m(H)) = 4 - c$$

So the amortised cost of *decrease* is in $O(c + 4 - c) = O(1)$

Estimation of Tree Degrees

We now tend to the (missing) proof that $D(n) \leq \log_\Phi n$ holds

This proof will also highlight why the cascading cut of marked nodes in the *decrease* is needed, and why the data structure is called a **Fibonacci heap**

For each node x in a Fibonacci heap let $size(x)$ denote the number of nodes in the subtree rooted at x (including x itself)

Lemma. Let x be any node in a Fibonacci heap and assume $x.degree = k$. Let $y_1, \ldots, y_k$ denote the children of x in the order they were linked to x. Then $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i \geq 2$.

Proof. Obviously, we have $y_i.degree \geq 0$ for all $1 \leq i \leq k$

For $i \geq 2$, when y_i was linked to x, then $y_1, \ldots, y_{i-1}$ were already children of x, so we had $x.degree \geq i - 1$

Proof (cont.)

As trees are merged by the *consolidate* operation, only if the have the same degree, we must have had $x.degree = y_i.degree$, hence $y_i.degree \geq i - 1$ at the time x was linked with y_i

Since x and y_i were linked, y_i can have lost at most one child—if it had lost a second child, it would have been cut out—which gives us $y_i.degree \geq i - 2$

This completes the proof of the lemma

Now recall the sequence of Fibonacci numbers $(f_i)_{i \in \mathbb{N}}$ defined by $f_0 = 0$, $f_1 = 1$ and $f_{i+2} = f_i + f_{i+1}$ for all $i \in \mathbb{N}$

A simple induction shows $f_{k+2} = 1 + \sum_{i=o}^{k} f_i$

Fibonacci Numbers

Claim. We have $f_{k+2} \geq \Phi^k$ for all $k \in \mathbb{N}$

We prove the claim by induction on k: for $k = 0$ we have $f_2 = 1 \geq \Phi^0$, and for $k = 1$ we have $f_3 = 2 \geq \Phi^1$—this constitutes the induction base

So let $k \geq 2$ and assume that $f_{i+2} \geq \Phi^i$ holds for all $0 \leq i \leq k-1$. Then we get

$$\begin{aligned} f_{k+2} &= f_{k+1} + f_k \\ &\geq \Phi^{k-1} + \Phi^{k-2} && \text{(by the induction hypothesis)} \\ &= \Phi^{k-2}(\Phi + 1) \\ &= \Phi^{k-2} \cdot \Phi^2 && \text{(as } \Phi \text{ is a root of } x^2 - x - 1 = 0) \\ &= \Phi^k \end{aligned}$$

which completes the proof of the claim

Estimation of Size

Lemma. Let x be any node in a Fibonacci heap and assume $x.degree = k$. Then we have $size(x) \geq f_{k+2} \geq \Phi^k$.

Proof. Let s_k denote the minimum possible $size(z)$ of any node z with $z.degree = k$

Clearly, the s_k form a monotone increasing sequence with $s_0 = 1$ and $s_1 = 2$

As in our previous lemma let $y_1, \ldots, y_k$ denote the children of x in the order they were linked to x

Using our previous lemma we get (for $k \geq 2$)

$$size(x) \;\geq\; s_k \;\geq\; 2 + \sum_{i=2}^{k} s_{y_i.degree} \;\geq\; 2 + \sum_{i=2}^{k} s_{i-2} \; .$$

Proof (cont.)

Next we show by induction over k: $s_k \geq f_{k+2}$—the induction base ($k = 0$ or $k = 1$) is obvious

So let $k \geq 2$ and assume $s_i \geq f_{i+2}$ holds for all $0 \leq i \leq k-1$; then we get

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} f_i \qquad \text{(by the induction hypothesis)} \\ &= 1 + \sum_{i=0}^{k} f_i \;=\; f_{k+2} \end{aligned}$$

Together with are previously shown claim on Fibonacci numbers we obtain $size(x) \geq \Phi^k$ as claimed; this completes the proof of he lemma

Estimation of Tree Degrees – Final Step

Corollary. The maximum degree $D(n)$ of any node in a Fibonacci heap with n nodes satisfies $D(n) \leq \log_\Phi n$, hence $D(n) \in O(\log n)$.

Let x be any node in a Fibonacci heap and assume $x.degree = k$

According to our previous lemma we have $size(x) \geq \Phi^k$, hence also $n \geq \Phi^k$

The maximum number k here is $D(n)$, so we get $n \geq \Phi^{D(n)}$

Taking the logarithm on bothe sides we get $D(n) \leq \log_\Phi n$, which proves the corollary

5.4 Van Emde Boas Trees

We have seen different data structures that can be used to support the operations on priority queues

All these structures can also be used for sorting: first insert m elements, then use *delete_min* m times

As we have a lower bound of $\Omega(n \log n)$ for sorting, the $O(\log n)$ complexity for the priority queue operations is already optimal, provided that we have to rely on comparisons only

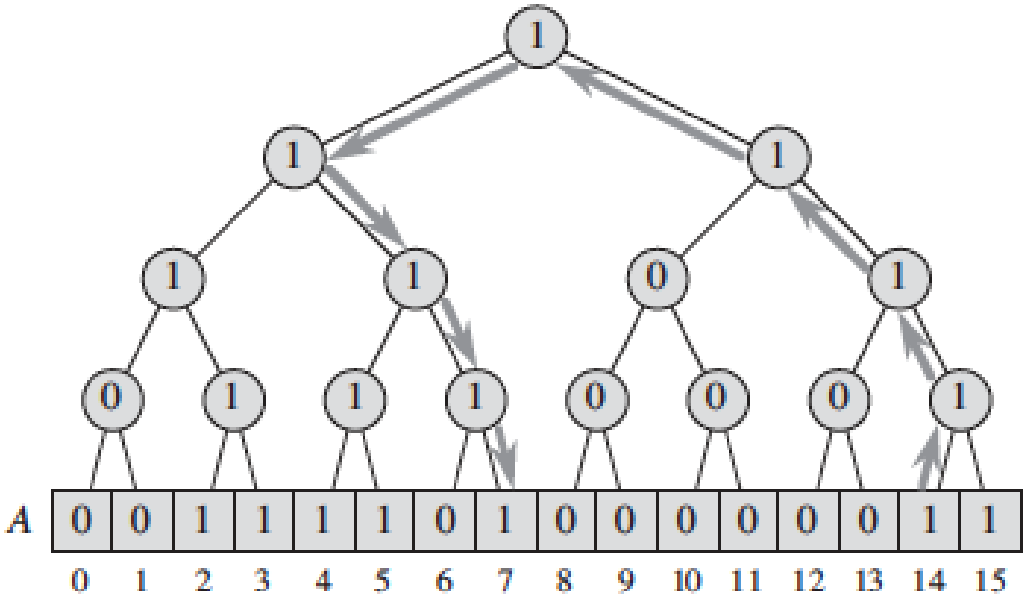
We have also seen that we can obtain almost linear complexity for sorting, if we exploit additional properties of the elements other than the possibility to compare them with respect to a given total order

van Emde Boas trees (**vEB-trees**) do exactly this: using integers in a range $0, \ldots, n-1$ without duplicates we can obtain $O(\log \log n)$ worst case complexity

Superimposing a Binary Tree Structure

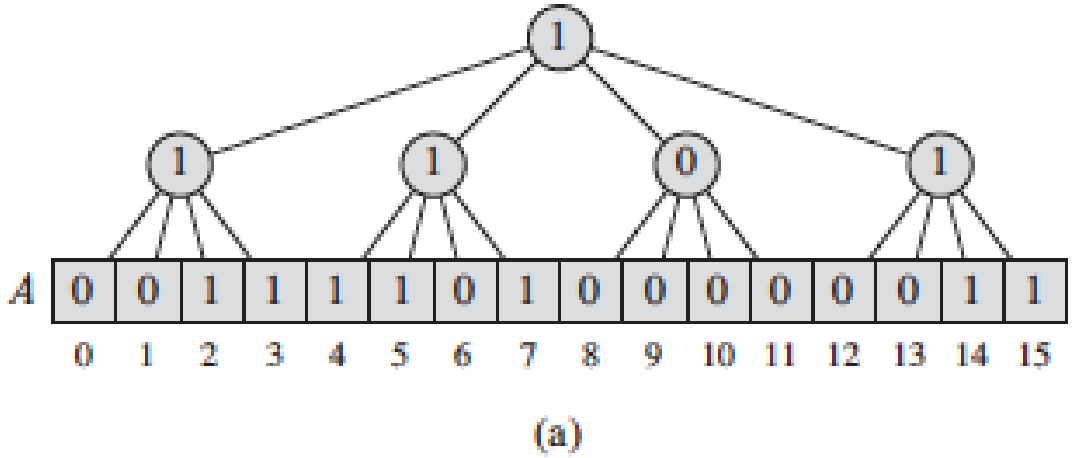
To motivate the idea behind vEB-trees let us first use a bit array $A[0, \ldots, u-1]$, where $A[x] = 1$ holds iff x is a member of a set (representing the queue)

Then we can superimpose a binary tree on top of the bit array, and indicate in every non-leaf node, if the subtree rooted at this node leads to an element in the set or not



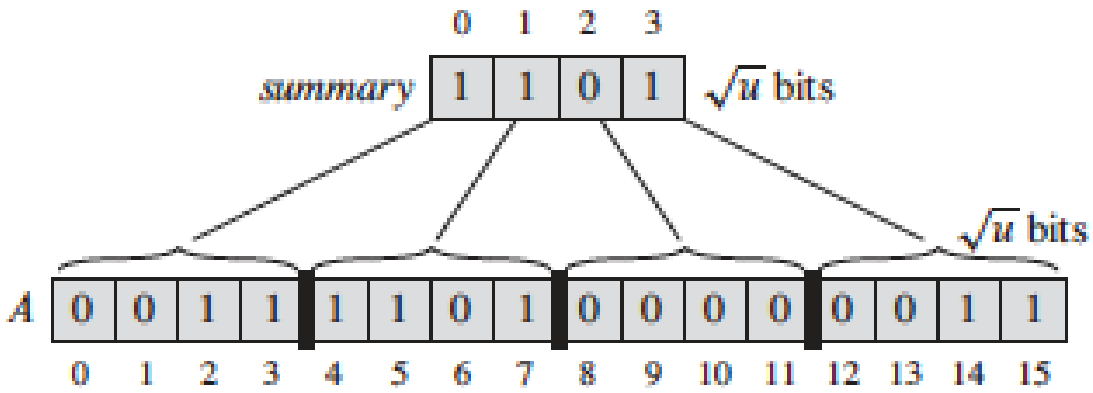
Superimposing a Non-Binary Tree Structure

We can generalise this idea to a tree with constant degree $\sqrt{u}$ (assuming $u = 2^{2k}$ for some k)



Superimposing a Non-Binary Tree Structure / cont.

Then we can use an array *summary* to summarise the information, which parts of the super-imposed tree lead to elements in the represented set



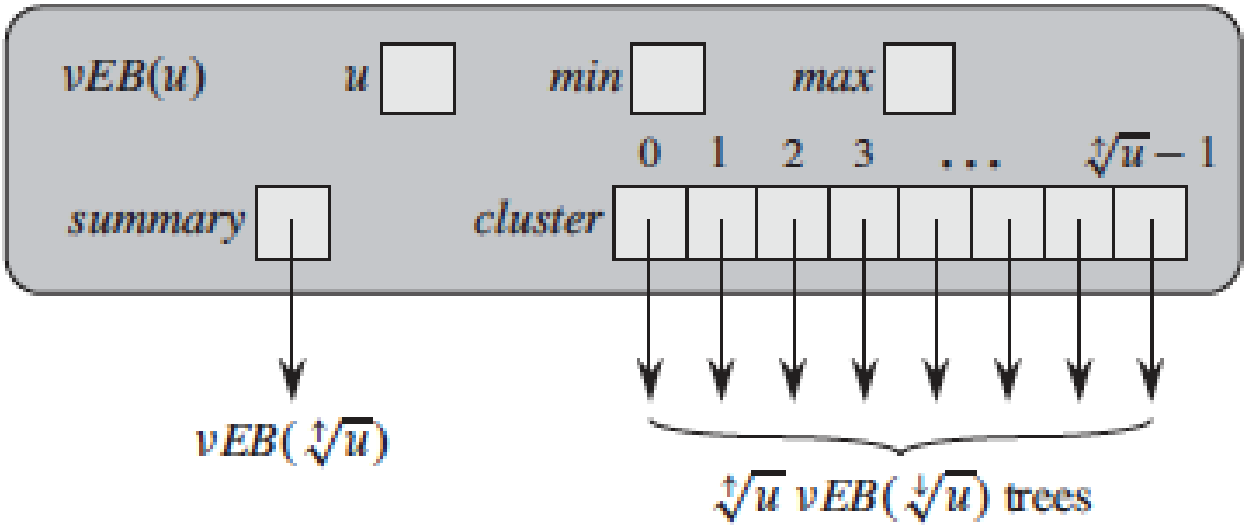
(b)

Nodes in van Emde Boas Trees

We may as well use this idea recursively, which leads us to vEB-trees

Nodes in vEB-trees contain pointers to a *summary* node, to $\sqrt[\uparrow]{u}$ children nodes

The sought complexity only results, if also the minimum and maximum of a vEB-tree are stored in a node



Preliminaries

Assuming $u = 2^{2^k}$ to ensure than recursive application of $\sqrt{\cdot}$ returns integers is very restrictive

Instead we only assume the u is a power of 2

Then the **upper root** $\sqrt[\uparrow]{u} = 2^{\lceil \log_2 u/2 \rceil}$ and the **lower root** $\sqrt[\downarrow]{u} = 2^{\lfloor \log_2 u/2 \rfloor}$ are well-defined with $u = \sqrt[\uparrow]{u} \cdot \sqrt[\downarrow]{u}$

We will need the following auxiliary functions:

$$\begin{aligned} high(x) &= \lfloor x / \sqrt[\downarrow]{u} \rfloor \\ low(x) &= x \mod \sqrt[\downarrow]{u} \\ index(x,y) &= x \cdot \sqrt[\downarrow]{u} + y \end{aligned}$$

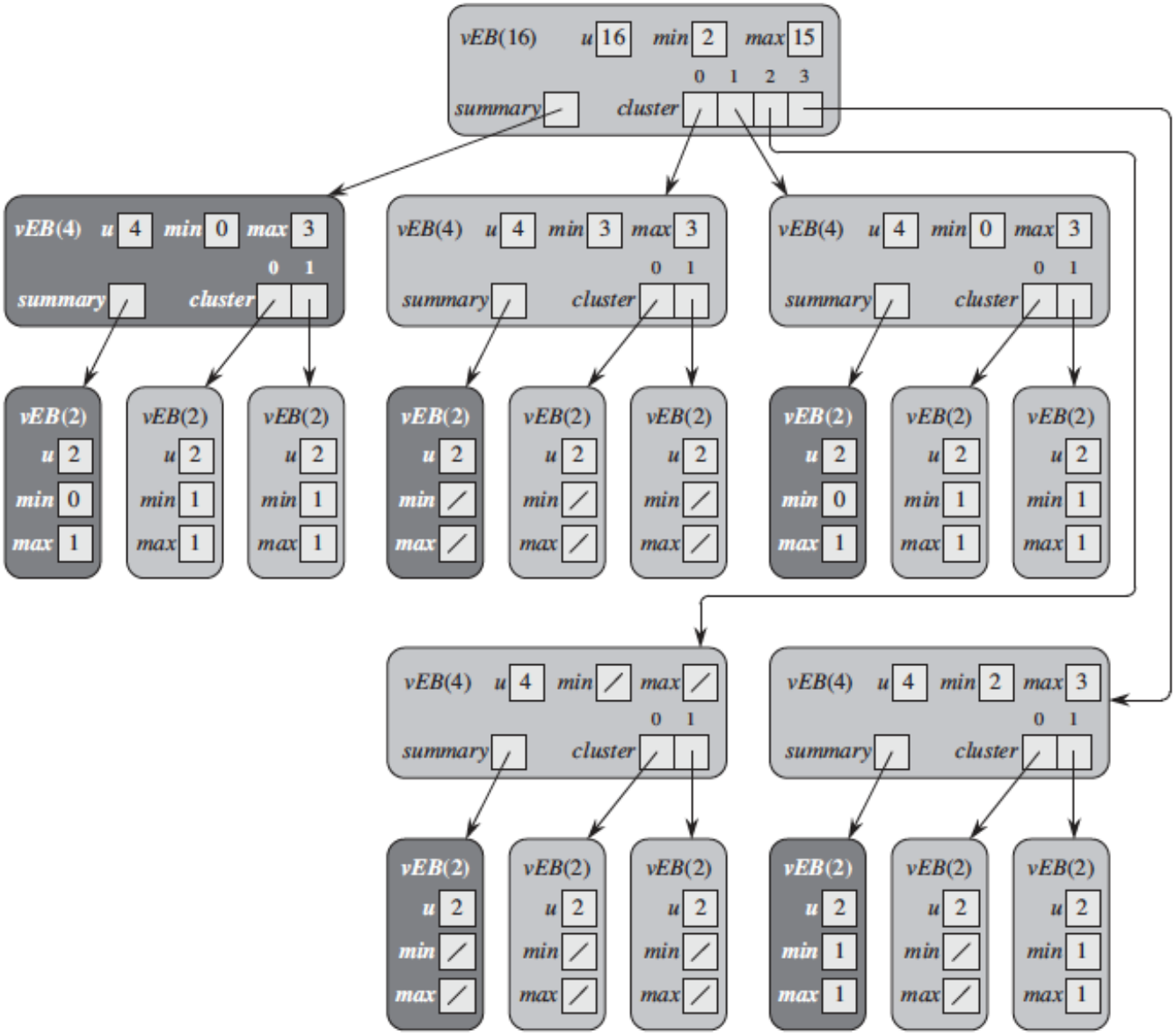
Definition

A **van Emde Boas tree** $vEB(u)$ over a universe of size u is a tree, where the root comprises

- the size u of the universe, the minimum element min in the tree, the maximum element max in the tree,
- a pointer $summary$ to a tree $vEB(\sqrt[\uparrow]{u})$ (unless $u = 2$),
- an array $cluster$ of size $\sqrt[\uparrow]{u}$, where each entry is a pointer to a tree $vEB(\sqrt[\downarrow]{u})$ (unless $u = 2$)

The element stored in min does not occur in any of the vEB-trees addressed by a pointer in $cluster$

Example


vEB(16) u 16 min 2 max 15
0 1 2 3
summary cluster
vEB(4) u 4 min 0 max 3
0 1
summary cluster
vEB(4) u 4 min 3 max 3
0 1
summary cluster
vEB(4) u 4 min 0 max 3
0 1
summary cluster
vEB(4) u 4 min / max /
0 1
summary cluster
vEB(4) u 4 min 2 max 3
0 1
summary cluster
vEB(2) u 2 min 0 max 1
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min / max /
vEB(2) u 2 min / max /
vEB(2) u 2 min / max /
vEB(2) u 2 min 0 max 1
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min / max /
vEB(2) u 2 min / max /
vEB(2) u 2 min / max /
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min / max /
vEB(2) u 2 min 1 max 1

Operations on vEB-Trees

The following operations are relevant for vEB-trees:

- Find the *minimum* and the *maximum* in the queue represented by the tree
- Search for an element in the queue and return *true* or *false*
- Find the *successor* and the *predecessor* of an element in the queue
- *Insert* a new element into the queue or *delete* an element from the queue

We will look through these operations, but spare the sophisticated complexity analysis

Minimum, Maximum, Search

As minimum and maximum are stored in the root, we only have to return these values

In order to search for an element x we use a recursive approach

If $x = min$ or $x = max$ holds, we return the result *true*

Otherwise, if $u = 2$ we return the result *false*, as there can be at most two elements, which then are min and max

If this is not the case, search for $low(x)$ in the vEB-tree indicated by $cluster[high(x)]$

Finding the Successor and Predecessor

In a vEB-tree with $u = 2$ the successor of x can only be max, if max is defined and $x = 0$ holds—otherwise it is undefined

In general, if min is defined and $x < min$ holds, then min is the successor of x

Otherwise, retrieve the maximum max_low from the vEB-tree indicated by $cluster[high(x)]$

If max_low is defined and $> low(x)$ search for the successor s_1 of $low(x)$ in the vEB-tree indicated by $cluster[high(x)]$ and return $index(high(x), s_1)$

Otherwise search for the successor s_2 of $high(x)$ in the vEB-tree indicated by $summary$

If s_2 is defined, retrieve the minimum s_1 of the vEB-tree indicated by $cluster[s_2]$ and return $index(s_2, s_1)$

Finding the predecessor of x is done analogously

Insertion into a vEB-Tree

Inserting an element x into an empty vEB-tree (i.e. min is undefined) requires only to set $min = max = x$

Inserting x in a non-empty vEB-tree with $x < min$ requires to first exchange x with min

For $u > 2$, if the minimum of the vEB-tree indicated by $cluster[high(x)]$ is undefined, insert $high(x)$ into the vEB-tree indicated by $summary$, and insert $low(x)$ into the empty vEB-tree indicated by $cluster[high(x)]$

Otherwise, if the minimum is defined, just insert $low(x)$ into the non-empty vEB-tree indicated by $cluster[high(x)]$

if $x > max$ holds, change max to x

Deletion from a vEB-Tree

Assume that x is an element of the vEB-tree

If the tree is trivial, i.e. $min = max$ holds, make min and max undefined

The case $u = 2$ is likewise trivial, as x must be 0 or 1—in the former case set $min = max = 1$; in the latter case set $min = max = 0$

In general, if $x = min$ holds, then retrieve the minimum m_1 from the vEB-tree indicated by *summary*, retrieve the minimum m_2 from the vEB-tree indicated by $cluster[m_1]$ and set min to $index(m_1, m_2)$

Then delete $low(x)$ from the vEB-tree indicated by $cluster[high(x)]$

If the minimum of the vEB-tree indicated by $cluster[high(x)]$ is undefined, delete $high(x)$ from the vEB-tree indicated by *summary*

Finally, consider the case that $x = max$ holds—we omit further details for this case