Algorithms and Data Structures

Priority Queues

Ulf Leser

Specialized Queues: Priority Queues

- Up to now, we assumed that all elements are equally important and that any of them could be searched next
- What if some elements are more important than others?
  - In many applications, elements have a priority
  - Next access always retrieves the currently most important element
  - Accessed elements are "finished" – remove from list
- Data structures supporting such requirements are called Priority Queues
  - Difference to SOL: We know by which property we should sort: The priority; elements taken from the list are removed
  - Difference to a queue: New elements need to be placed such that the priority ordering is preserved

Simple Example

- Scheduler: Part of an OS which assigns computational resources (cores) to jobs (programs)
  - Assume a machine with one core / thread
  - 10 jobs should run concurrently
  - Time slicing: Give every job the core for some time, then next …
  - Fair: Every job gets 10% of the time
  - What about OS jobs, e.g., the scheduler itself?
- Often, assignments are not fair, but obey priorities
  - OS jobs get high priority
  - Users may assign priorities to their jobs (unix **`nice`**)
  - Users may pay for high priorities
  - Student's jobs get lower priorities than staff's jobs
  - Etc.

Scheduler and Priority Queue

- Scheduler may use a priority queue (PQ)
- Main operations: **`getNextJob(), putJob(Job, priority)`**
  - **`putJob`** inserts new job into queue at "right" position
  - **`getNextJob`** returns the job with currently highest priority
- Desirable: Both operations should be fast
  - Sorted array: O(1) for **`getNextJob`**, but O(n) for **`putJob`**
  - Unsorted array: O(1) for **`putJob`**, but O(n) for **`getNextJob`**
  - We'll get O(1) for **`getNextJob`** and O(log(n)) for **`putJob`**
- Note: This doesn't suffice for a scheduler
  - Using only a PQ would be extremely unfair – most jobs would never start because high-priority OS jobs never terminate

Second Example: Compression

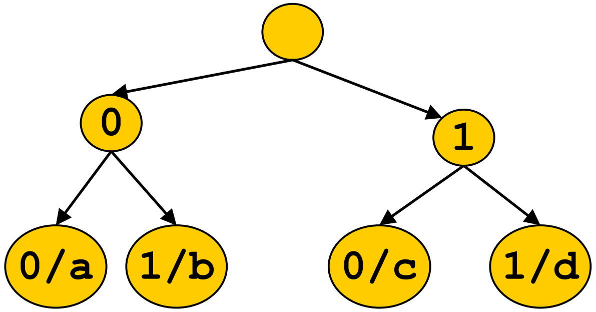
- Less data is usually better than more data
  - Less storage, faster to load, cheaper to transmit, …
- Compression: Represent much data D with few bits C
  - D: Message to be compressed, C: Compressed representation
  - Lossless: D can be reconstructed completely from C
  - Not lossless (lossy): jpeg, mpeg, …
- Example
  - D= "I will will that my will will will" (34 chars)
  - C= <1: will>; "I 1 1 that my 1 1 1" (19 chars + codebook)
    - Careful: Recognize "1" as codebook entry
- Popular idea: Use few bits for frequent substrings, and more bits for rare substrings
  - For instance used in ZIP and its variants

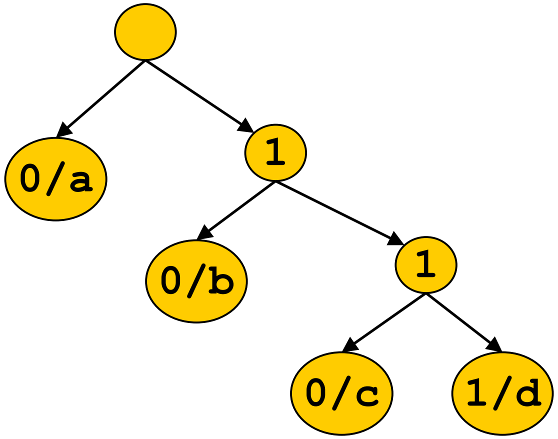
Huffman Codes

- Huffman coding: Optimal and efficient de-/compression
  - David A. Huffman, 1951 – as seminar thesis (!)
  - Compresses representation of characters, not substrings
  - Optimality: Least-space requiring code (under certain assumptions)
- Framework
  - Input message D
  - Compute optimal codebook B for all characters of D
    - Fewer bits for more frequent characters
  - Compress D into C using B
  - Transmit C and B
- Can easily be extended to compress n-grams

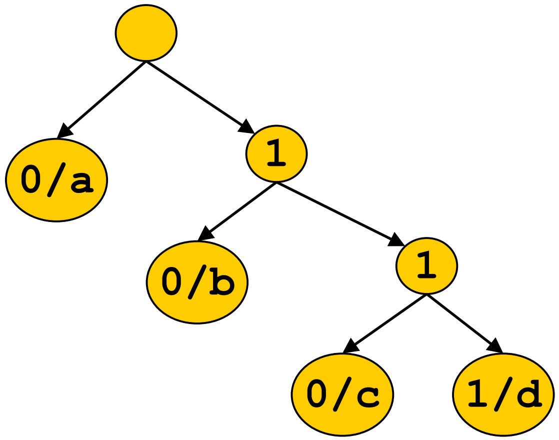
Approach

- We create a binary tree (will be defined precisely later)
  - Root is unlabeled
  - Every left child is labeled with 0, every right child with 1
  - Leaves are labeled with 0/1 and a character
  - All characters are represented as leaves



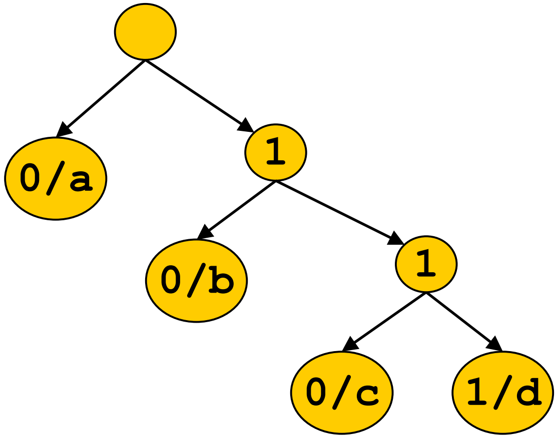


Compression



- D=aaaabaacaddaac;
  C=00001000110011111100110
  - Decompression is unique: Following the path from root to leave defines next character in D
  - Huffman codes are prefix-free: No code B(c) of a char c is prefix of the code B(c’) of a char c’ with c≠c’
    - Not prefix-free: B(a)=01, B(b)=011
- Compression?
  - |D| = 2*14=28 bits (assume equal length per char = 2 bit)
  - |C| = 23

Compression?



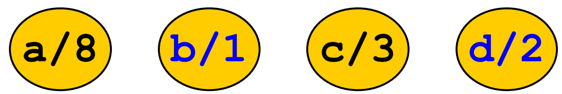
- D=addccdaadccbbd; C=0111111101101110011110…
- We only compress if frequent characters are represented with few bits
- Huffman coding: Which characters? How many bits? How frequent?

Algorithm

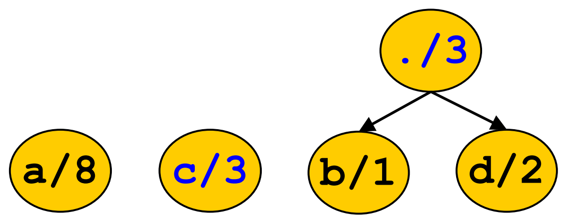
- Pre-processing: Count (relative) frequencies of all chars
- We build the tree bottom-up, first ignoring 0/1 labels
- Start with leaves, annotated with characters+frequencies
- Loop
  - Chose two least frequent nodes (chars) n, n'
    - If tie: Chose node with lowest subtree
  - Connect by new parent node p; freq(p) = freq(n)+freq(n')
  - Remove n, n' from further consideration (but leave in tree!)
- Until only two nodes remain
- Add root
- Label all left children with 0, all right children with 1

Example: D=aaaabaaccddaac

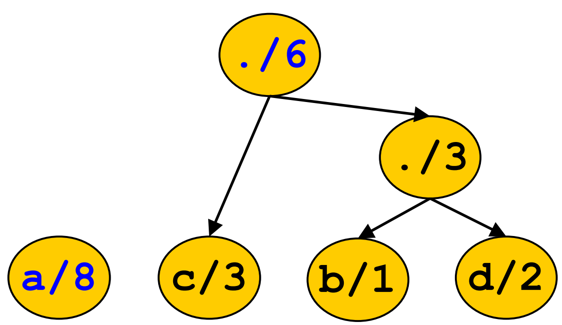
freq(a) = 8
freq(b) = 1
freq(c) = 3
freq(d) = 2

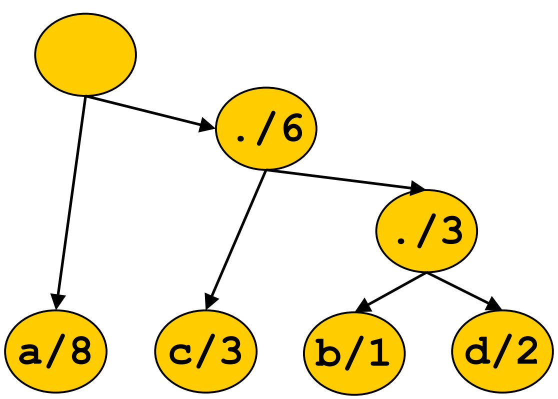


freq(a) = 8
freq(c) = 3
freq(p) = 3

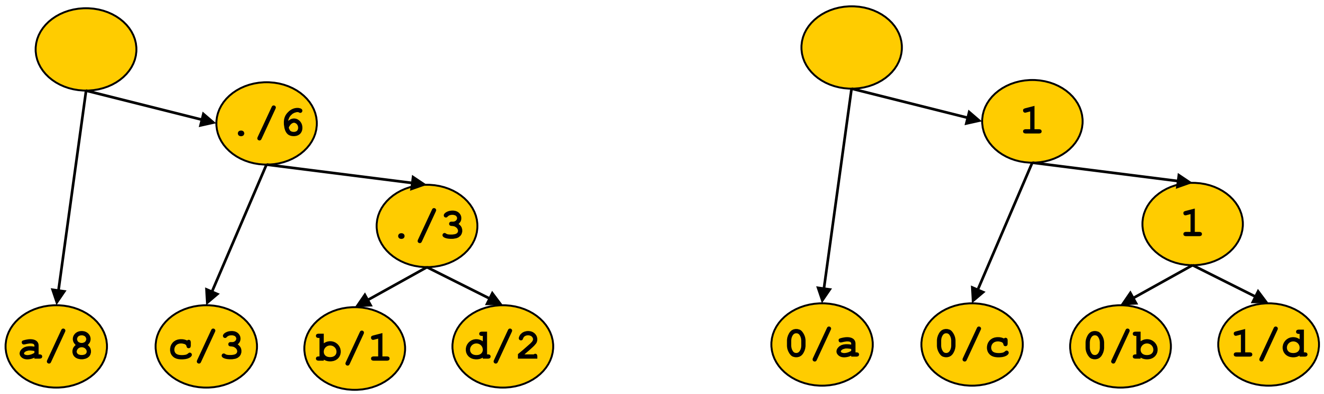


freq(a) = 8
freq(q) = 6





Example



- Code book B
  - B(a) = 0
  - B(c) = 10
  - B(b) = 110
  - B(d) = 111

Huffman and Priority Queues

- Complexity of computing the codebook
  - Let m=|Σ| and n=|D|
  - Preprocessing (freq counting): O(n)
  - Recall: A binary tree with m leaves has O(m) inner nodes
  - Every loop creates an inner node: O(m) iterations
    - Core: We need to find two nodes with smallest frequency
    - If nodes kept in sorted array: O(1), but inserting p will cost O(m)
    - If kept in unsorted linked list: O(m), but inserting p will cost O(1)
  - Anyway: $O(n+m^2)$
- Better: Use a priority queue for managing nodes
  - Yields O(1) for **`getInfrequentNodes`**, and O(log(m)) for **`putNode`**
  - Together: O(n+m*log(m))
    - One can actually get O(n+m)

Content of this Lecture

- Priority Queues
- Using Heaps
- Using Fibonacci Heaps

Priority Queues

- A (min) priority queue (PQ) is an ADT with 3 essential operations
  - **`add(o,v)`**: Add element o with priority (value) v
  - **`getMin()`**: Retrieve element with highest priority
  - **`removeMin()`**: Remove element with highest priority
- Typical additional operations
  - **`merge(p1,p2)`**: Merge two PQs into one
  - **`create(L)`**: Convert a list in a priority queue
  - **`delete(o)`**: Delete element o from PQ
  - **`update(o,v)`**: Change priority of element o to v

Maybe Arrays?

- Using a sorted array
  - **add** requires O(n) (bad)
    - We find the position in log(n), but then have to free a cell by moving all elements after this cell by one position
  - **getMin** requires O(1)
  - **deleteMin** requires O(n) (bad)
- PQs are typically used in applications where elements are inserted and removed (and updated) all the time
- We need an efficient DS that can change its size dynamically at very low cost while keeping a certain order (min element)

Content of this Lecture

- Priority Queues
- Using Heaps
  - Heaps
  - Operations on Heaps
  - Heap Sort
- Using Fibonacci Heaps

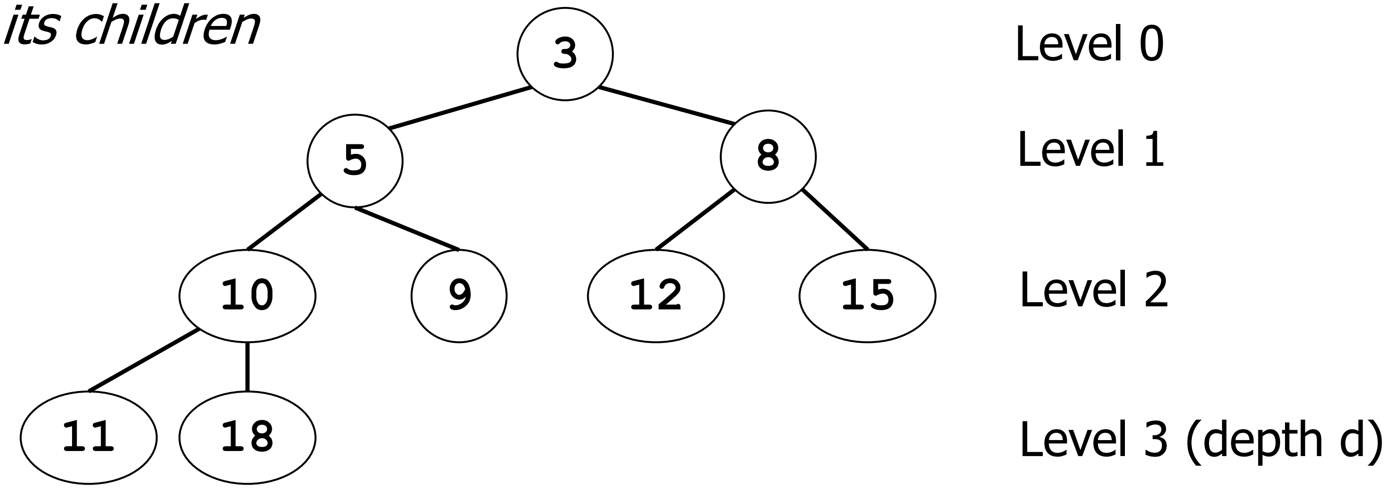
Heap-based PQ

- Can we find a way to keep a list "a little sorted"?
  - We only need the smallest element at a fixed position
  - All other elements can be at arbitrary places
  - But **`add/deleteMin`** should be faster than O(n)
- One such structure is called a heap

Heaps

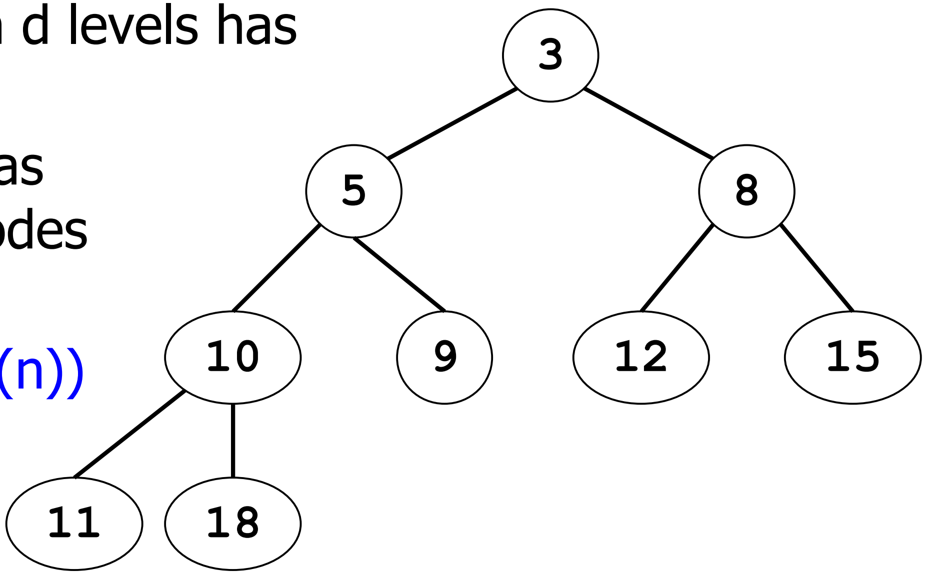
- Definition

  A heap is a labeled binary tree of depth d for which the following constraints holds
  - *Nodes are labeled with integers (the priorities)*
  - *Form-constraint (FC): The tree is complete except the pre-last level*
    - *I.e.: Every node at level l<d-1 has exactly two children*
  - *Heap-constraint (HC): The label of every node is smaller than that of all its children*



Properties

- Order
  - A heap is "a little" sorted: We know the smallest element (root)
  - We know the order for some pairs of elements (parent-successors), but for many pairs we don't know which is bigger
    - E.g. nodes at the same level
- Size
  - A complete binary tree with d levels has $2^{d+1}-1$ nodes
  - A heap with d levels thus has between 2^d-1 and $2^{d+1}-1$ nodes
  - A heap with n nodes has $\text{ceil}(\log(n+1))-1 \in O(\log(n))$ levels

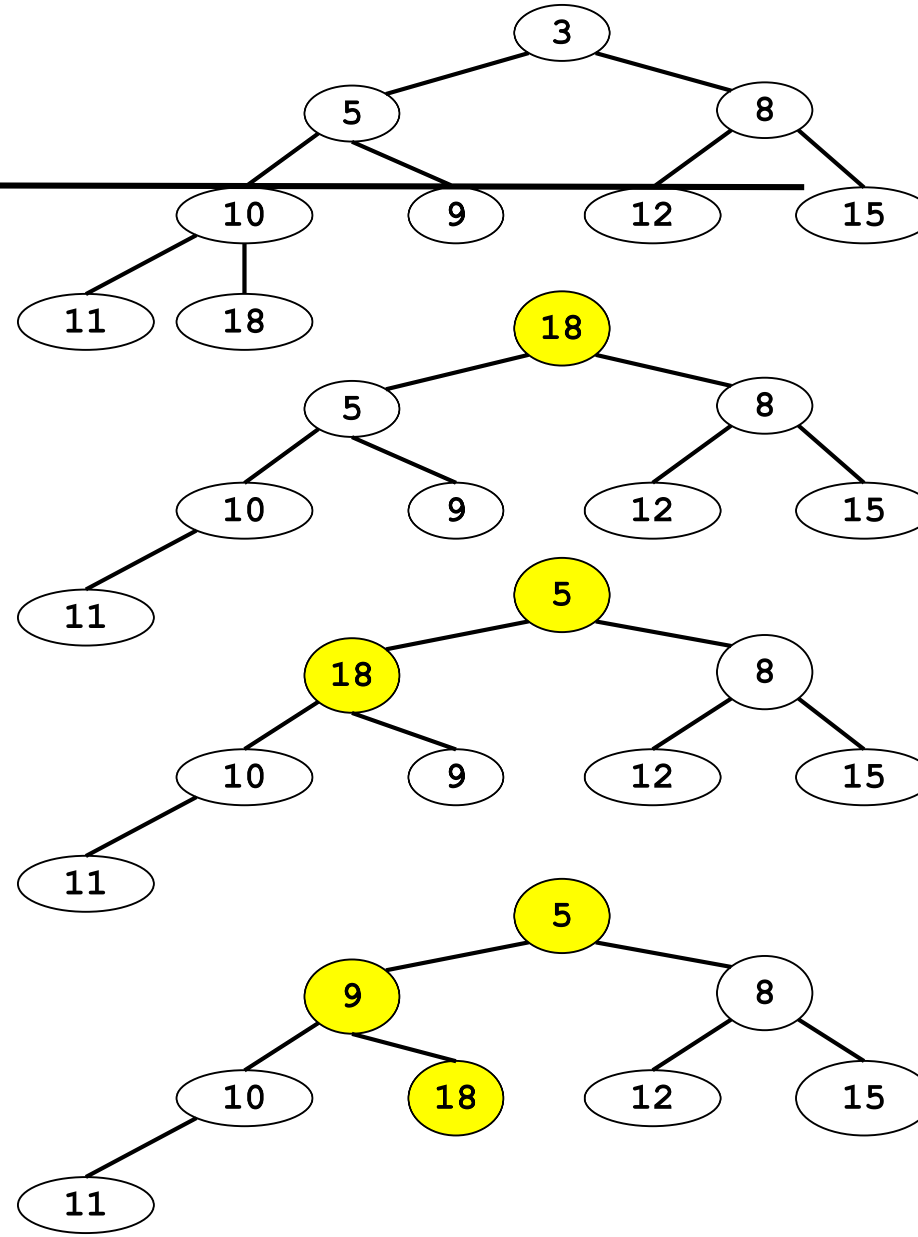


Operations

- Assume we store our PQ as a heap
- Clearly, `getMin()` is possible in O(1)
  - Keep a pointer to the root
- But …
  - How can we cheaply perform `deleteMin()` – such that the new structure again is a heap?
  - How can we cheaply add an element to a heap – such that the new structure again is a heap?
  - How can we cheaply create a heap – from a given list?

DeleteMin()

- We first remove the root
  - Creates two heaps
  - We must connect them again
- We take the „last" node, place it in root, and "sift" it down the tree
  - Last node: right-most in the last level (actually, we can take any from the last level)
  - Sifting down: Exchange with smaller of both children as long as at least one child is smaller than the node itself



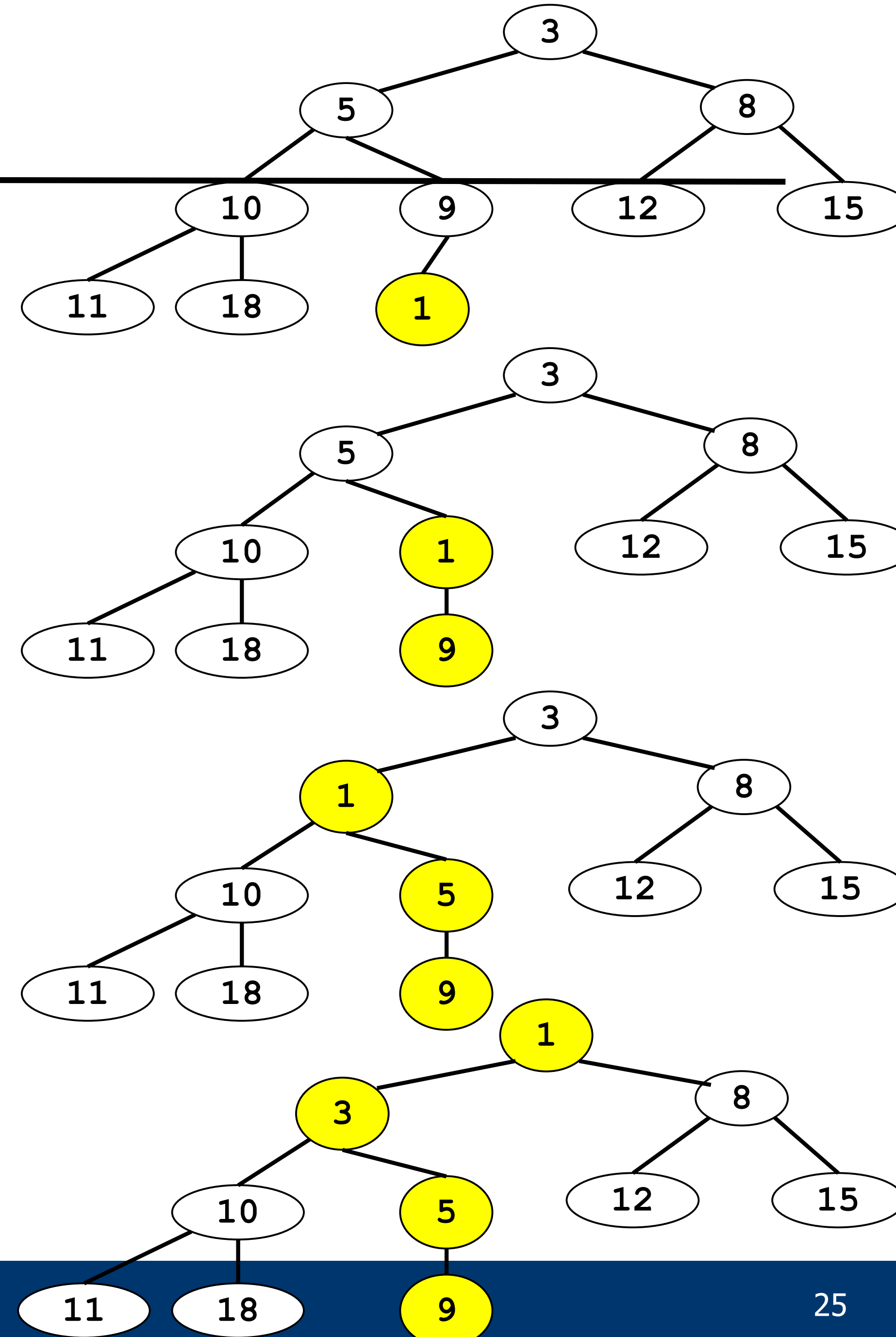
Analysis - Correctness

- We need to show that FC and HC still hold
- HC: Look at the tree after we choose new root k. k may
  - … be smaller than its children. Then HC holds and we are done
  - … be larger than at least one child k2. Assume that k2 is the smaller of the two children (k1, k2) of k. We next swap k and k2. The new parent (k2) now is smaller than its children (k1, k), so the HC holds
  - After the last swap, k has no children – HC holds and we are done
- FC: We remove one node, then we sift down
  - Removing last node doesn't affect FC as we remove in the last level
  - Sifting does not change the topology of the tree (we only swap)

Analysis - Complexity

- Recall that a heap with n nodes has O(log(n)) levels
- During sifting, we perform at most one comparison and one swap in every level
- Thus: WC is in O(log(n))

Add() on a Heap



- Cannot simply add on top
- Idea: We add new element somewhere in last level and sift up
  - We might need a new level
  - Sifting up: Compare to parent and swap if parent is larger

Analysis

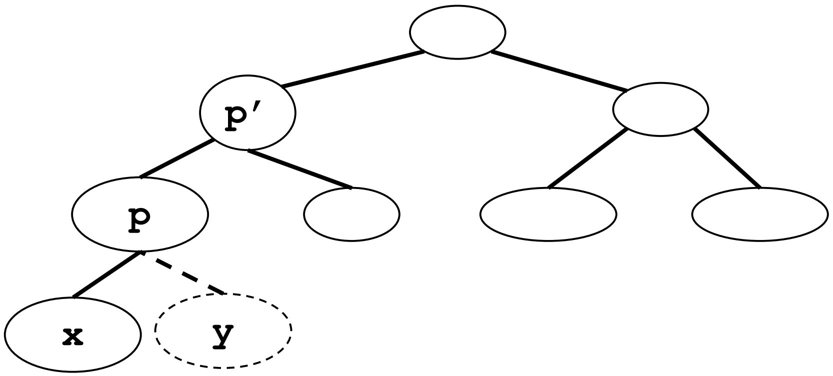
- Correctness
  - HC
    - If parent has only one child, HC holds after each swap
    - Assume a parent k has children k1 and k2, k2 was swapped there in the last move, and k2<k. Since HC held before, k<k1, thus k2<k<k1. We swap k2 and k, and thus the new parent is smaller than its children. On the other hand, if k2≥k, HC holds immediately (and we don't swap).
  - FC: See **`deleteMin()`**
- Complexity: O(log(n))
  - See **`deleteMin()`**

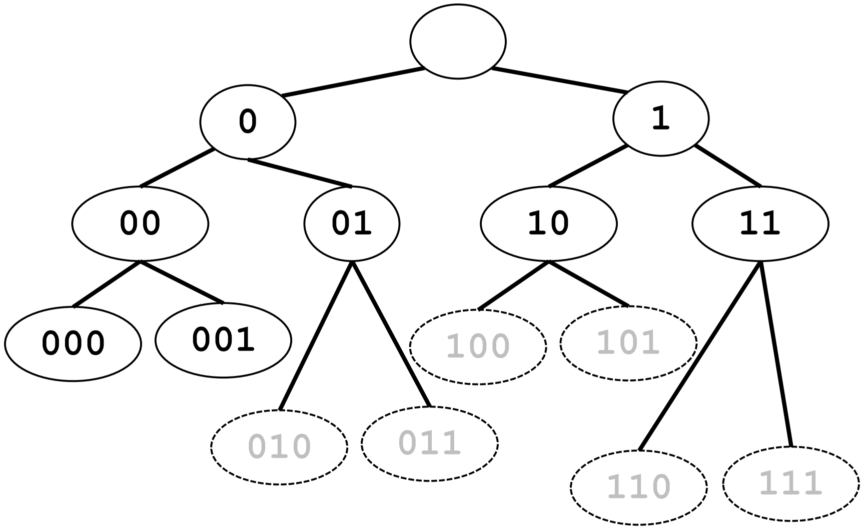
How to Find the Next Free / Last Occupied Node

- What do we need to find?
  - For **`deleteMin`**, we use the right-most leaf on the last level
  - For **`add`**, we add the leaf right from the last leaf (or new level)
  - Note: We actually need the parent node k
- How to get there?
  - We can compute in O(1) the index c(x) of the last leaf x in the last level: c = n – 2^(floor(log(n)))
    - Or log(n+1) for **`add`**
  - Fast trick for accessing the parent node p of x: Perform tree traversal from root using the binary representation of p as guide

Illustration

- For **`deleteMin`**, we need x; for **`add`**, we need y
  - c(x)=0, p(y)=1
  - Binary: 000, 001
  - Bitstring length is depth d of tree
- Go through bitstring from left-to-right and through tree from top to bottom
  - Next bit=0: Go left
  - Next bit=1: Go right
- Allows finding x/p in O(log(n))





Creating a Heap

- We start with an unsorted list with n elements
- Naïve: Start with empty heap and perform n additions
  - Obviously O(n*log(n))
- Better: Bottom-Up-Sift-Down
  - Build a "naïve" tree fulfilling the FC (but not HC)
    - Simple fill a tree level-by-level – this is in O(n)
  - Sift-down all nodes on the second-last level
  - Sift-down all nodes on the third-last level
  - …
  - Sift down root

Analysis

- Correctness
  - After finishing one level, all subtrees starting in this level are heaps because sifting-down ensures FC and HC (see **`deleteMin()`**)
  - Thus, when we are done with the first level (root), we have a heap
- Analysis
  - We look at the cost per level h (h∈{0, … ,d-1})
  - At every level h≠d, there are 2^h nodes
    - For nodes at level d, we don't do anything
  - For every node at level h, we need at most d-h swaps
  - This yields

$$T(n) = \sum_{h=0}^{d-1} 2^h * (d-h) =$$

$$\sum_{h=0}^{d-1} h * 2^{d-h} = 2^d * \sum_{h=0}^{d-1} \frac{h}{2^h} \leq n * \sum_{h=0}^{\infty} \frac{h}{2^h} = n * 2 \in O(n)$$

Analysis ??? NEU

- Correctness
  - After finishing one level, all subtrees starting in this level are heaps because sifting-down ensures FC and HC (see **`deleteMin()`**)
  - Thus, when we are done with the first level (root), we have a heap
- Analysis
  - We look at the cost per level h ($h \in \{0, \dots, d-1\}$)
  - At every level $h \neq d$, there are 2^h nodes
    - For nodes at level d, we don't do anything
  - For every node at level h, we need at most d-h swaps
  - This y

$$T(n) = \sum_{h=0}^{d-1} 2^h * (d-h) =$$

$$\sum_{h=0}^{d-1} (h+1) * 2^{(d-h-1)} = 2^d * \sum_{h=1}^{d} \frac{h}{2^h} \leq n * \sum_{h=0}^{\infty} \frac{h}{2^h} = n * 2 \in O(n)$$

Summary

	Linked list	Sorted linked list	Heap
getMin()	O(n)	O(1)	O(1)
deleteMin() (after getMin())	O(1)	O(1)	O(log(n))
add()	O(1)	O(n)	O(log(n))
merge()	O(1)	$O(n_1+n_2)$	$O(log(n_1)*log(n_2))$
create()	O(n)	O(n*log(n))	O(n)
Space	O(n) add. pointer	O(n) add. pointer	O(n) add. pointer

Heaps can be kept efficiently in an array – no extra space, but limit to heap size

Side Note: Heap Sort

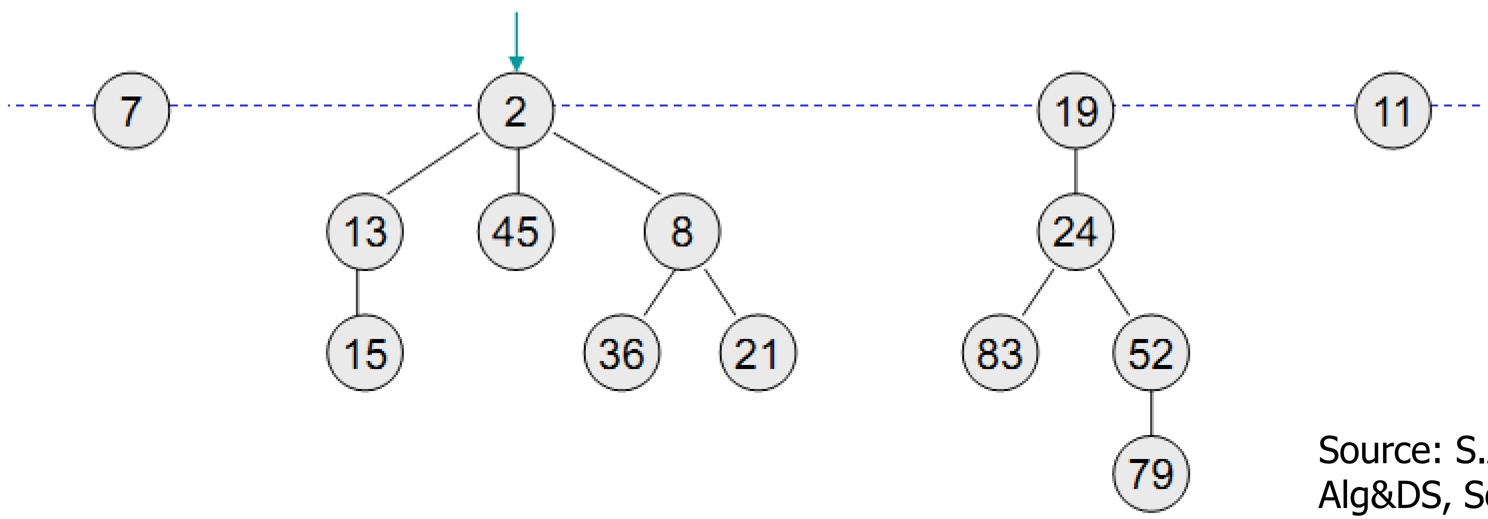
- Heaps also are a suitable data structure for sorting
- Heap-Sort (a classical sorting algorithm)
  - Given an unsorted list, first turn it into a heap (O(n))
  - Repeat
    - Take the smallest element and store in array in O(1)
    - Remove smallest element in O(log(n)) (**`deleteMin()`**)
  - Until heap is empty – after n iterations
- This runs in O(n*log(n))
- Can be implemented in-place when heap is stored in array
  - See [OW93] for details
- Note: Empirically, heap-sort is slower than quick-sort

Content of this Lecture

- Priority Queues
- Using Heaps
- Using Fibonacci Heaps

Fibonacci-Heaps (very rough sketch)

- A Fibonacci Heap (FH) is a forest of (non-binary) heaps with disjoint values
  - All roots are maintained in a double-linked list
  - Special pointer (`min`) to the smallest root
  - Accessing this value (`getMin()`) obviously is O(1)



Source: S.Albers, Alg&DS, SoSe 2010

Maintainance of a FH

- FHs are maintained in a lazy fashion
  - `add(v)`: We create a new heap with a single element node with value v. Add this heap to the list of heaps; adapt min-pointer, if v is smaller than previous min
    - Clearly O(1)
  - `merge()`: Simply link the two root-lists and determine new min (as min of two mins)
    - Clearly O(1)
- Deleting an element (`deleteMin()`) needs more work
  - Until now, we just added single-element heaps
  - Thus, our structure after n `add()` is an unsorted list of n elements
  - Finding the next min element after `deleteMin()` in a naïve manner would require O(n)

deleteMin() on FH

- Method
  - We first remove the min element
  - We then go through the root-list and merge pairs of heaps with the same rank (=# of children) until all heaps have different ranks
  - Merging two heaps in O(1): (1) Find the heap with the smaller root value; (2) Add it as child to the root of the other heap
- But analysis is fairly complicated
  - The above method is O(n) in worst case
    - But after every clean-up, the root-list is much smaller than before
    - Subsequent clean-ups need much less time
  - Amortized analysis shows: Average-case complexity is O(log(n))
  - Analysis depends on the growth of the trees during merge – these grow as the Fibonacci numbers

Disadvantage

- Though faster on average, Fibonacci Heaps have unpredictable delays
- No log(n) upper bound for every operation
- Not suitable for real-time applications etc.

Summary

	Linked list	Sorted linked list	Heap	Fibonacci Heap
getMin()	O(n)	O(1)	O(1)	O(1)
deleteMin()	O(1)	O(n)	O(log(n))	O(log(n))*
add()	O(1)	O(n)	O(log(n))	O(1)
merge()	O(1)	$O(n_1+n_2)$	O(log(n))	O(1)
create()	O(n)	O(n*log(n))	O(n)	O(n)

*: Amortized analysis

Exemplary Questions

- The PQ we described is a MinHeap. Describe insert and getMin() operations for a maxHeap, wheren a parent node must always be larger than ist children.
- Describe an algorithm for searching an arbitrary key in a MinHeap. Analyze the WC complexity. Also analyze the AC, assuming that the key being searching is contained in the PQ.
  - Searching keys is, for instance, necessary to change priorities
- What is the complexity of searching thr k-smallest element in a MinHeap?
- Descrbe an algorithm that merges two minHeaps in $O(\log(n_1)*\log(n_2))$, where n_1, n_2 are the sizes of the original heaps.