Contents Lecture 8

- Review of the heap data structure
- Array-based heap (binary heap)
- Overview of Fibonacci heap
- Hollow heap

Review of the heap data structure

- $(key, value)$ pairs are stored
- Primarily used for priority queues
- Operations:
  - make heap
  - insert pair
  - change key — some heaps only support decrease the value of the key
  - min
  - delete min
- Efficient search is not supported

Overview of array-based heap data structure

- Can store up to n pairs $(key, value)$
- An array indexed from 1 to n is used
- Normally best to allocate $n+1$ elements and just waste one element
- The root is stored at index 1
- Let k_j denote key of pair stored at index j
- The heap order means that $k_j \leq k_{2j}$ and $k_j \leq k_{2j+1}$
- But nothing about k_{2j} vs. k_{2j+1}

Operations on array based heap

- Assume the heap contains n pairs
- The min pair is at index 1
- To delete the min pair it is saved somewhere and the pair at index n is moved to index 1, and n is decremented
- This pair is then moved down which takes $O(\log n)$ time
- A new pair is inserted at index $n + 1$
- The new pair is then moved up which also takes $O(\log n)$ time
- Changing the priority takes $O(\log n)$ time as well

Initializing a heap from an array

- One option: n inserts for $O(n \log n)$ total time
- Instead view the array as consisting of n heaps with one element each

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}

- $k = \lfloor n/2 \rfloor = 5$
- If both a_{2k} and a_{2k+1} exist they are roots of valid heaps
- Select one of a_k, a_{2k} and a_{2k+1} to be the root of a new heap consisting of these three
- That is done by moving a_k down in the heap, if needed
- After that a_k is the root of a valid heap
- Continue with $a_{k-1}, a_{k-2}, ..., a_1$
- We need to do this from right to left since the children of a_k $(a_{2k}$ and $a_{2k+1})$ must be valid heaps which they would not be if we started at a_1

An example

- initial array

34	3	55	2	13	0	1	8	5	1	21

- $k = 5$, $a_5 = 13$, $a_{10} = 1$, $a_{11} = 21$ so move down 13

34	3	55	2	1	0	1	8	5	13	21

- $k = 4$, $a_4 = 2$, $a_8 = 8$, $a_9 = 5$ so nothing to do

34	3	55	2	1	0	1	8	5	13	21

- $k = 3$, $a_3 = 55$, $a_6 = 0$, $a_7 = 1$ so move down 55

34	3	0	2	1	55	1	8	5	13	21

- $k = 2$, $a_2 = 3$, $a_4 = 2$, $a_5 = 1$ so move down 3

34	1	0	2	3	55	1	8	5	13	21

- $k = 1$, $a_1 = 34$, $a_2 = 1$, $a_3 = 0$ so move down 34

0	1	34	2	3	55	1	8	5	13	21
0	1	1	2	3	55	34	8	5	13	21

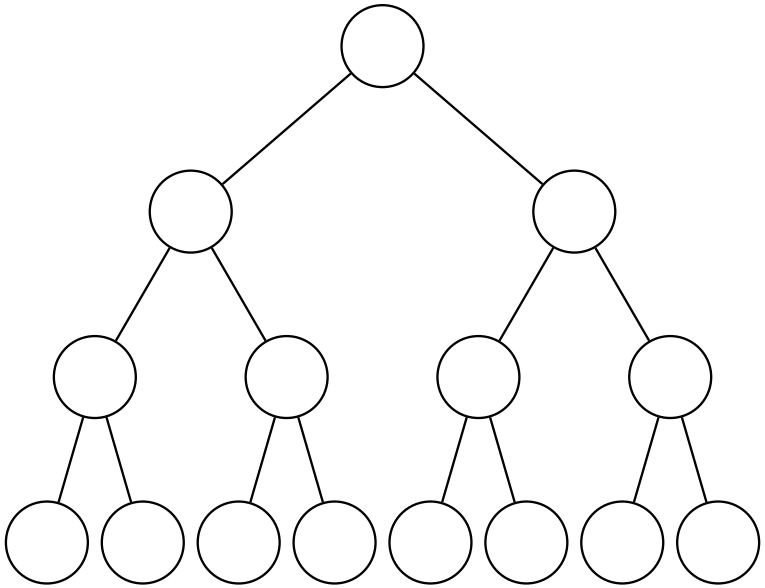
now 34 cannot move any further

Time complexity of initializing a heap

- Each move down is $O(\log n)$
- We do $n/2$ move down
- Pessimistic time bound: $\frac{n}{2} \log n = O(n \log n)$
- But most move down are far less than $O(\log n)$
- Can we make a more accurate analysis?

More about binary heap

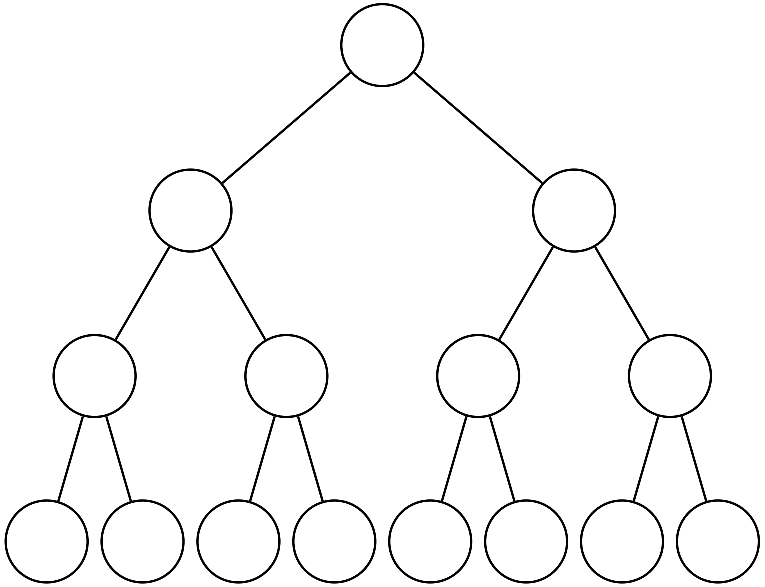
- Consider first a full heap
- When we increase the height by one, we double the number of leaves
- The height of an n element heap is $\lfloor \log_2 n \rfloor$
- For $n = 15$, height $h = 3 = \lfloor \log_2 15 \rfloor$



- The height is 3 for $8 \leq n \leq 15$ as expected

Number of nodes at a certain height

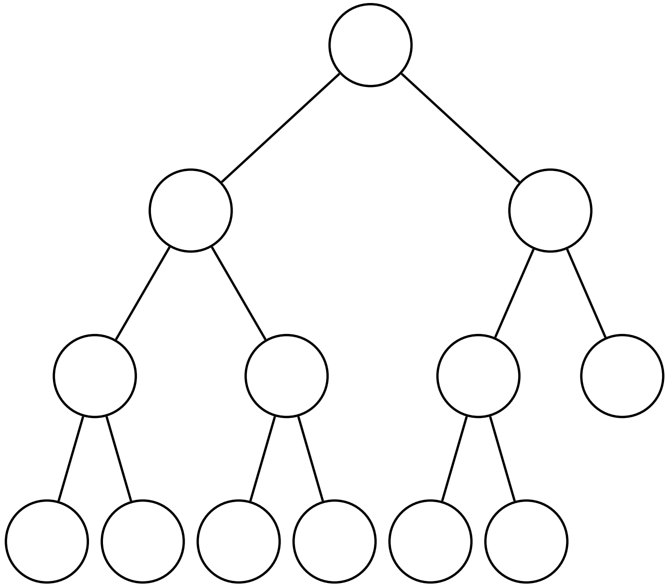
- Again first a full heap, with $n = 15$
- A leaf is at height 0 and we have 8 leaves
- We have 4 nodes at height 1, 2 at height 2, and 1 node at height 3



- At height i there are $\lceil 15/2^{i+1} \rceil$ nodes
- For example: $i = 0$ gives $\lceil 15/2^{0+1} \rceil = \lceil 7.5 \rceil = 8$ nodes
- And $i = 2$ gives $\lceil 15/2^{2+1} \rceil = \lceil 1.875 \rceil = 2$ nodes
- In general $x \leq \lceil n/2^{i+1} \rceil$ nodes at height i

Number of nodes at a certain height

- Now a heap that is not full, with $n = 13$



- In general we have $x \leq \lceil n/2^{i+1} \rceil$ nodes at height i
- 3 nodes at height 1
- $3 \leq \lceil 13/2^{1+1} \rceil = \lceil 3.25 \rceil = 4$

A note

- Recall a geometric series (geometrisk summa)
- For $|x| < 1$ we have:

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}$$

- We also see:

$$\begin{aligned} \frac{d}{dx}\sum_{h=0}^{\infty} x^h &= \frac{d}{dx}\frac{1}{1-x} \\ \sum_{h=0}^{\infty} hx^{h-1} &= \frac{1}{(1-x)^2} \end{aligned}$$

- Multiply by x:

$$x\sum_{h=0}^{\infty} hx^{h-1} = \frac{x}{(1-x)^2}$$

- With $x = 1/2$ we get: $\dfrac{\frac{1}{2}}{(1-\frac{1}{2})^2} = \dfrac{\frac{1}{2}}{\frac{1}{4}} = 2$

A more accurate analysis of initializing a heap

$$
\begin{aligned}
\sum_{h=0}^{\lfloor \log n \rfloor} (\textit{nodes at height } h) O(h) &= \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \\
&= O(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}) \\
&= O(n \sum_{h=0}^{\infty} h(\tfrac{1}{2})^h) \\
&= O(n \frac{\frac{1}{2}}{(1-\frac{1}{2})^2}) \\
&= O(2n) \\
&= O(n).
\end{aligned}
$$

Fibonacci heap

- A list of trees instead of an array
- Each tree satisfies heap order
- Worst-case constant time to insert a new $(key, value)$ pair
- Insert: create a new tree and check if it is the minimum
- Basic idea of decrease-key: remove it from the parent and make it a new root and possibly make additional updates
- Recall from preflow-push: amortized time takes multiple operations into account and not only worst case for each
- Amortized $O(\log n)$ time to remove minimum
- Each tree node uses five pointers, an integer and a boolean

Hollow heaps

- Simpler and better than Fibonacci heaps
- A disadvantage is that some nodes have no data and still consume memory
- They can be cleaned away when needed though
- This is research published in 2015 and 2017 by Dueholm Hansen, Tarjan, Kaplan and Zwick
- Hollow heaps also uses trees, just as Fibonacci heaps

Nodes and elements

- A node is a tree node in the hollow heap
- An element is the data stored in the heap: a $(key, value)$ pair
- A node with an element is full
- A node with no element is hollow

Hollow nodes

- An element can be removed from a node which then becomes a hollow node
- A node is not the element but instead has a pointer to an element (or null)
- Thus a node cannot be an element — only point to an element
- A node also has a key: identical to the element's or to the key of the element the node previously had
- A hollow node never gets a new element
- Hollow nodes which are children of the minimum node are thrown away when the minimum is deleted
- Hollow nodes can be garbage collected and thrown if memory is needed

Three versions of Hollow heaps

- Focus on the exam is first version
- Probably fastest version: multiple root nodes
- Not so important versions for the course: one root node, and two parents (i.e., I will not ask about them)
- The purpose is to give you key insights what hollow heaps are about but not detailed proofs or implementation
- The exam may have a simple question about hollow heaps
- I am supervising a MSc thesis about hollow heaps for a parallel implementation of Dijkstra's algorithm: very interesting (I think)

Version 1: Multiple root nodes

- We have a list of root nodes
- When an element is inserted, a new node is created
- This node becomes a new root
- It is then checked if this is the new minimum node

Link operation

- Compare the keys of two nodes and make the one with smaller key the parent of the other
- The heap order of a tree is maintained using link operations
- A node has a single linked list of children
- A new child is inserted first in this list
- Links are only performed at a delete-min and when merging two heaps — but not at an insert

Decrease-key operation

- If the element is a root, then the key is simply reduced — and check if this is the new min
- If not, a new root is created with the element
- The element is then moved from the previous node which becomes hollow
- Some of the children are moved to the new node as well

Delete operation

- If the deleted element is is not the minimum, the node with it simply becomes hollow and we are done
- If it is the minimum element, all hollow root nodes are destroyed by making their children new full root nodes
- To reduce the number of root nodes, a number of link operations are performed
- Quiz: why should we try to reduce the number of root nodes?

Rank

- Each node has a rank, which is a non-negative integer initially zero
- When reducing the number of hollow roots, link operations are performed on root nodes with the same rank
- The node which becomes the parent at a link has its rank incremented by one

Invariant

- A node with rank r has exactly r children, except if $r > 2$ and the node has become hollow when the key of its former element was decreased
- In that case, the node has two children with ranks $r - 2$ and $r - 1$
- Otherwise its r children have ranks $r - 1$, $r - 2$, ... 2, 1, 0.
- Let r_u be the rank of u
- When an element is moved from a node u to a node v the rank of v is set to $\max\{0, r_u - 2\}$: -2 because up to two children stay at u
- All children of u with rank less than r_v are moved to v, with their children
- If the rank of u is at least 2, then u keeps two children with ranks $r - 2$ and $r - 1$
- If the rank of u is one, then u keeps its child (with rank zero).

Fibonacci numbers

- Recall Fibonacci numbers: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$
- $F_0 = 0, F_1 = 1$ and $F_i = F_{i-1} + F_{i-2}$
- $F_{i+2} \geq \phi^i$ with $\phi = (1 + \sqrt{5})/2$

Number of descendants

- Descendants = the node itself and children and their children etc
- A node with rank r has at least $F_{r+3} - 1$ descendants (full and hollow)
- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$
- For $r = 0$ it is the node itself and $F_{0+3} - 1 = 2 - 1 = 1$
- For $r = 1$ it the node itself plus one child and $F_{1+3} - 1 = 3 - 1 = 2$
- For $r \geq 2$ the node itself and its children with ranks $r - 2$ and $r - 1$ are among the descendants.
- By induction, and counting only the first two children the number of descendants is at least:
  $1 + (F_{r+2} - 1) + (F_{r+1} - 1) = (F_{r+2}) + (F_{r+1} - 1) = F_{r+3} - 1$
- That is, at least $F_{r+3} - 1$ descendants with rank r
- We will use this to find the maximum rank, r_{max} since we need an array with r_{max} elements

Maximum rank of a root with n descendants

- What can r be at most?
- $F_{i+2} \geq \phi^i$ with $\phi = (1 + \sqrt{5})/2$
- n nodes in a tree $\leftrightarrow$ n descendants of the root
- A node with rank r has at least $F_{r+3} - 1$ descendants
- $F_{r+3} - 1 \geq F_{r+2} \geq \phi^r$ so $n \geq \phi^r$ and $r \leq \log_\phi n$

Efficient moving of children and efficient links

- The children of a node are stored in the order of decreasing rank
- To move all except the first two children is therefore a constant time operation
- When the minimum element is removed we need to find roots with the same rank in constant time
- This is done using an array and the rank of a node as the index to the array.
- The first time you see a node with rank r it is stored in the array at index r
- The next time you see a node with rank r you can therefore find it in constant time
- Then you link and put back the new parent at index $r + 1$ and do a new link if any node already was stored at $r + 1$

Time complexities

- Recall: deleting a non-minimum element is a constant time operation
- N includes hollow nodes, and n is only full nodes
- Deleting the minimum element is done by destroying hollow roots and then doing links to reduce the number of roots to at most $\log N$
- To delete a hollow root and making its children new roots is a constant time operation
- The following can be shown:
  - The worst case time of all hollow heap operations except delete take constant time
  - The amortized time of delete (and delete-min) takes $O(\log N)$ on a heap with N nodes
- Thus: hollow heaps have constant time insert and reduce-key
- And array-based heaps instead have $O(\log n)$ insert and reduce-key
- If insert and reduce-key are frequent, hollow heaps can be faster

Version 2: One-root hollow heaps

- Allow links of nodes with different ranks
- By allowing this, it is possible to have only one root
- Now a child must be marked as coming either from a ranked or unranked link
- Either the heap is empty or the root is full (i.e. never a hollow root)
- When moving children of u to v, all the unranked children of u are always moved to v plus the ranked children as before (i.e. keep one or two children in u)

Version 3: Two-parent hollow heaps

- Instead of moving some children of u to v, v becomes a parent of u
- That is, v becomes a second parent of u
- Thus, the data structure is no longer a tree
- It becomes a directed acyclic graph, or a dag
- The heap order terminology is translated to dags
- A child must have a key which is at least as big as the key of any of its parents

Observations

- A node in a two-parent hollow heap has at most one parent if it is full, and at most two parents if it is hollow.
- Motivation: there are only two ways to get a parent:
  1. a full root can get a first parent by becoming a child at a link, and
  2. a full node can become hollow at a decrease-key and get a second parent
  3. a hollow node cannot become full and therefore not get any additional parent

Implementations

- Array-based
- Fibonacci heap
- Two-parent hollow heap
- Note insert and decrease_key (i.e. change_position in array-based)