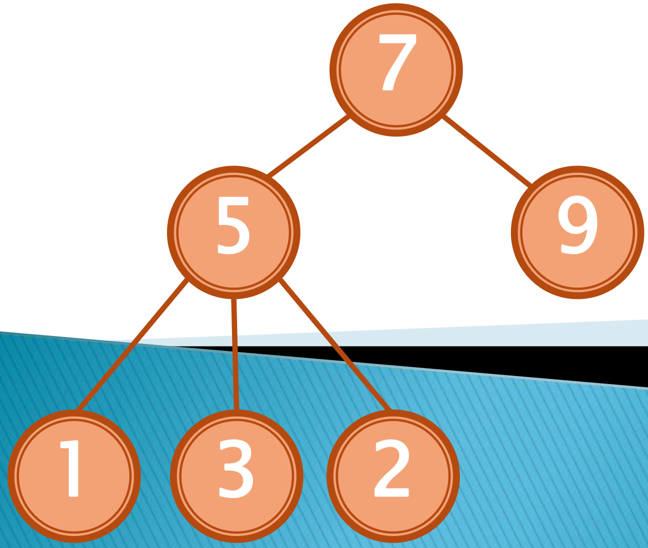
Efficient Data Structures



Summer 2014
Paweł Gawrychowski
Mayank Goswami
Patrick Nicholson

About the course: Marking, etc.

- This is a 9 credit point course: 2+2

- Prerequisites: Basic course in data structures
  - You *should* know asymptotic analysis $(O, o, \Theta, \Omega, \omega)$
  - You *should* know about linked lists/balanced trees
  - You *should* know at least one programming language
    - ADA DOC ASM AWK BASH BF C C# C++ 4.3.2 C++ 4.0.0-8 C99 strict CLPS CLOJ LISP sbcl LISP clisp D ERL F# FORT GO HASK ICON ICK JAR JAVA JS LUA NEM NICE NODEJS CAML PAS fpc PAS gpc PDF PERL PERL 6 PHP PIKE PS PRLG PYTH 2.7 PYTH 3.2.3 PYTH 3.2.3 n RUBY SCALA SCM guile SCM qobi SED ST TCL TECS TEXT WSPC

- Marking scheme:
  - 60% exam
  - 30% homework sheets (must get 50% on homework)
  - 10% project (research/survey/implementation)
    - Groups of up to 3 people; more details will follow

About the course: Marking, etc.

- We will have weekly homework sheets
  - Each homework sheet will have
    - Theory problems (i.e., proofs)
    - Programming problems (at most 20% of homework)
      - These are to be submitted on SPOJ
      - See homework sheet for details

- We will also have weekly tutorials
  - Each tutorial review the previous week's assignment
  - You must actively participate in the tutorial sessions

About the course: Short Outline

- Models of computation
- Implicit Data Structures (Comparison)
  - Membership (Dictionary) Problem, Multikey Search
- Succinct Data Structures (Word-RAM)
  - Static problems: rank/select, trees, graphs, etc.
  - Cell Probe lower bounds for succinct data structures
  - Discussion of dynamic memory models
- Static predecessor searching (Word-RAM)
- Making data structures dynamic
- Persistence and applications (Pointer Machine/Word-RAM)
- Lower bounds (Comparison, Pointer Machine, Cell-probe, etc.)
- Introduction to the External Memory (I/O) Model
  - classic data structures: B-trees, Buffer trees.
- Efficient data structures in external memory
  - Generalizing word-RAM structures to the I/O model
  - Lower bounds on external memory data structures

Pat

Paweł

Mayank

Let's get Philosophical

- Why do we do algorithm analysis?
  - What are the goals?
    - Compare different algorithms
    - Determine which algorithm to use in which case
  - What is the end result of the analysis?
    - Input: an algorithm and some input parameters
      - We want a number: lower better than higher
- How do we do the analysis?
  - Computers are very complicated
  - Instead we analyse simpler *models of computation*

What model to use?

- There are *many* different models
  - Comparison-based, Word-RAM, Cell-Probe, I/O, Pointer machine, Cache-oblivious, etc.

- It is important to understand the limitations
  - This helps with understanding practicality
  - Models often focus on one particular aspect
  - We will discuss cases where it can be misleading

- Example: Sorting

IMPLICIT DATA STRUCTURES

Summer 2014

Efficient Data Structures

Patrick Nicholson

Space Efficiency

- Why do we care about space efficiency?
  - Practical reasons:
    - In many computations the limiting factor is memory
    - The memory hierarchy
    - Saving even a small constant factor in space means big money
    - Many computing devices often have less memory resources:
      - Smartphones
      - Microcontrollers
      - Sensors
      - Facebook enabled toaster

Space Efficiency

- Why do we care about space efficiency?
  - Theoretical Reasons:
    - Answer fundamental questions about computation:
      - "How much extra space do we need to answer queries about data?"
      - "Can we compress data and still answer questions about it?"
      - "Which types of queries are impossible to efficiently support?"
      - "Are pointers necessary?"
    - It is fun ☺

Implicit Data Structures

- What is the model?
  - Basic Idea: data is stored in an array $A[1..n]$
    - The "structure" consists of the order of the data
    - A "pointer" is just an integer in $A[1..n]$
  - Only need to know the value n
    - AKA: strict implicit data structure
    - Another option: $O(1)$ extra data allowed
  - Only allowed to make comparisons:
    - $a < b, a = b, a > b$
- Comments?

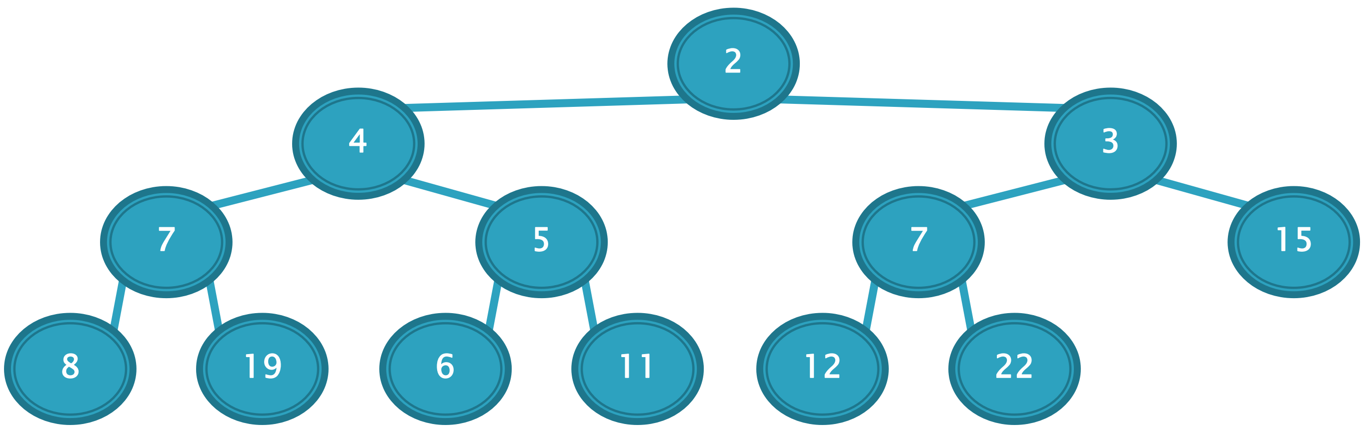
Implicit Data Structures

- You probably already know one…
- Heaps perform the following operations:
  - Insert(x): add key x
  - Delete-Min(): delete and return the smallest key
  - Get-Min(): return the smallest key

  - Insert(x) and Delete-Min() take $\Theta(\log n)$ time
  - Get-Min() takes $\Theta(1)$ time

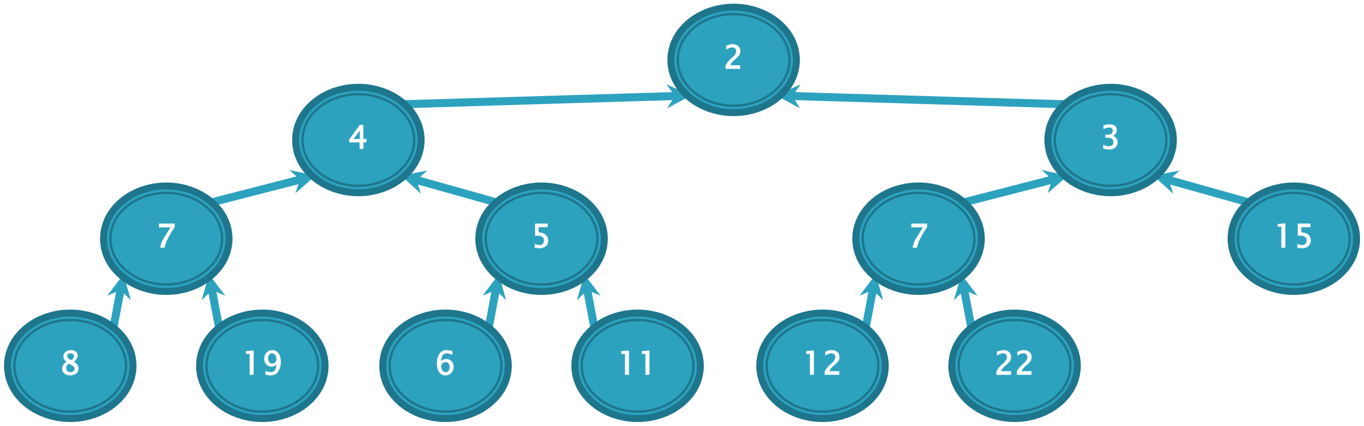
Binary Heap

- Heap Properties:
  - Complete binary tree except for the last level
  - Each node's key is at least as small as its children's



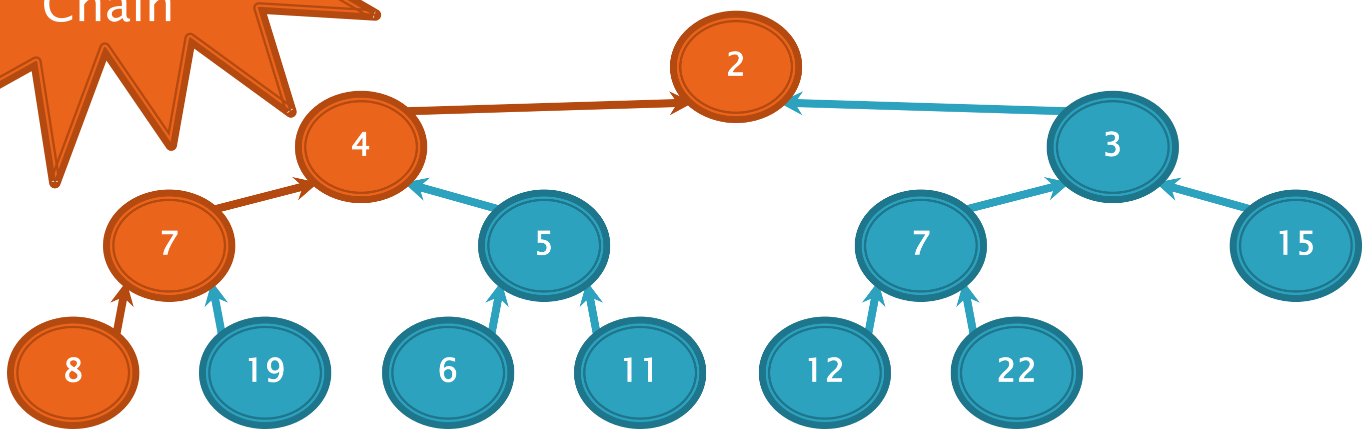
Partial Orders

- The heap structure is a partial order
  - A partial order is a binary relation that is:
    - Reflexive, Antisymmetric, and Transitive
  - Think of a directed acyclic graph with/without shortcuts



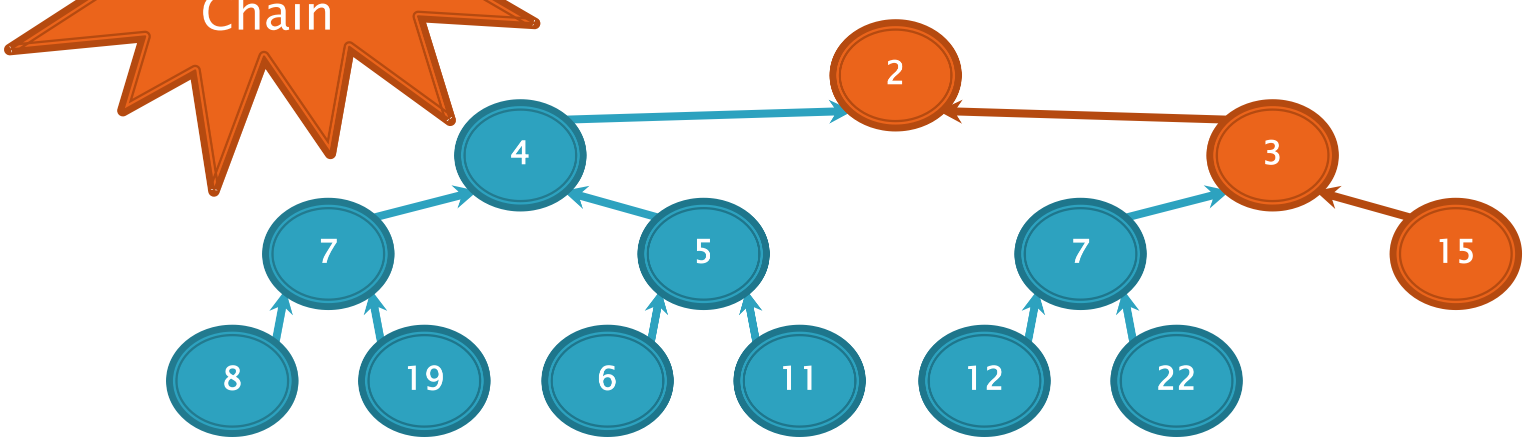
Partial Orders

- The heap structure is a partial order
  - A partial order is a binary relation that is:
    - Reflexive, Antisymmetric, and Transitive
    - a directed acyclic graph with/without shortcuts



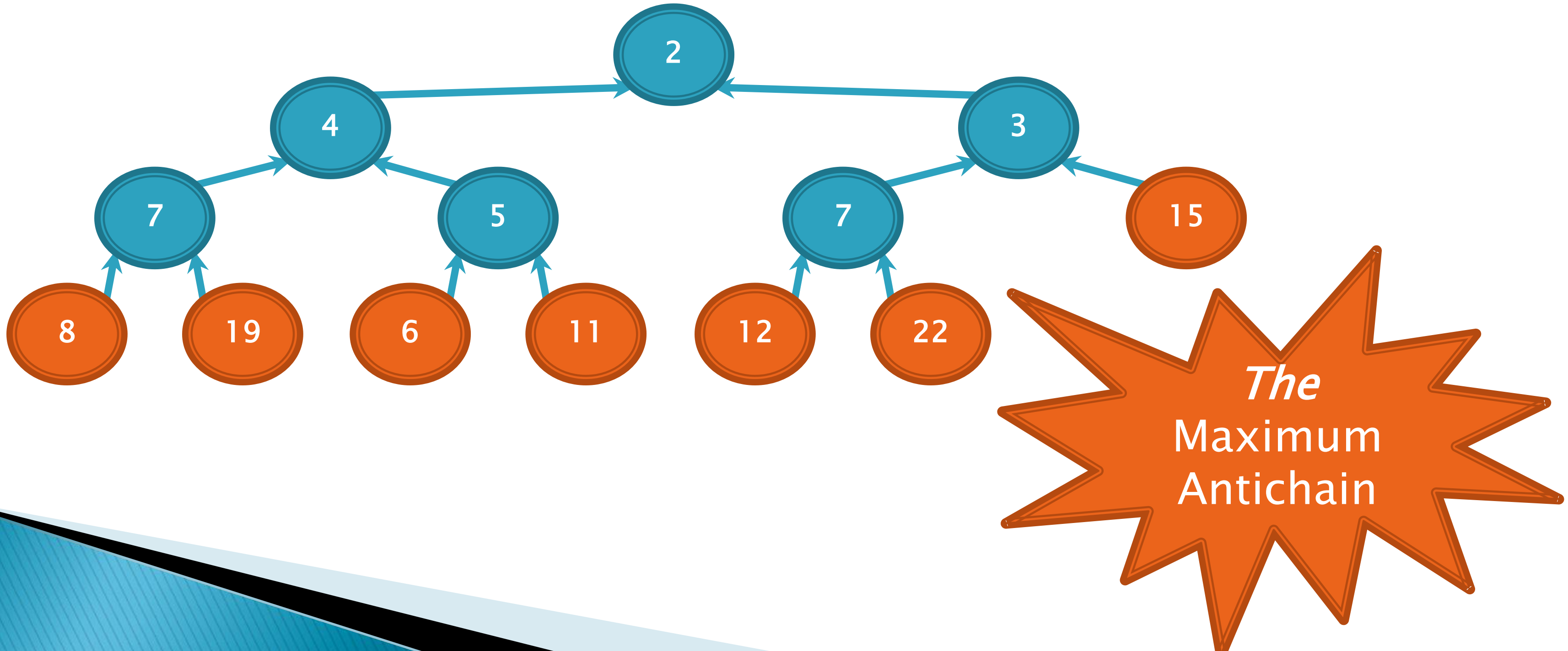
Partial Orders

- The heap structure is a partial order
  - A partial order is a binary relation that is:
    - Reflexive, Antisymmetric, and Transitive
  - … a directed acyclic graph with/without shortcuts



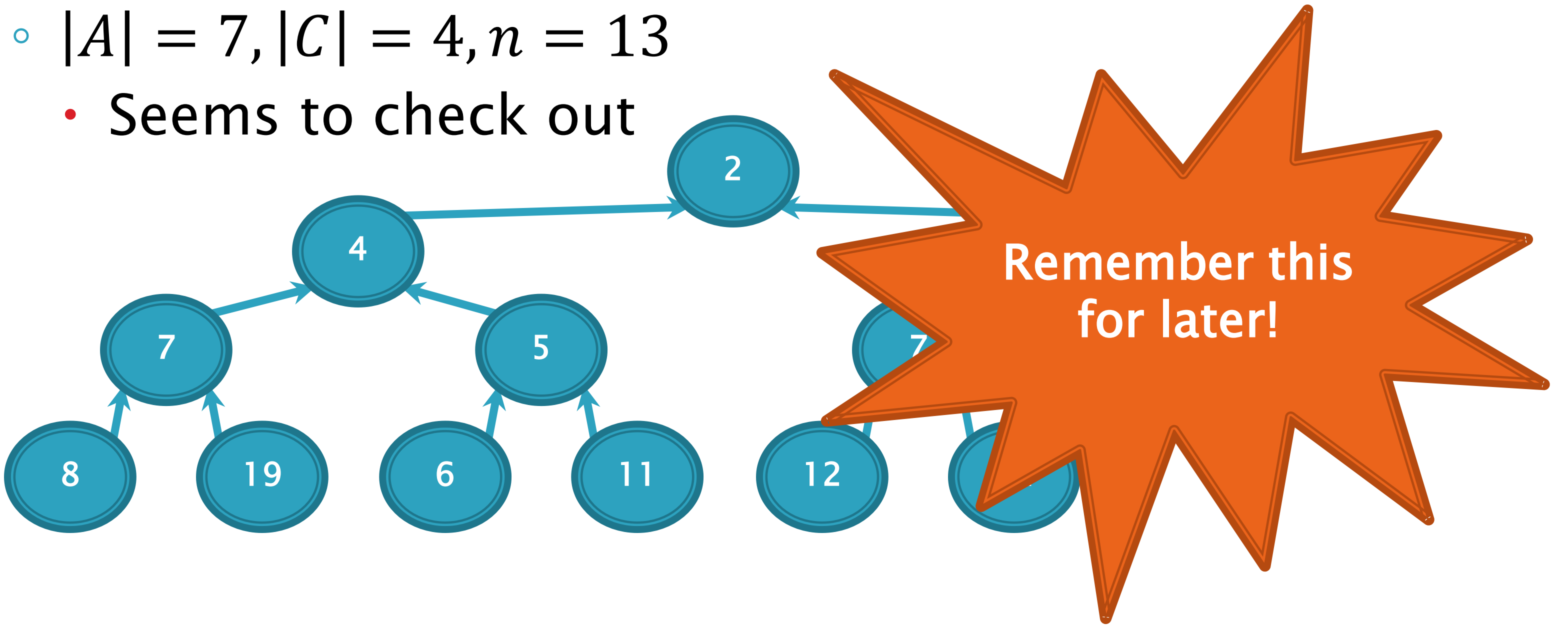
Partial Orders

- The heap structure is a partial order
  - A partial order is a binary relation that is:
    - Reflexive, Antisymmetric, and Transitive
  - Think of a directed acyclic graph with/without shortcuts



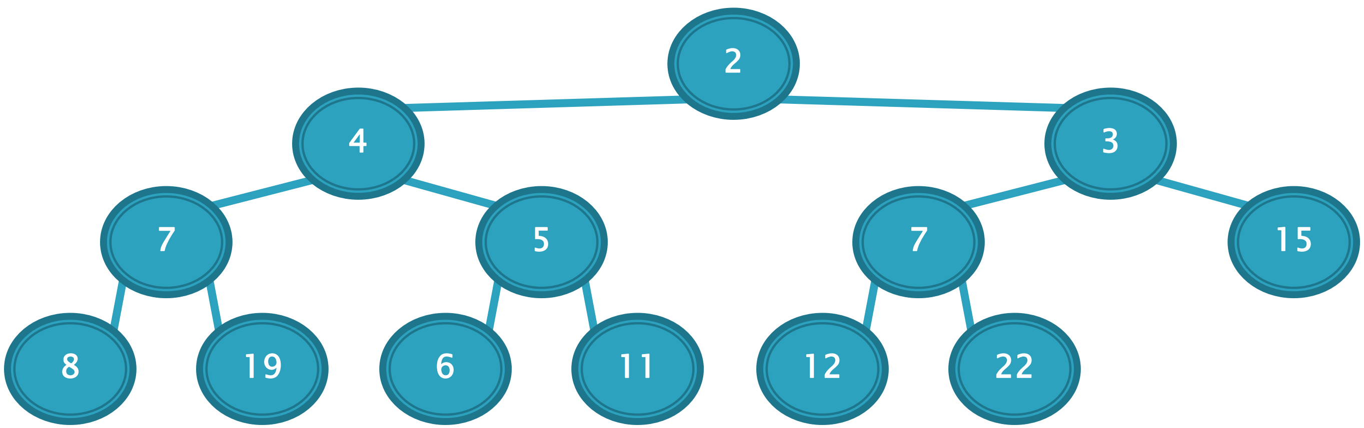
Partial Orders (Cont')

- Let C and A be *maximum* chain and antichain
- **Dilworth's Lemma**: Given an arbitrary partial order on n elements the product $|C| \times |A| \geq n$
  - $|A| = 7, |C| = 4, n = 13$
    - Seems to check out



Back to the Binary Heap

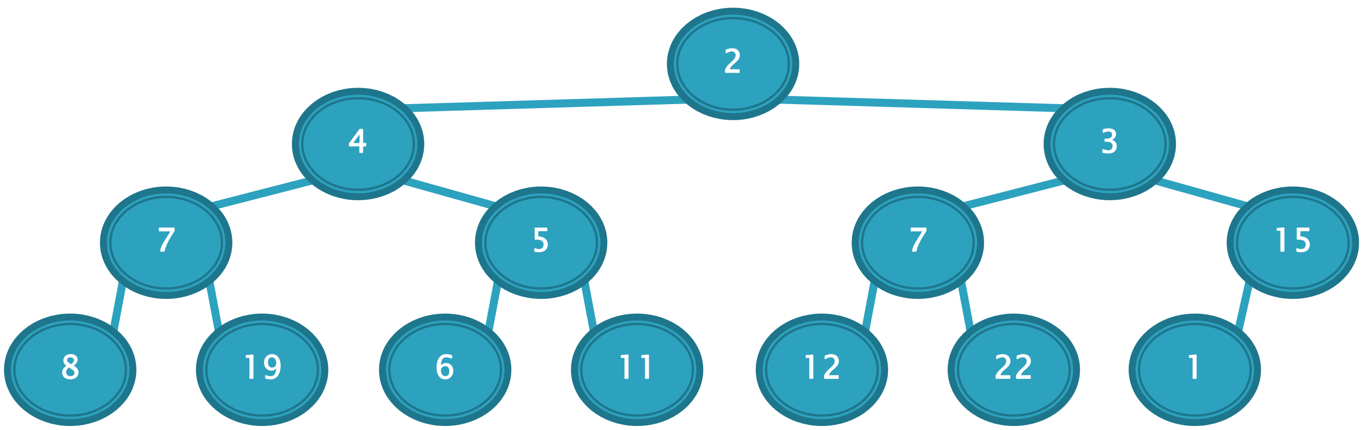
- Heap Embedding:
  - Left-child of node $i = 2i$
  - Right-child of node $i = 2i + 1$
  - Parent of $i = \lfloor i/2 \rfloor$



A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
13	2	4	3	7	5	7	15	8	19	6	11	12	22			

Binary Heap

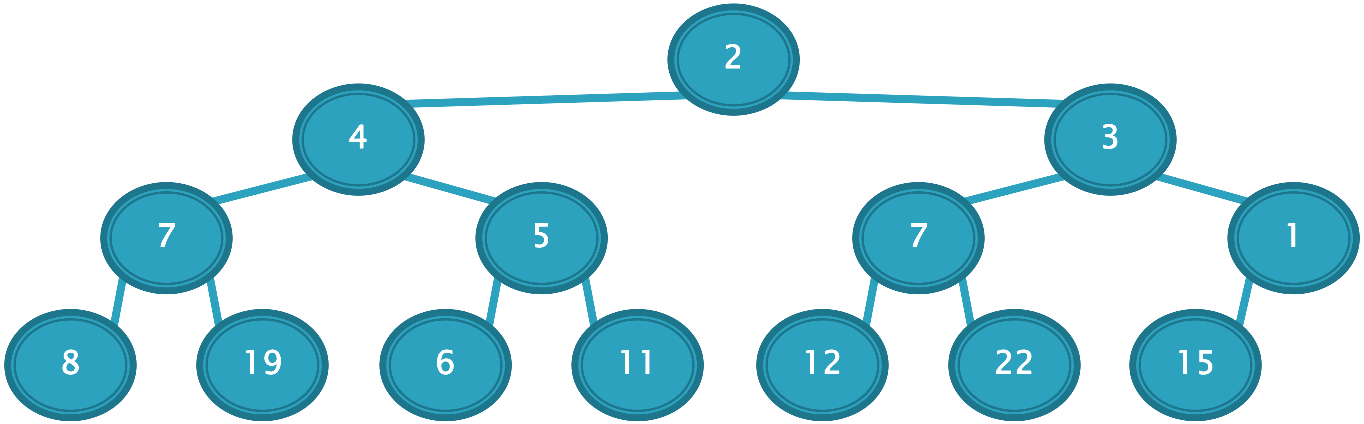
- Insertion



A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
14	2	4	3	7	5	7	15	8	19	6	11	12	22	1		

Binary Heap

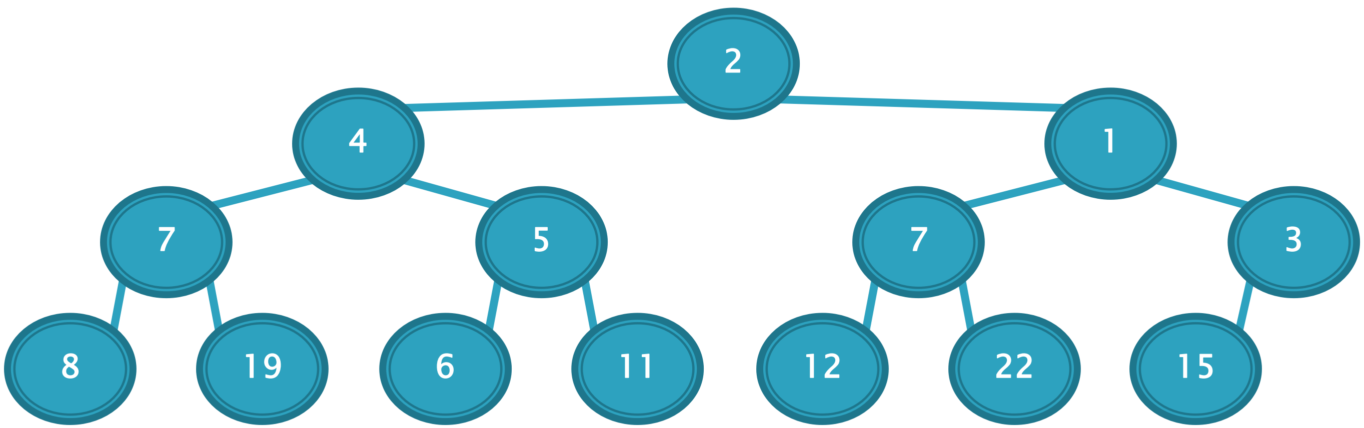
- Insertion



A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
14	2	4	3	7	5	7	1	8	19	6	11	12	22	15		

Binary Heap

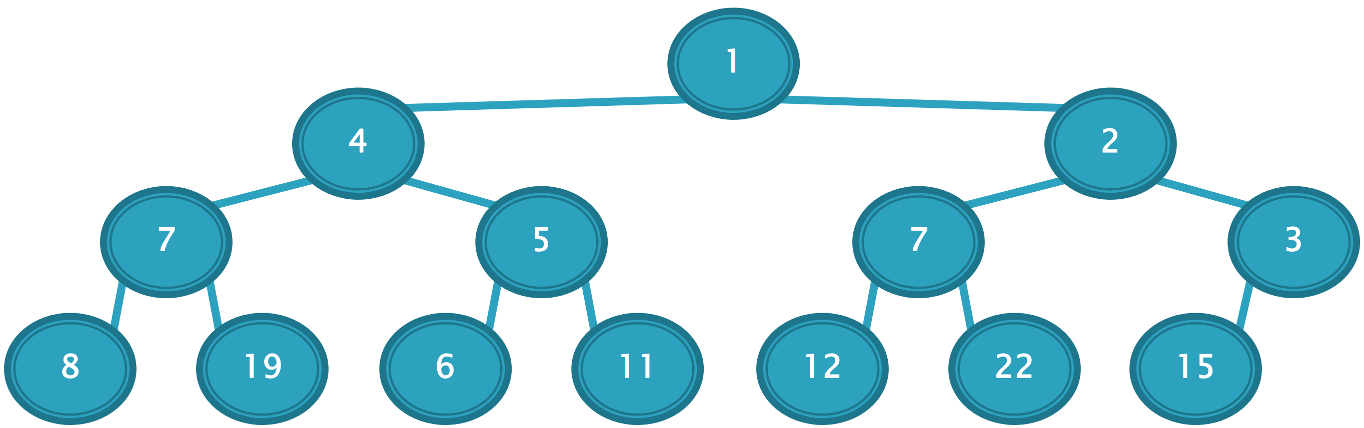
- Insertion



A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
14	2	4	1	7	5	7	3	8	19	6	11	12	22	15		

Binary Heap

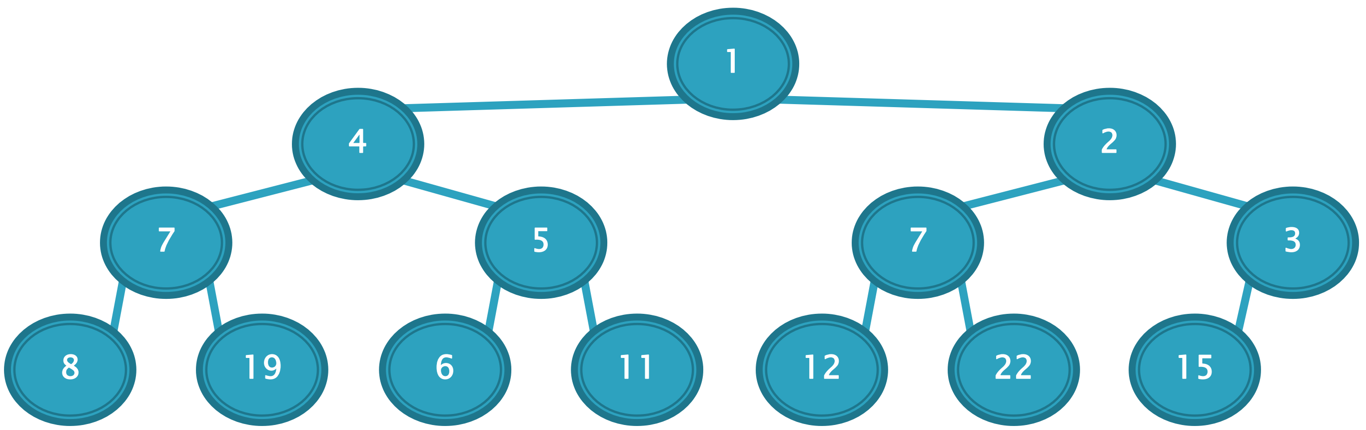
- Insertion



A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
14	1	4	2	7	5	7	3	8	19	6	11	12	22	15		

Binary Heap

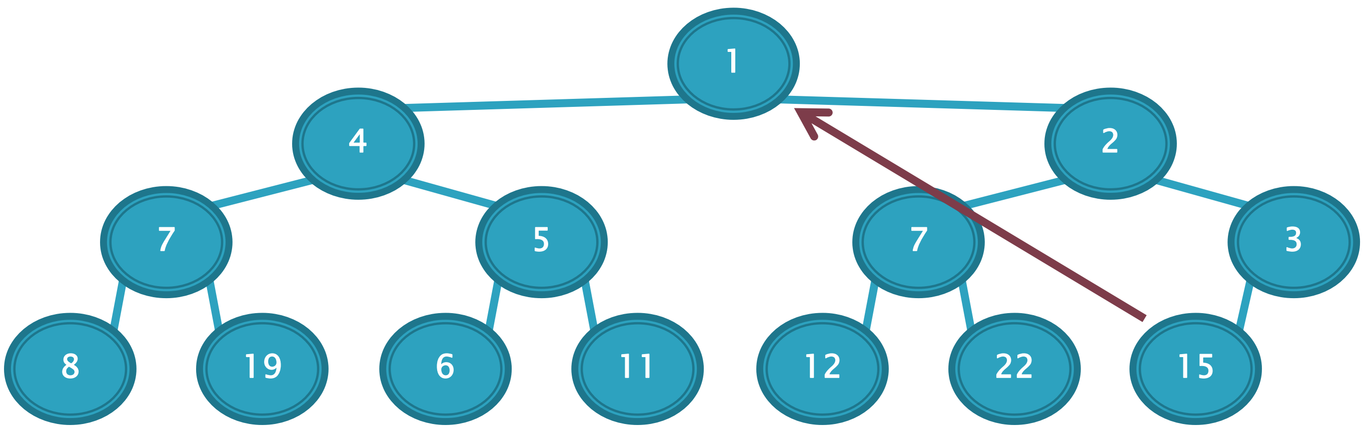
- Deletion



A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
14	1	4	2	7	5	7	3	8	19	6	11	12	22	15		

Binary Heap

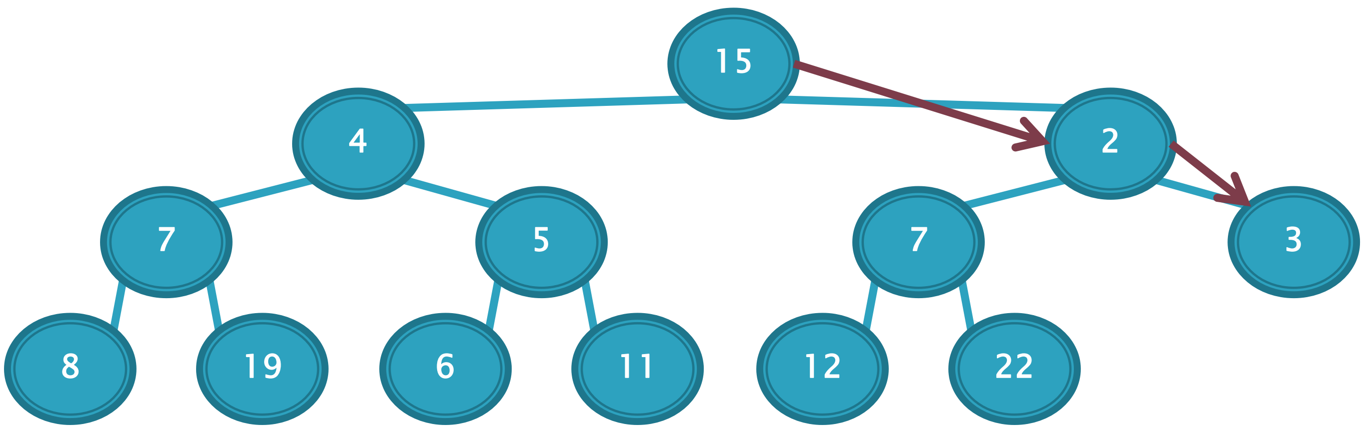
- Deletion



A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
14	1	4	2	7	5	7	3	8	19	6	11	12	22	15		

Binary Heap

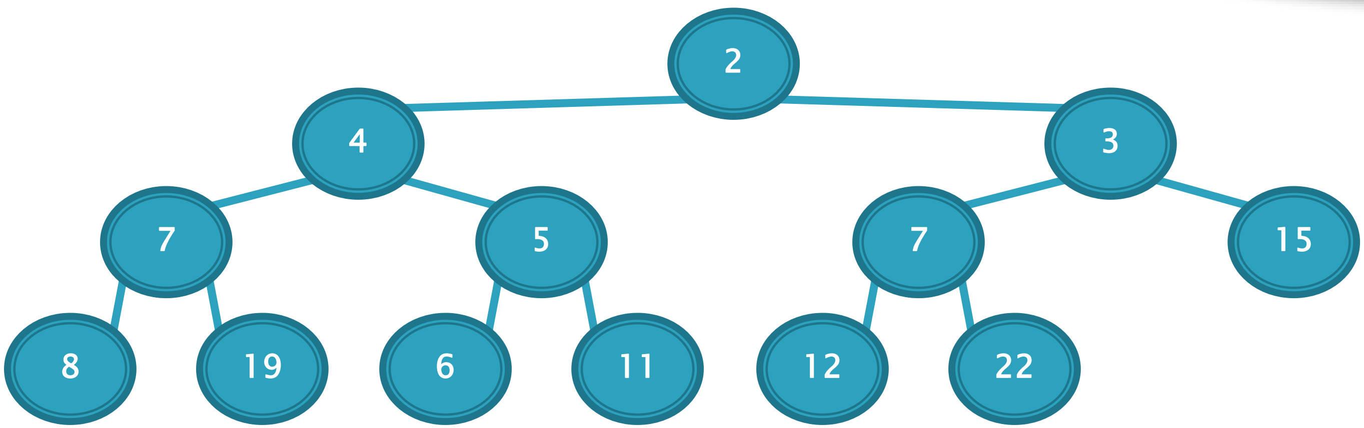
- Deletion



A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
13	15	4	2	7	5	7	3	8	19	6	11	12	22			

Binary Heap

- Deletion



A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
13	2	4	3	7	5	7	15	8	19	6	11	12	22			

Beyond the Heap

- What else can be made implicit?

- Toy Problem: Dynamic Membership
  - Design a data structure that can:
    - Insert(x)
    - Delete(x)
    - Member(x)

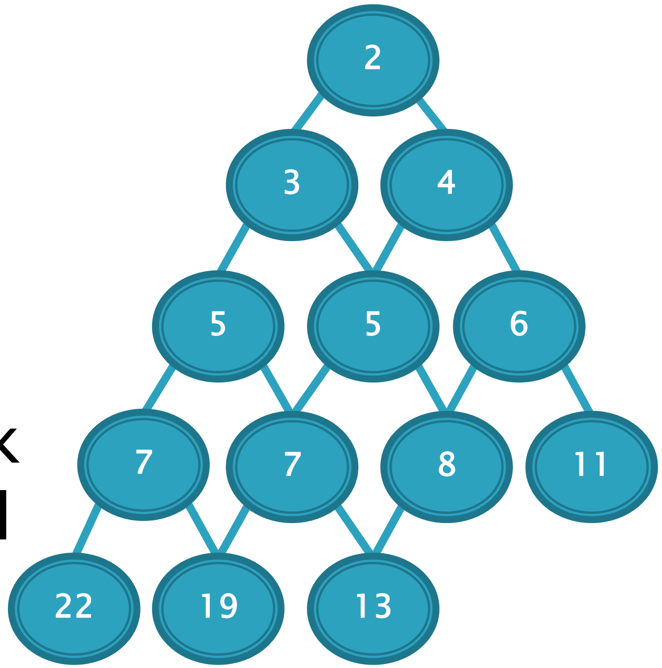
- Heap doesn't work well for member
  - Has very large antichains

Beyond the Heap

- Dynamic Membership
  - Insert(x)
  - Delete(x)
  - Member(x)
  - Heap:
    - Insert $\to \Theta(\log n)$, Delete $\to \Theta(n)$, Member $\to \Theta(n)$
  - Unsorted list:
    - Insert $\to \Theta(1)$, Delete $\to \Theta(n)$, Member $\to \Theta(n)$
  - Sorted list:
    - Insert $\to \Theta(n)$, Delete $\to \Theta(n)$, Member $\to \Theta(\log n)$
- What other trade-offs exist?

Beaps: Biparental Heaps

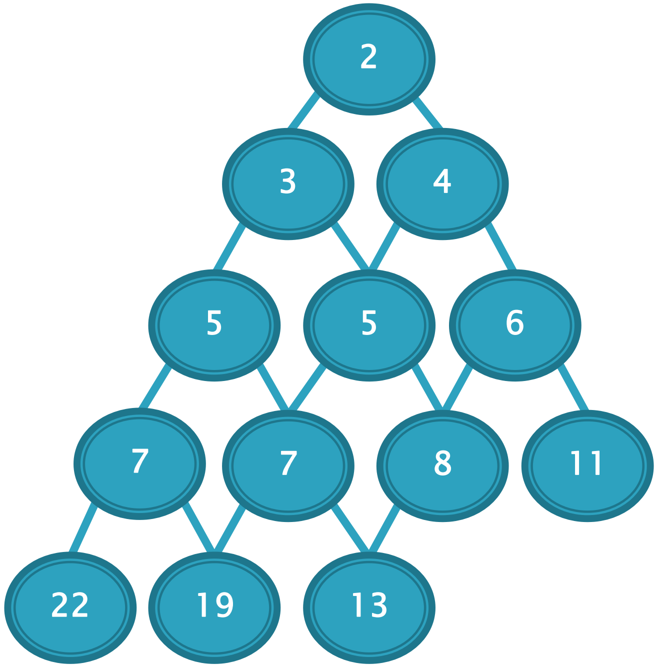
- Beap Properties:
  - Partitioned into $\sqrt{2n}$ blocks:
    - i–th block $[i(i+1)/2\ +1..i(i+1)/2]$
  - k–th element in the j–th block is no larger than the k–th and $(k+1)$–th in $(j+1)$–th block



A[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
13	2	3	4	5	5	6	7	7	8	11	22	19	13			

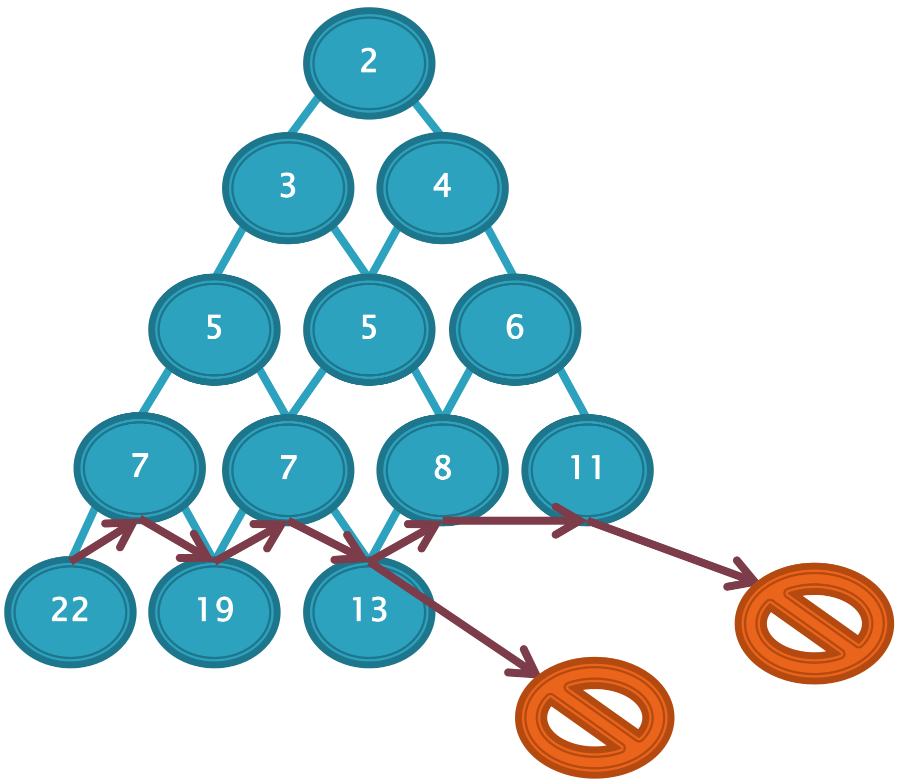
Beaps: Biparental Heaps

- Searching for 17



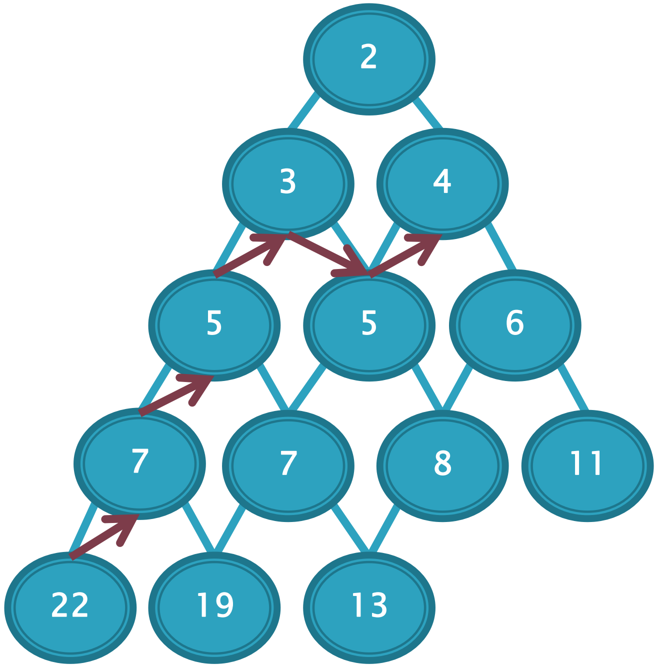
Beaps: Biparental Heaps

- Searching for 17



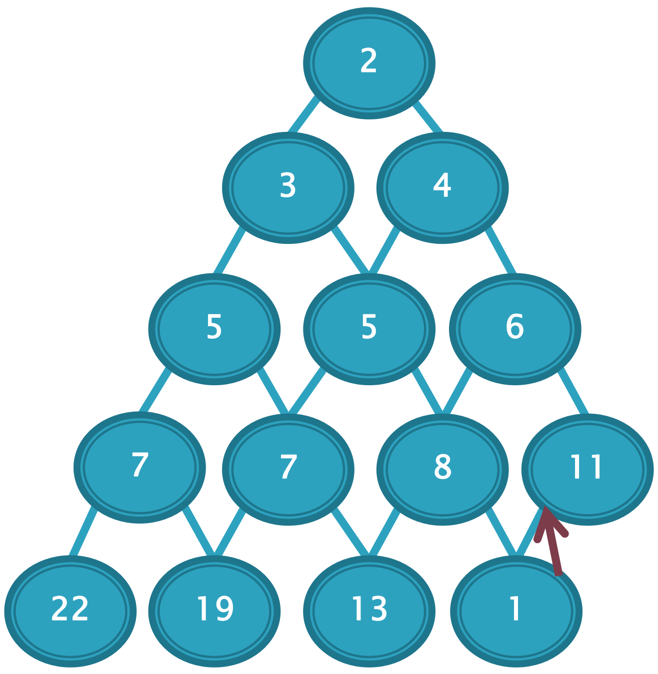
Beaps: Biparental Heaps

- Searching for 4



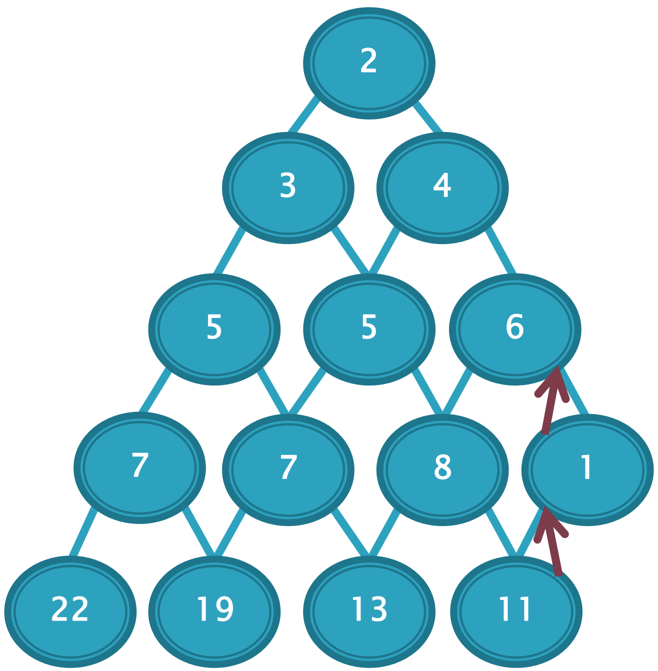
Beaps: Biparental Heaps

- Inserting 1



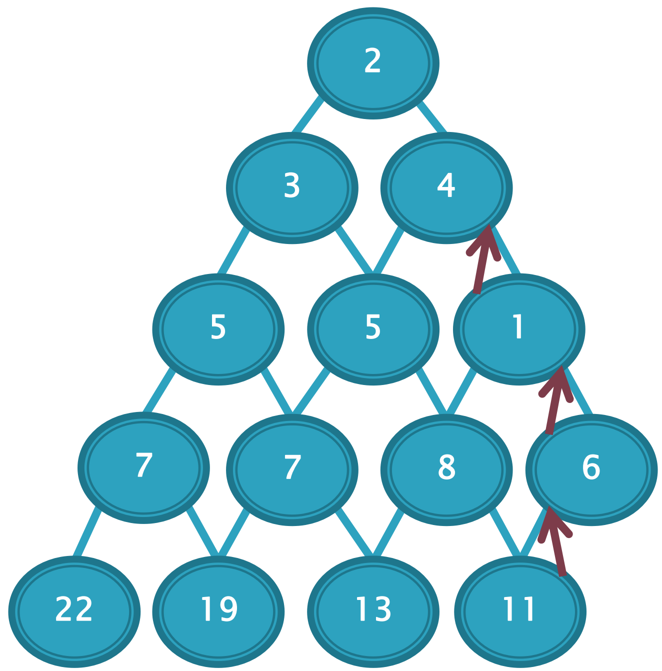
Beaps: Biparental Heaps

- Inserting 1



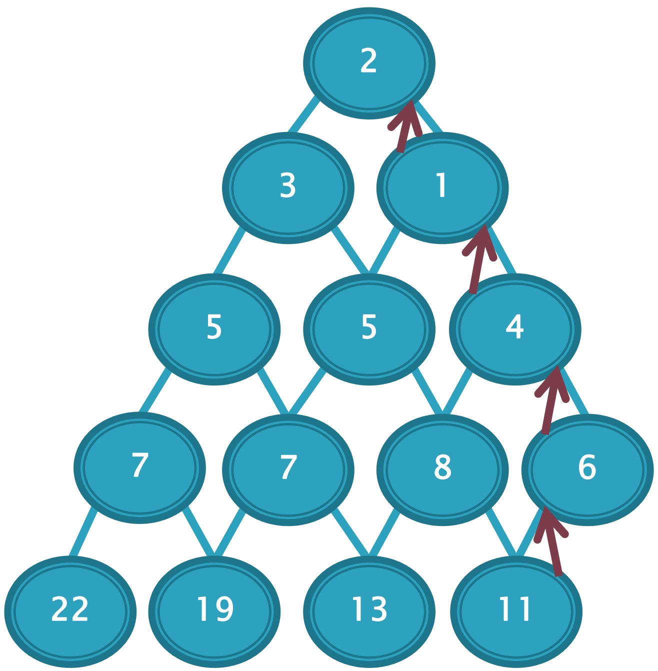
Beaps: Biparental Heaps

- Inserting 1



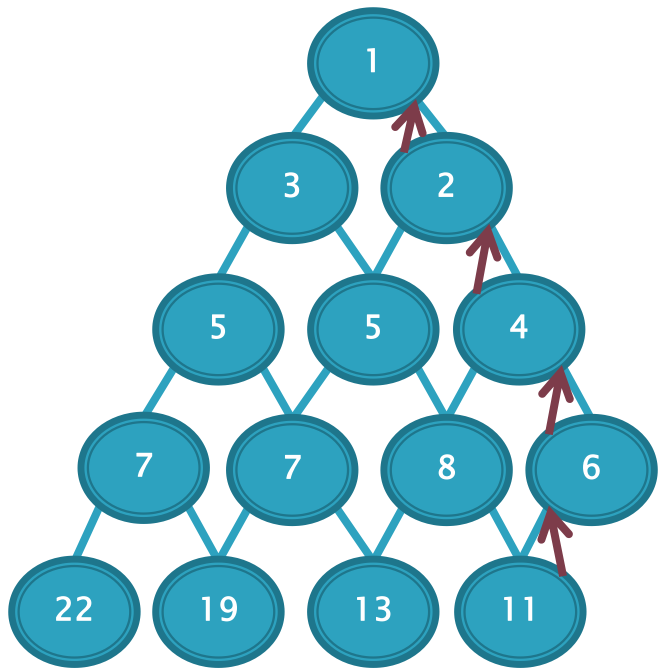
Beaps: Biparental Heaps

- Inserting 1



Beaps: Biparental Heaps

- Inserting 1



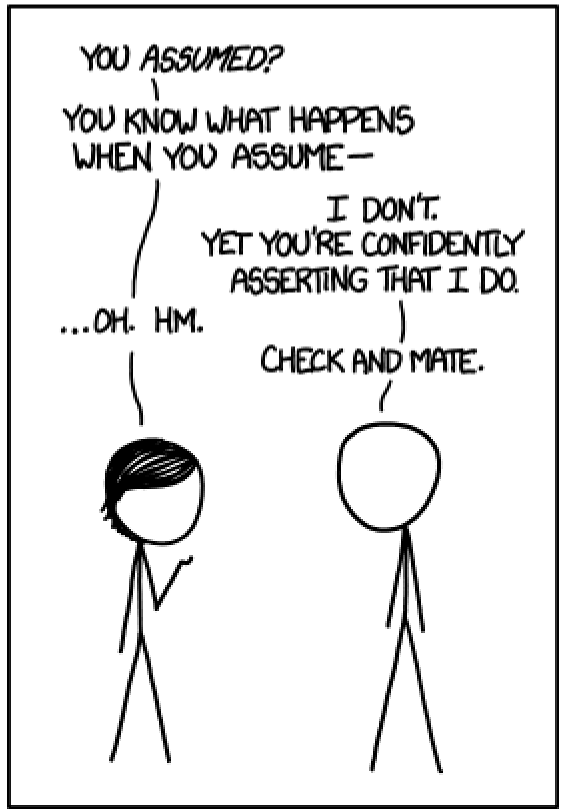
Beaps: Biparental Heaps

- Same idea as binary heap for deletion
- All three operations take $\Theta(\sqrt{n})$ time
- Elements stored in fixed partial order
  - Just as in the heap

Beaps: Biparental Heaps

- **Theorem (Munro and Suwanda 1980)**: If an implicit data structure containing n elements carries no structural information other than a fixed partial order on the stored values, then $U \cdot S \geq n$
  - $U \leftarrow$ worst case # of data moves during an update
  - $S \leftarrow$ worst case # of comparisons made during a search

But there is an assumption…



Rotated Lists

- What about non-partial orders?

- A rotated list: $\{7, 11, 13, 14, 1, 4, 5, 6\}$
  - Not hard to see that it is possible to modify binary search to find the minimum in the list
    - Caveat: (most) of the elements have to be distinct

- We can do better by using rotated lists
  - **But we must make the distinctness assumption!**

Basic Rotated List Scheme

- Data structure:
  - Keep $\sim\sqrt{2n}$ rotated lists, list i is of length i.
  - Invariant: Elements in list i are smaller than list $i + 1$
- Member:
  - Find two consecutive blocks that straddle query element
  - Search in the smaller block
  - Total cost: $\Theta(\log n)$
- Insertion:
  - Find block, insert
  - Swap max to min for each larger block
  - Total cost: $\Theta(\sqrt{n}\log n)$

Extensions to Rotated Lists

- **Munro and Suwanda (1980):**
  - Combine Beap and Rotated List to get
    - $\Theta(n^{1/3} \log n)$ for each operation

- **Fredrickson (1983):**
  - Applied recursion to Rotated Lists to get
    - $\Theta(\log n)$ time for Member(x)
    - $\Theta(n^{\sqrt{2/\log n}} \log^{3/2} n)$ time for Insert(x) and Delete(x)

Fredrickson's Rotated Lists

- Fredrickson considered blocking schemes:
  - Partition the array into r blocks $B(1), \dots, B(r)$
  - There is a function f s.t. $|B(i)| = f(i)$
  - The j-th block contains elements $1 + \sum_{i=1}^{j-1} f(i)$ to $\sum_{i=1}^{j} f(i)$
    - The Basic Rotated List Scheme has $f(i) = i$

- Data structure idea: *Bootstrapping*
  - Sometimes we can plug a data structure into itself
    - Let D_1 have $f(i) = i$ and each block be a rotated list
    - Let D_2 have $f(i) = i^2$ and each block be D_1*
      - This gives us $\Theta(\log n)$ search, and $\Theta(n^{\frac{1}{3}} \log n)$ updates!
    - Let D_3 have …

*Numerous missing details regarding the base case

Beyond Rotated Lists

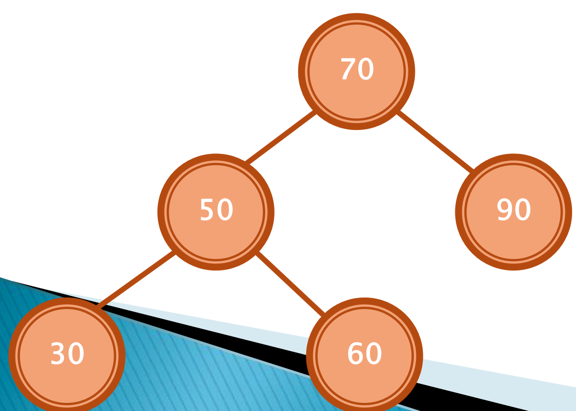
- **Theorem (Munro 1986)**: There is an implicit data structure for the membership problem that has worst case $\Theta(\log^2 n)$ time for Member(x), Insert(x), and Delete(x)

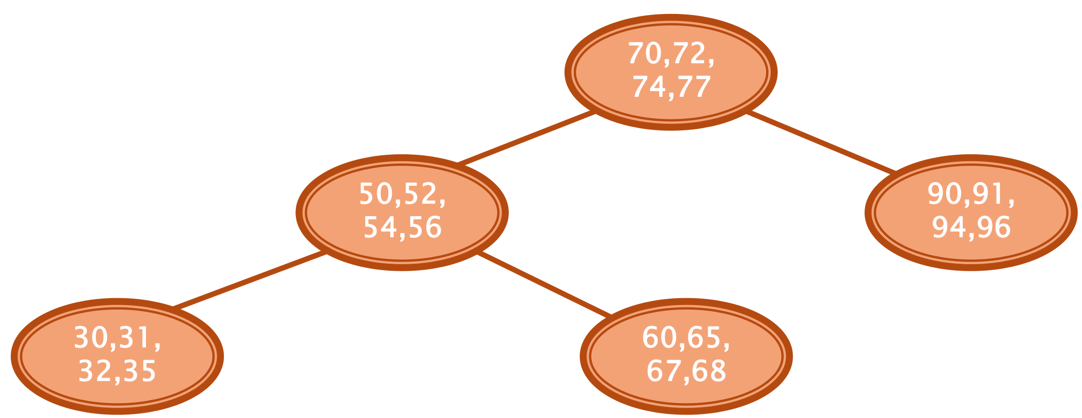
- What we really want is an balanced search tree
  - So, lets see if we can make such a tree implicit

Semi-Implicit AVL Tree

- **Theorem (Munro 1986):** There is a data structure for the membership problem that occupies $n + k^2$ array locations, and uses an additional $k + \Theta(n/k)$ pointers, counters, and flags. Member(x) takes $\Theta(\log n)$ time, and Insert(x) and Delete(x) take time $\Theta(k + \log n)$ time.

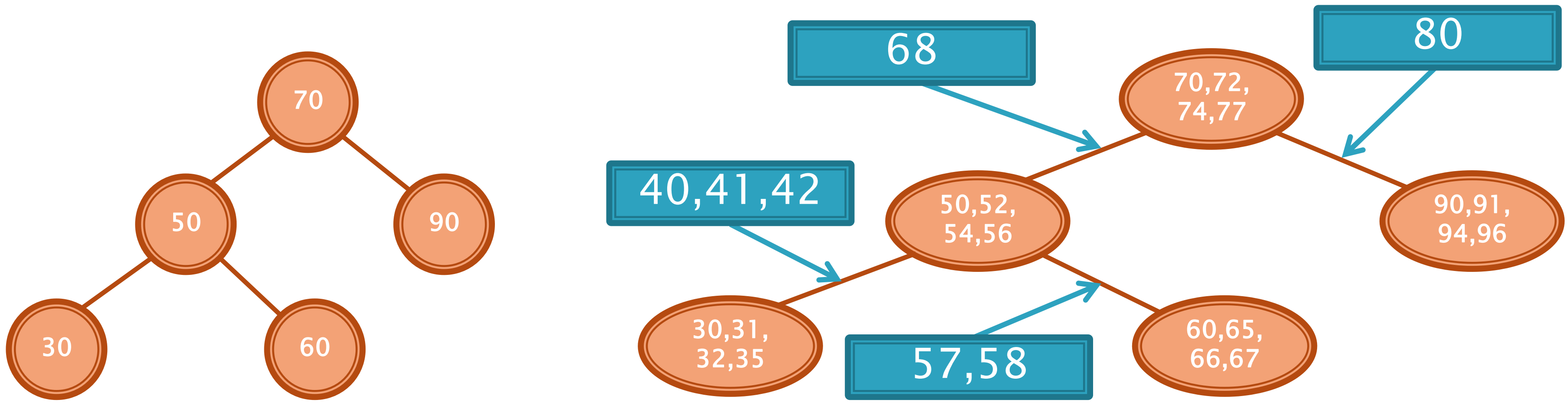
- **Invariant #1**: AVL node stores k consecutive elements
  - A *node* consists of k locations for elements
    - Also a constant number of pointers, flags, and counters
  - We take node sized blocks from the end of the data array





Semi-Implicit AVL Tree (2)

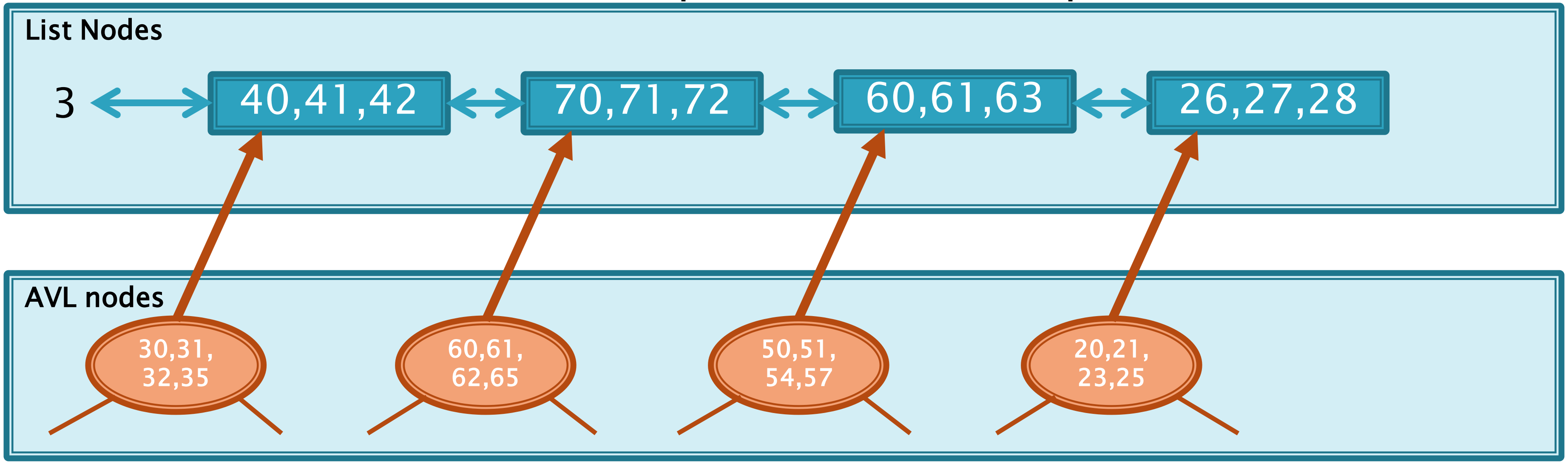
- We need some extra mechanism to update



- **Invariant**: 0 to k-1 consecutive elements between AVL nodes
- The elements between two nodes are called a *maniple*

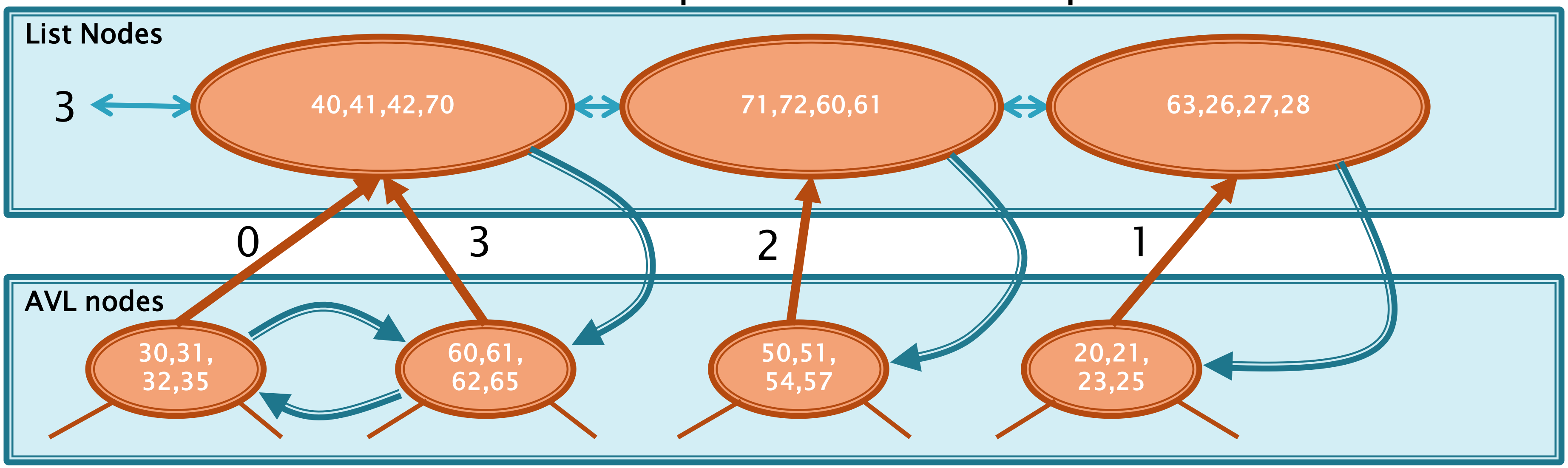
Managing Maniples (Logical)

- We keep pointers to $k-1$ doubly linked lists
  - Each linked list will also consist of nodes
  - List i will consist of all maniples of i elements
  - Each AVL node stores a pointer to its maniple



Managing Maniples (Physical)

- We keep pointers to $k-1$ doubly linked lists
  - Each linked list will also consist of nodes
  - List i will consist of all maniples of i elements
  - Each AVL node stores a pointer to its maniple



- **Each list node may contain maniples for up to k AVL nodes**
  - **This set of AVL nodes is called the *cohort* of the list node**
    - **We keep circular linked lists so we can find all AVL nodes in a cohort**

(Yes, there are a lot of pointers!)

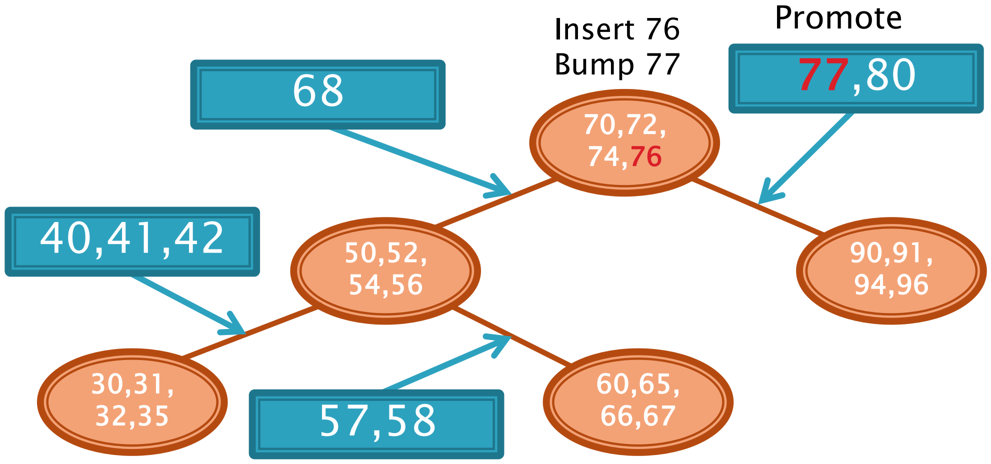
Managing Maniples (Updates)

- Memory Management:
  - When we need a new node, get it from the array
    - New list nodes inserted at the head of the list
  - To delete a maniple, swap contents with head
    - Must update maniple/cohort pointers in process
    - If head underflows, swap with final node in array
    - Overall this requires $\Theta(\log n + k)$ time

- Thus, we can assume the following primitives:
  - PromoteManiple(p, i, x): move maniple pointed to by p, of size i into maniple list $i + 1$, while inserting x into the correct position
  - DemoteManiple(p, i, x): move maniple pointed to by p, of size i into maniple list $i - 1$, and delete x

Performing Operations

- Insert is conceptually very easy:
  - Two cases: both more or less the same
    - Insert into an AVL node → bump max element into maniple
    - OR Insert directly into maniple
  - So, we what we really need is to handle maniple insertion:
    - If the maniple is empty, make a new one in list 1
    - If the maniple is already of size k-1, make AVL node
    - Otherwise, we use PromoteManiple
- Deletion is analogous ☺
- Search:
  - In the AVL tree: $\Theta(\log n)$
  - In a node: $\Theta(\log k)$
  - Total: $\Theta(\log n)$



Making it Implicit

- Recall that nodes store k consecutive values:
  - We can encode $k/2$ bits in these values!

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
2	3	4	5	7	12	13	17	18	20	22	29

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
3	2	4	5	12	7	13	17	20	18	29	22

1 0 1 0 1 1

  - Takes $\Theta(k)$ time to decode/encode a pointer!
    - We will set $k = \log n$ and get $\Theta(\log^2 n)$ time for all ops.

Making it Implicit (2)

- We set $k = c[\log n]$, where c is a big constant
  - e.g., $c = 10$ it will be large enough
- Dealing with the cruft:
  - There are k−1 linked lists of maniples
    - Each list can have up to k−1 unused locations
    - Thus, we are wasting $\Theta(k^2)$ locations in total!
    - We store these in the final locations of the array
      - Problem solved with extra pointers
- Are we done?

Making it Implicit (3)

- Annoying issue:
  - The value of $\lfloor \log n \rfloor$ will change eventually
  - Luckily, there is an easy solution:
    - Keep $\Theta(\log\log n)$ copies of the membership structure
      - Structure i stores 2^{2^i} elements
      - Perform search/updates on all the dictionaries
        - Similar to the rotate list idea for updates
    - We can maintain the running time of $\Theta(\log^2 n)$
- The end?

Wrap Up

- Several improvements since:
  - Franceschini et al. (2004):
    - All operations $\Theta(\log^2 n / \log\log n)$
  - Franceschini and Grossi (2003,2006):
    - All operations $\Theta(\log n)$
  - Brodal et al. (2012, 2013)
    - Other desirable properties

Next Problem: Multikey Search

- Unlike the last problem, this one will be **static**

- Input:
  - A set of n records, each record has k keys
- Goal:
  - Order records for efficient searching using *any* key

Two Key Case: Attempt #1

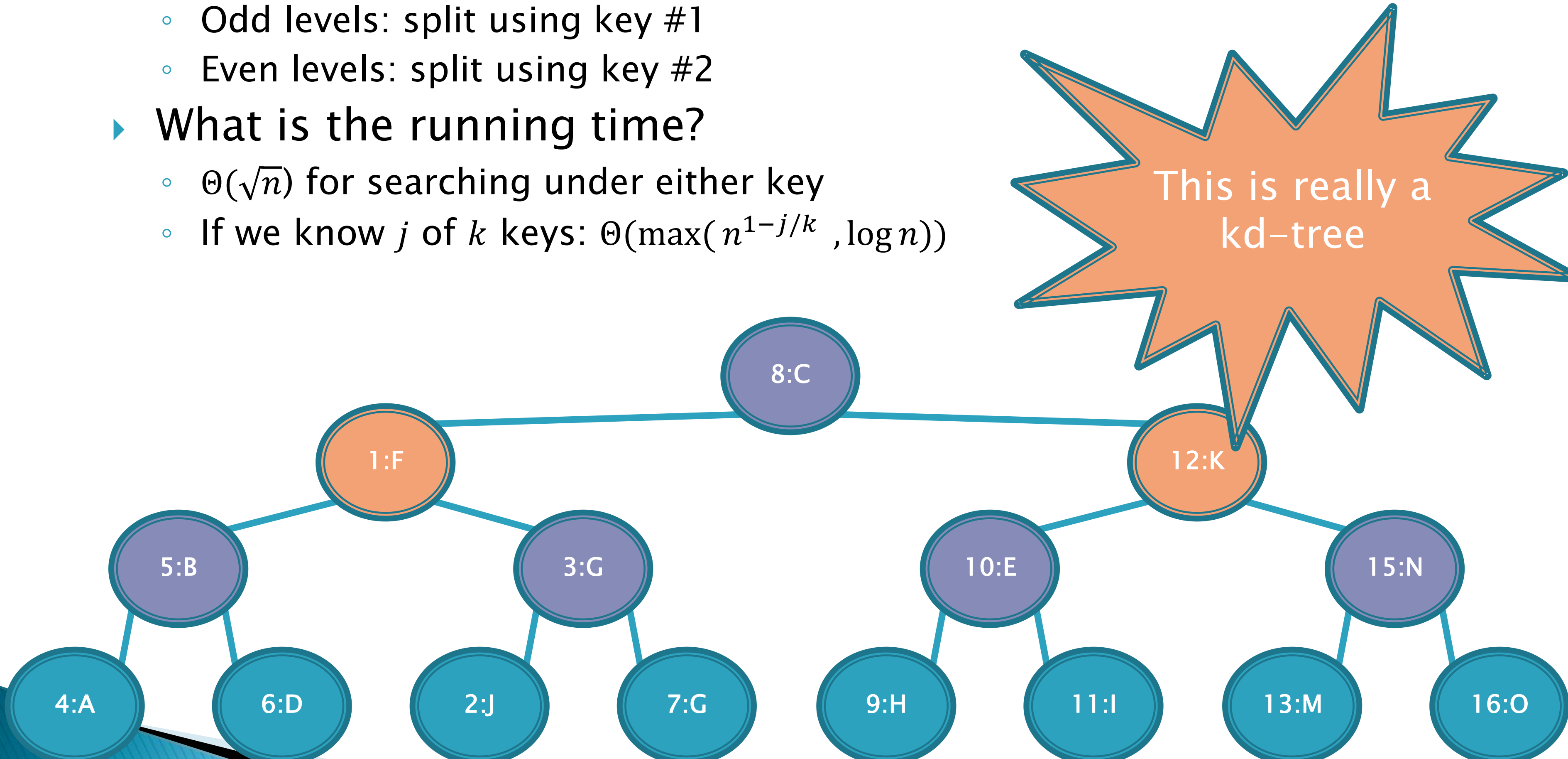
- Sort the records according to key #1
- Break it up into blocks of size $\sqrt{n}$
- Sort each block according to key #2

- Search using key #1 takes $\Theta(\sqrt{n})$ time
- Search using key #2 takes $\Theta(\sqrt{n}\log n)$ time

- Can we do better?

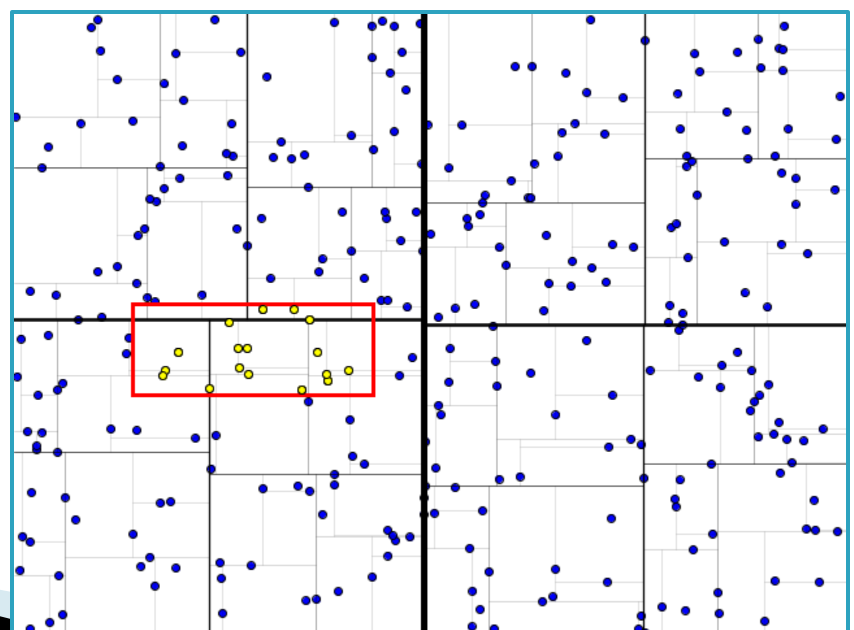
Two Key Case: Attempt #2

- We store the elements in a BST layout (like the heap)
  - Odd levels: split using key #1
  - Even levels: split using key #2
- What is the running time?
  - $\Theta(\sqrt{n})$ for searching under either key
  - If we know j of k keys: $\Theta(\max(n^{1-j/k}, \log n))$



Kd-trees

- We can also do *orthogonal range reporting:*
  - Time complexity: $\theta(\sqrt{n} + t)$ where t is output size
  - Proof: Consider the number of *cells* that are cut by a horizontal or vertical line…

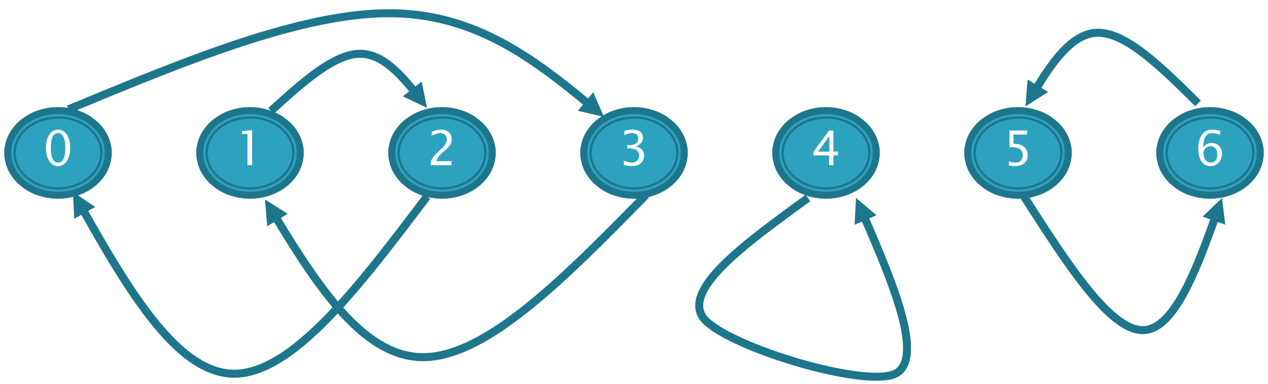


A Relevant Lower Bound

- **Theorem (Alt, Mehlhorn, Munro 1984):** Assume all comparisons are required to involve the element for which we are searching. If n elements can be arranged in an array such that any of p different permutations of the ascending order may occur, then searching requires $\Omega(p^{1/n})$ comparisons.

Permutations, Cycles, and Involutions

- Consider the following permutation:
  - $\pi = (3,2,0,1,4,6,5)$ as a directed graph:



- A permutation induces a set of *cycles*
  - The length of a cycle is the number of elements
- A permutation which is its own inverse is called an *involution*
  - In an involution, all cycles are of length ≤ 2
  - Example: $\pi = (1,0,3,2,5,4,7,6)$ or the bit encoding trick

Feldman's Involution Idea

- Consider the following ordering scheme:
  - Take the first $n/4$ odd elements and pair them arbitrarily with the last $n/4$ odd elements
    - This admits $(n/4)!$ permutations

  - Lower bound says search time should be $\Omega(n^{1/4})$…
    - But we can still search in $\Theta(\log n)$ time if we make comparisons that **don't** involve the query element!

Two Key Case: Attempt #3

- We will use the involution trick to show:
- **Theorem (Munro 1987):** The static two-key search problem is solvable in $\Theta(\log^2 n \log\log n)$ time for searching under either key.

Two Key Case: Attempt #3 (2)

- Feldman's scheme:
  - Elements in position $0 \; mod \; 2$ in sorted order
  - Elements in position $1 \; mod \; 2$ permuted
- Munro's 2-key scheme:
  - Start by sorting by key 1
  - Records in $0 \; mod \log n$ sorted by key 1
    - Call these 1-guides
  - Conceptually $\log n \; - 1$ data structures
    - D_i for records in position $i \; mod \log n$
  - **Invariant**: $x \in \; D_i$ straddled by 1-guides

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	3	4	5	7	9	12	14	17	19	22	25	29	31	33	37
9	7	12	22	46	17	11	13	33	34	37	10	2	3	8	1

Two Key Case: Attempt #3 (3)

- For each D_i:
  - Put first half of the records into second half of array sorted by key 2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	3	4	5	7	9	12	14	17	19	22	25	29	31	33	37
9	7	12	22	46	17	11	13	33	34	37	10	2	3	8	1

Two Key Case: Attempt #3 (3)

- For each D_i:
  - Put first half of the records into second half of array sorted by key 2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	3	4	5	7	9	12	14	17	19	22	25	29	31	33	37
9	7	12	22	46	17	11	13	33	34	37	10	2	3	8	1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	19	4	5	7	31	12	14	17	3	22	25	29	9	33	37
9	34	12	22	46	3	11	13	33	7	37	10	2	17	8	1

Two Key Case: Attempt #3 (3)

- For each D_i:
  - Put first half of the records into second half of array sorted by key 2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	3	4	5	7	9	12	14	17	19	22	25	29	31	33	37
9	7	12	22	46	17	11	13	33	34	37	10	2	3	8	1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	19	33	5	7	31	22	14	17	3	12	25	29	9	4	37
9	34	8	22	46	3	37	13	33	7	11	10	2	17	12	1

Two Key Case: Attempt #3 (3)

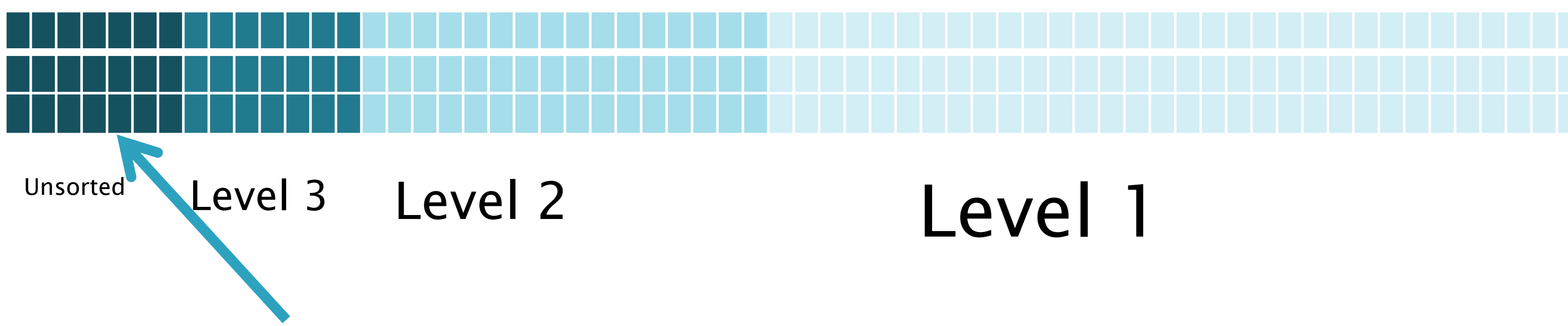
- For each D_i:
  - Put first half of the records into second half of array sorted by key 2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	3	4	5	7	9	12	14	17	19	22	25	29	31	33	37
9	7	12	22	46	17	11	13	33	34	37	10	2	3	8	1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	19	33	37	7	31	22	25	17	3	12	14	29	9	4	5
9	34	8	1	46	3	37	10	33	7	11	13	2	17	12	22

Two Key Case: Attempt #3 (4)

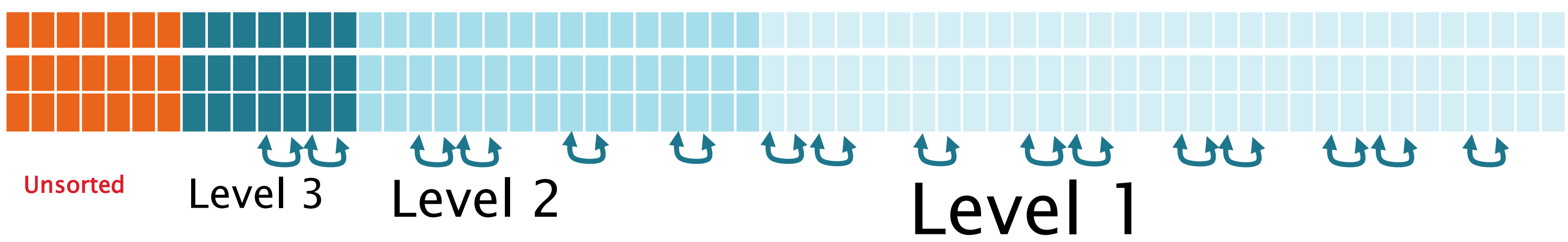
- Keep doing this recursively for each D_i:
  - Put the first half into the second sorted by key 2
  - Put the first quarter into the second sorted by key 2
  - Put the first eighth into the second sorted by key 2
  - …
  - Stop after $\log\log n + c$ recursive calls for some $c > 0$
  - Call the j-th sorted chunk from the right *level* j



$\Theta\left(\frac{n}{\log^2 n}\right)$ records

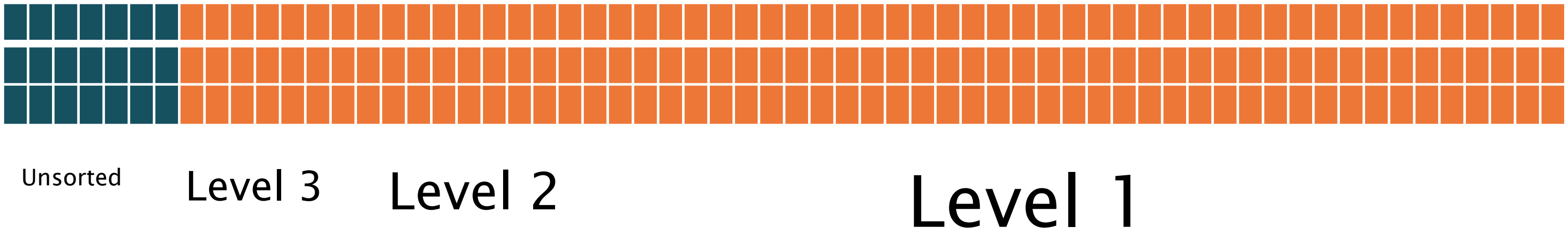
Supporting Search (Part 1)

- We now show how to:
  - Search among the 1-guides using key 2
  - Search among the unsorted portions of D_i (either key)
- Idea that we have seen before:
  - Encode pointers in the pairs of records sorted by key 2
  - We have $\Theta(n)$ such records $\rightarrow$ can encode $\Theta(n/\log n)$ pointers
  - We can use these pointers to encode search trees



Supporting Search (Part 2)

- Next: how to search using key 2 on the remaining records
  - We have $\Theta(\log n)$ data structures
  - Each structure has $\Theta(\log \log n)$ levels
  - Each level is sorted using key 2
  - Overall time: $\Theta(\log^2 n \log \log n)$



Searching (Part 3)

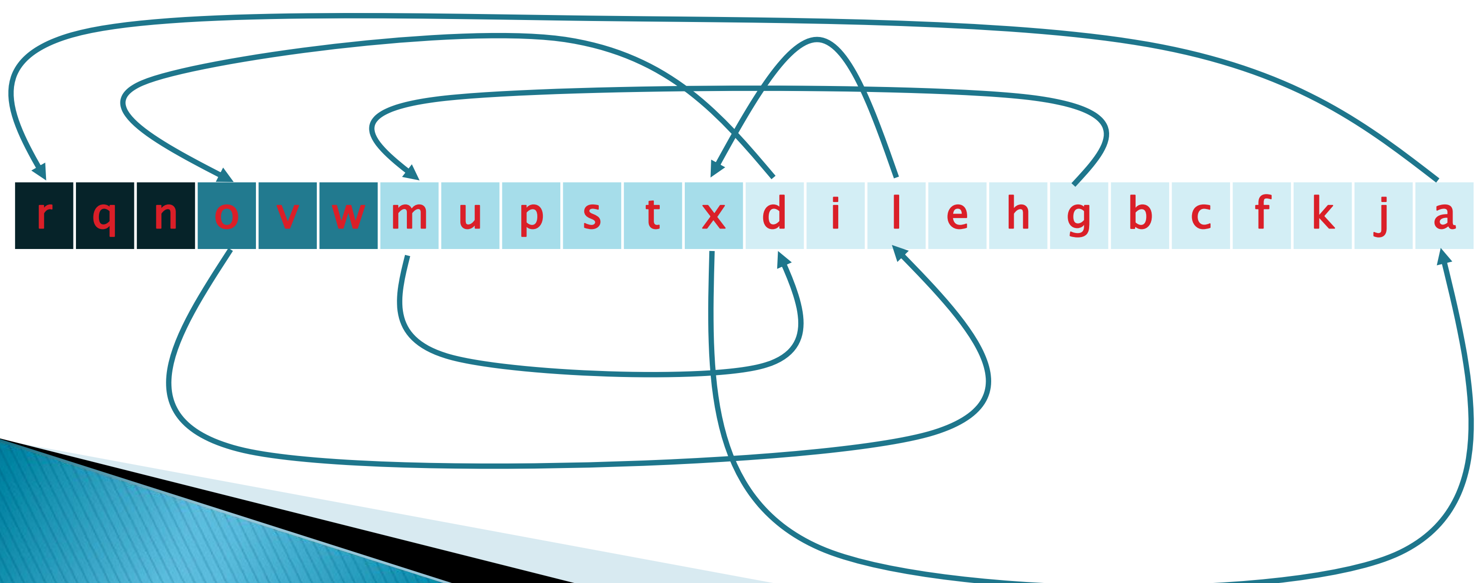
- Finally: searching using key 1
  - The "much more interesting case"
  - **Remember (Invariant)**: each $y \in D_i$ is straddled by 1-guides
    - Thus, we can determine where the query element x **should** be
      - That is, we can find a range r of $\log n$ positions (1 per D_i)
  - We need to do a binary search within r
    - $\Theta(\log \log n)$ to search r
    - For each D_i we have to track down the correct record
      - How long does this take?

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	?	?	?	7	?	?	?	17	?	?	?	29	?	?	?

Tracking down elements

- Consider a single D_i

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
x	s	t	m	p	u	r	q	n	w	v	o	d	i	l	e	h	g	b	c	f	k	j	a
o	w	v	r	n	q	m	u	p	s	t	x	d	i	l	e	h	g	b	c	f	k	j	a
r	q	n	o	v	w	m	u	p	s	t	x	d	i	l	e	h	g	b	c	f	k	j	a



Supporting Search (Part 3)

- Finally: searching using key 1
  - The "much more interesting case"
  - **Remember (Invariant)**: each $y \in D_i$ is straddled by 1–guides
    - Thus, we can determine where the query element x **should** be
      - That is, we can find a range r of $\log n$ positions (1 per D_i)
  - We need to do a binary search within r
    - $\Theta(\log \log n)$ to search r
    - For each D_i we have to track down the correct record
      - Tracking down: $\Theta(\log n)$ moves, each move: $\Theta(\log n)$ cost
- Overall time: $\Theta(\log^2 n \log \log n)$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	?	?	?	7	?	?	?	17	?	?	?	29	?	?	?

Wrap Up

- These results all generalize to 3 or more keys
- Fiat et al. (1988) essentially settled it:
  - With k keys we can search in $\Theta(k \log k \log n)$ time
  - This solution is somewhat complicated
    - Basic Idea: select guides using Hall's Theorem