Priority Queues: Introduction

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Data Structures
Data Structures and Algorithms

Outline

1. Overview
2. Naive Implementations

Learning objectives

You will be able to:

- Implement a priority queue
- Explain what is going on inside built-in implementations:
  - C++: `priority_queue`
  - Java: `PriorityQueue`
  - Python: `heapq`

Queue

A queue is an abstract data type supporting the following main operations:

- `PushBack(`e`)` adds an element to the back of the queue;
- `PopFront()` extracts an element from the front of the queue.

Priority Queue (Informally)

A priority queue is a generalization of a queue where each element is assigned a priority and elements come out in order by priority.

Priority Queues: Typical Use Case

Scheduling jobs

- Want to process jobs one by one in order of decreasing priority. While the current job is processed, new jobs may arrive.
- To add a job to the set of scheduled jobs, call `Insert`(*job*).
- To process a job with the highest priority, get it by calling `ExtractMax`().

Priority Queue (Formally)

Definition

Priority queue is an abstract data type supporting the following main operations:

- `Insert`(p) adds a new element with priority p
- `ExtractMax`() extracts an element with maximum priority

Example

Contents:

Queries:

`Insert(5)`

Example

Contents:

5

Queries:

Example

Contents:

5

Queries:

`Insert(7)`

Example

Contents:

5

7

Queries:

Example

Contents:

5

7

Queries:

`Insert(1)`

Example

Contents:

5

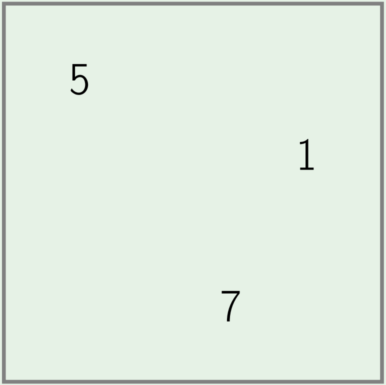
1

7

Queries:

Example

Contents:



Queries:

`Insert`(4)

Example

Contents:

5

1

4

7

Queries:

Example

Contents:

5

1

4

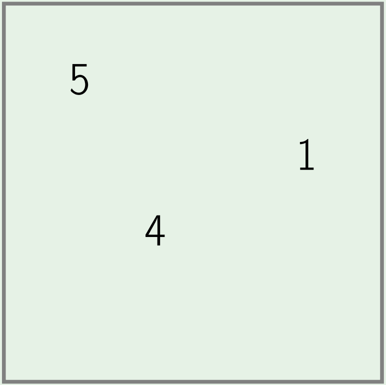
7

Queries:

$\texttt{ExtractMax}() \rightarrow 7$

Example

Contents:



Queries:

Example

Contents:

5

1

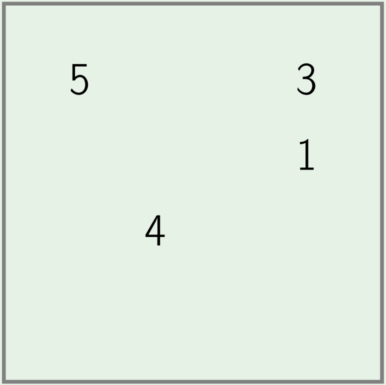
4

Queries:

`Insert`(3)

Example

Contents:



Queries:

Example

Contents:

5 3 1 4

Queries:

$\texttt{ExtractMax}() \to 5$

Example

Contents:

3
1
4

Queries:

Example

Contents:

3

1

4

Queries:

$\texttt{ExtractMax}() \rightarrow 4$

Example

Contents:

3

1

Queries:

Additional Operations

- `Remove`(it) removes an element pointed by an iterator it
- `GetMax`() returns an element with maximum priority (without changing the set of elements)
- `ChangePriority`(it, p) changes the priority of an element pointed by it to p

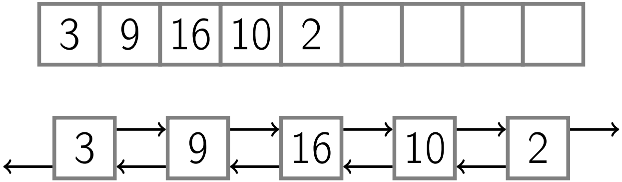
Algorithms that Use Priority Queues

- Dijsktra's algorithm: finding a shortest path in a graph
- Prim's algorithm: constructing a minimum spanning tree of a graph
- Huffman's algorithm: constructing an optimum prefix-free encoding of a string
- Heap sort: sorting a given sequence

Outline

1. Overview
2. Naive Implementations

Unsorted Array/List



- `Insert`(e)
    - add e to the end
    - running time: $O(1)$
- `ExtractMax`$()$
    - scan the array/list
    - running time: $O(n)$

Sorted Array

2	3	9	10	16				

- `ExtractMax()`
  - extract the last element
  - running time: $O(1)$
- `Insert`(e)
  - find a position for e ($O(\log n)$ by using binary search), shift all elements to the right of it by 1 ($O(n)$), insert e ($O(1)$)
  - running time: $O(n)$

Sorted List

- `ExtractMax()`
  - extract the last element
  - running time: $O(1)$
- `Insert`(e)
  - find a position for e ($O(n)$; note: cannot use binary search), insert e ($O(1)$)
  - running time: $O(n)$

Summary

	`Insert`	`ExtractMax`
Unsorted array/list	$O(1)$	$O(n)$
Sorted array/list	$O(n)$	$O(1)$
Binary heap	$O(\log n)$	$O(\log n)$

Priority Queues: Binary Heaps

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Data Structures
Data Structures and Algorithms

Outline

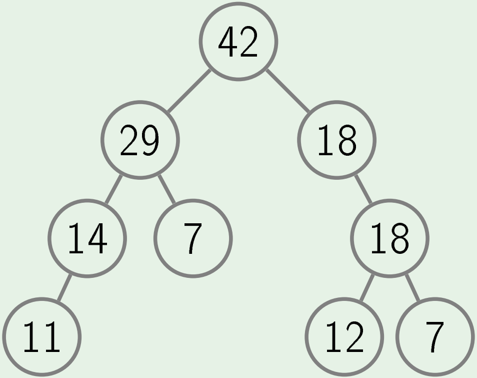
Definition

Binary max-heap is a binary tree (each node has zero, one, or two children) where the value of each node is at least the values of its children.

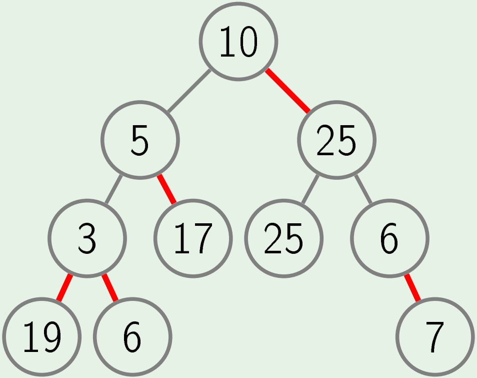
In other words

For each edge of the tree, the value of the parent is at least the value of the child.

Example: heap



Example: not a heap

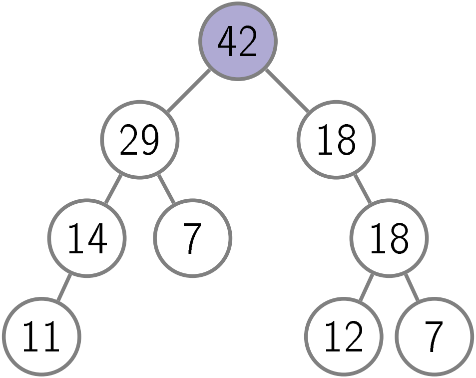


Outline

1. Binary Trees
2. Basic Operations
3. Complete Binary Trees
4. Pseudocode
5. Heap Sort
6. Final Remarks

GetMax

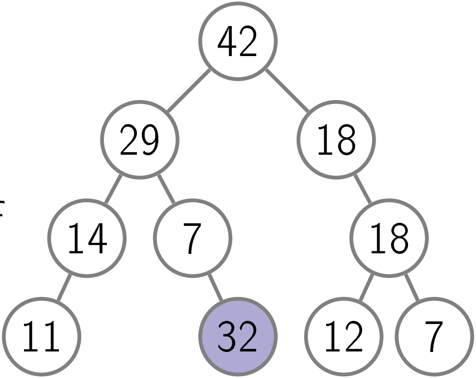
return the root value



running time: $O(1)$

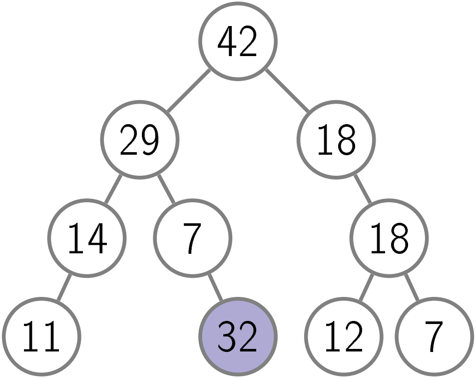
Insert

attach a new
node to any leaf



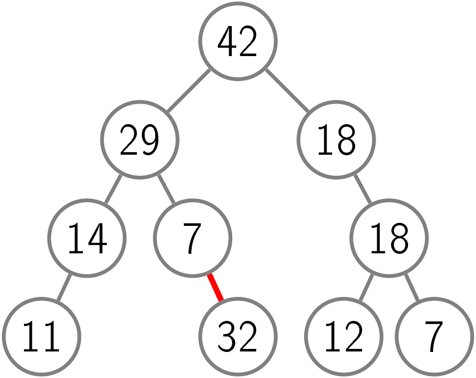
Insert

this may violate the heap property



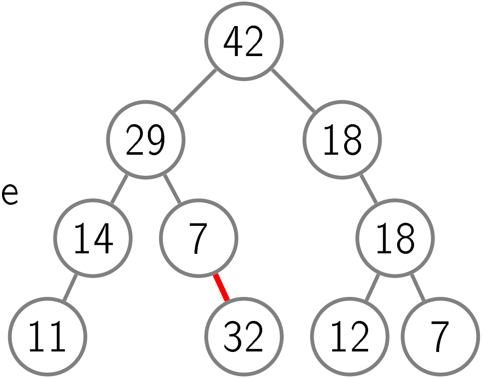
Insert

this may violate the heap property



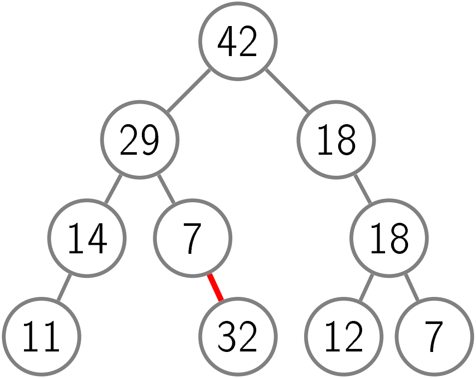
Insert

to fix this, we let the new node sift up

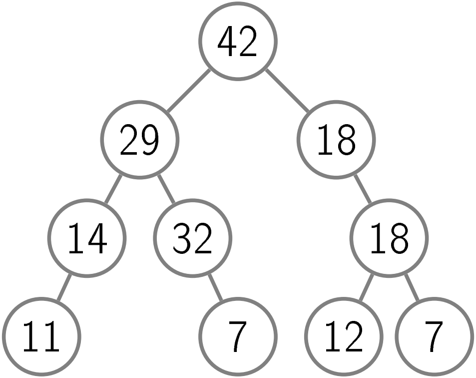


SiftUp

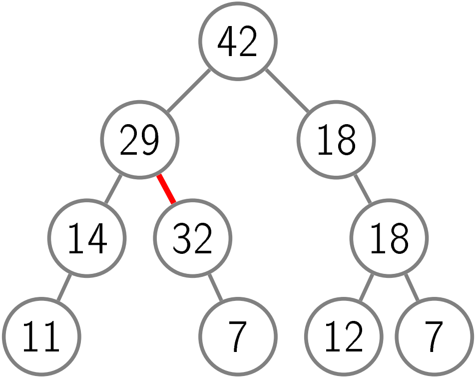
for this, we swap the problematic node with its parent until the property is satisfied



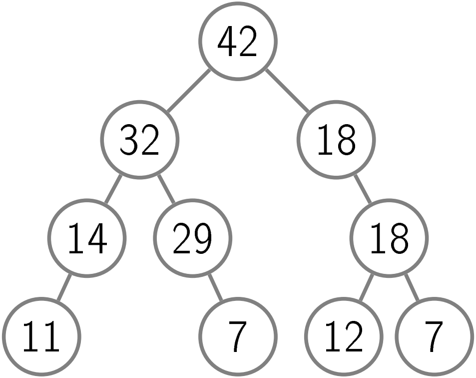
SiftUp



SiftUp

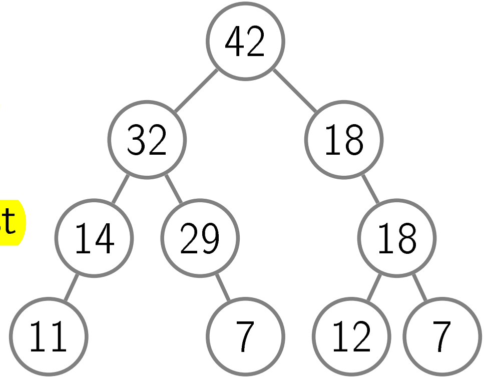


SiftUp



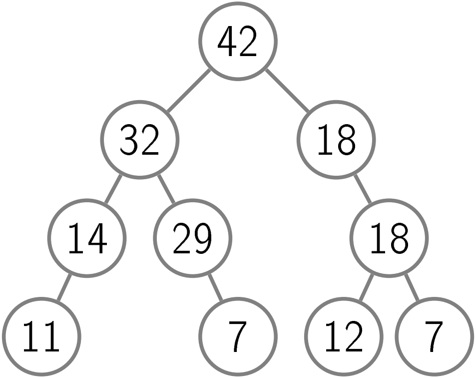
SiftUp

invariant: heap property is violated on at most one edge

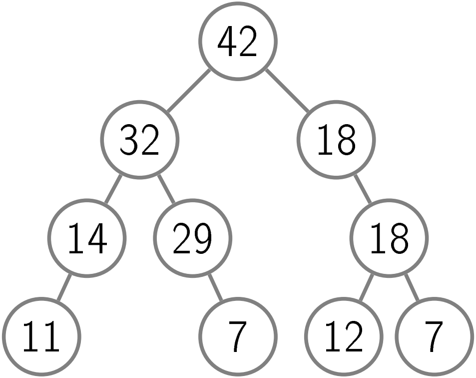


SiftUp

this edge gets closer to the root while sifting up



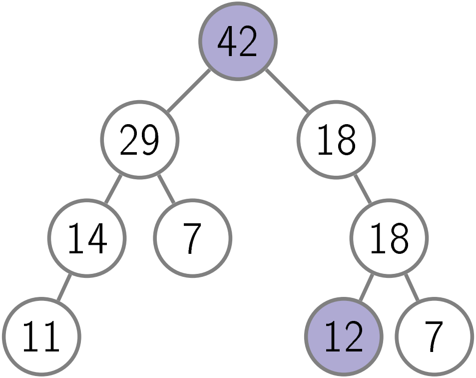
SiftUp



running time: O(tree height)

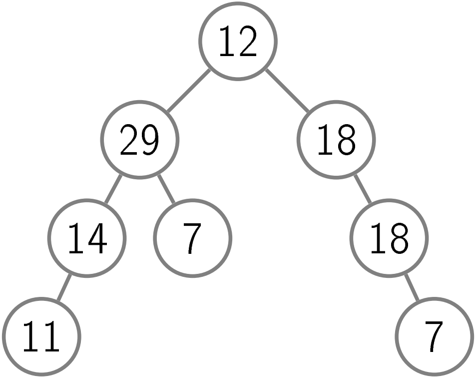
ExtractMax

replace the root
with any leaf



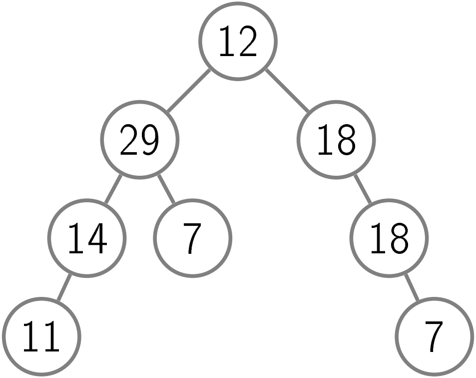
ExtractMax

replace the root
with any leaf



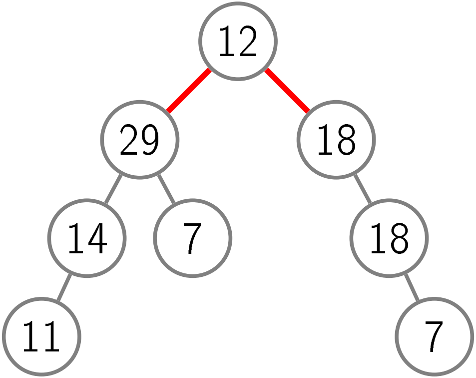
ExtractMax

again, this may violate the heap property



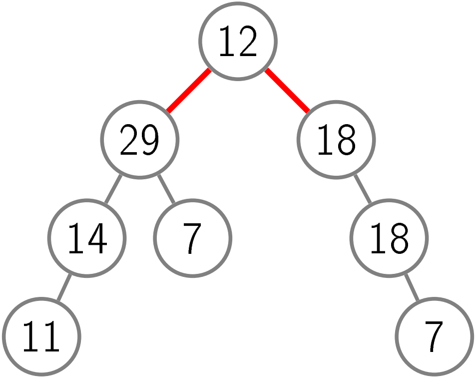
ExtractMax

again, this may violate the heap property



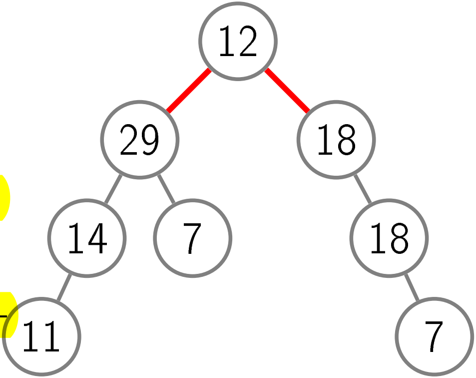
ExtractMax

to fix it, we let the problematic node sift down

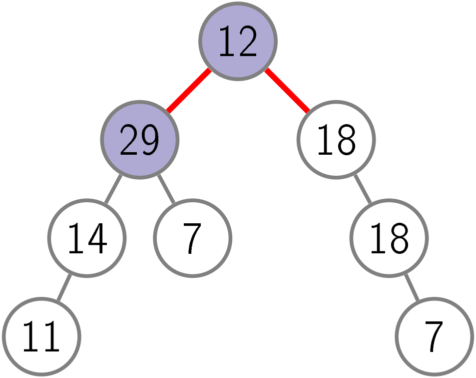


SiftDown

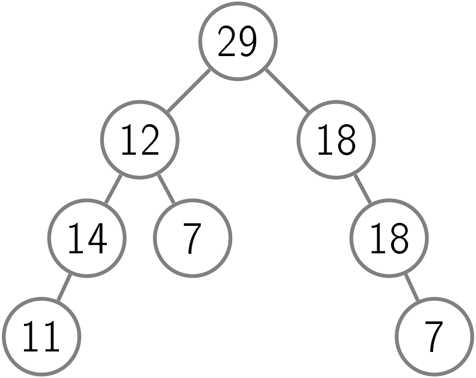
for this, we swap the problematic node with larger child until the heap property is satisfied



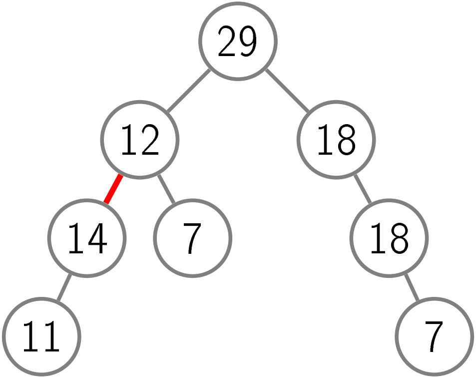
SiftDown



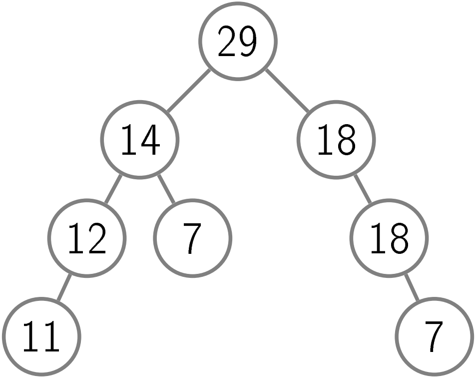
SiftDown



SiftDown

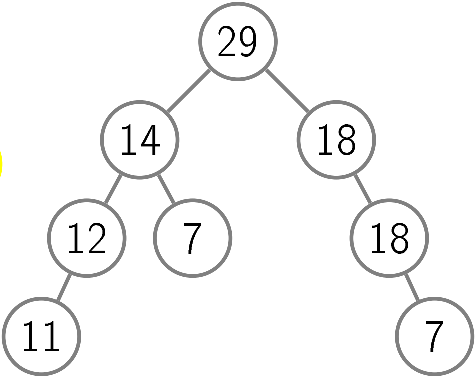


SiftDown

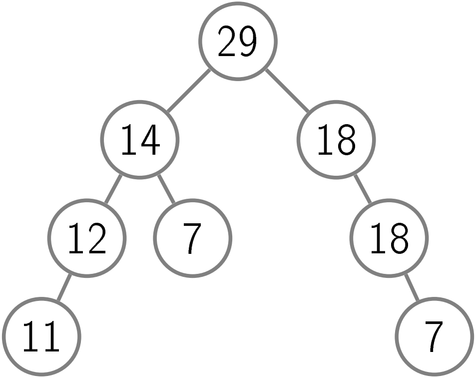


SiftDown

we swap with the larger child which automatically fixes one of the two bad edges



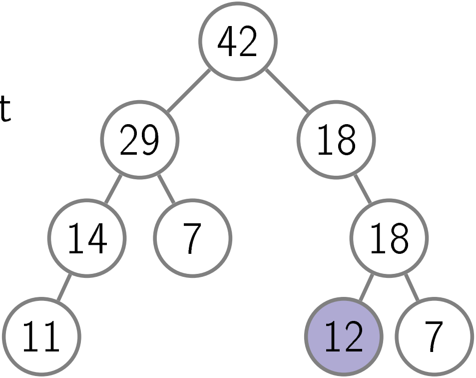
SiftDown



running time: O(tree height)

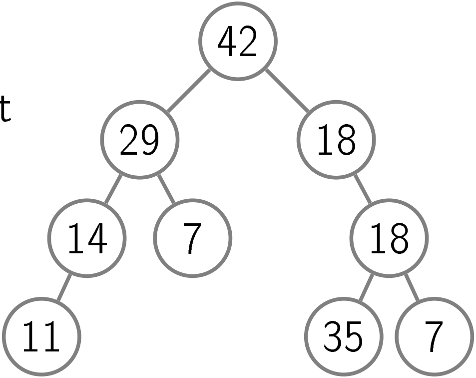
ChangePriority

change the priority and let the changed element sift up or down depending on whether its priority decreased or increased

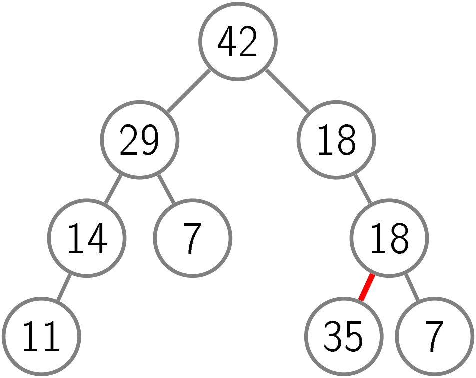


ChangePriority

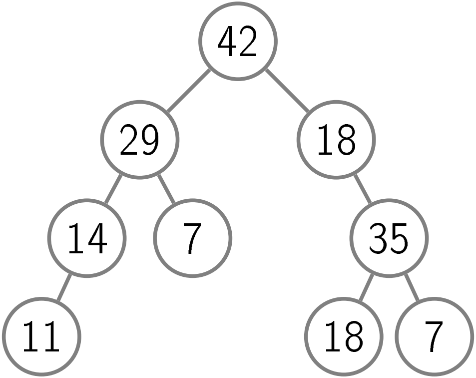
change the priority and let the changed element sift up or down depending on whether its priority decreased or increased



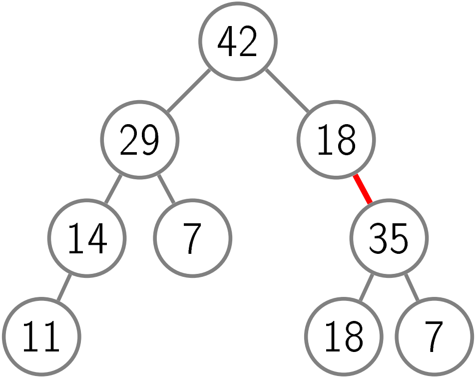
ChangePriority



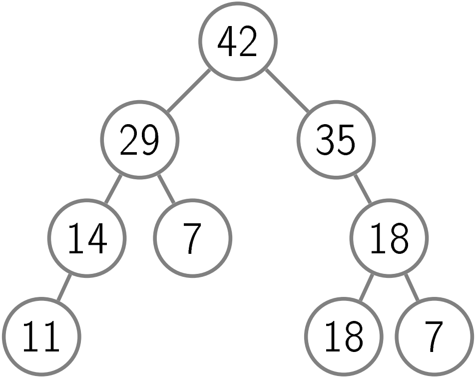
ChangePriority



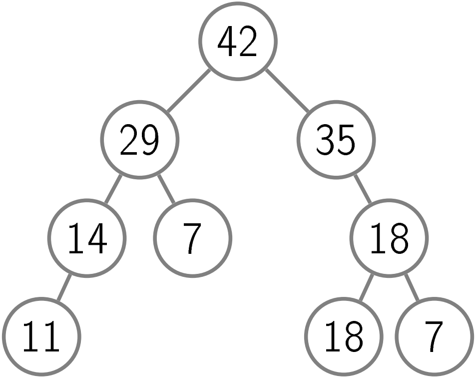
ChangePriority



ChangePriority



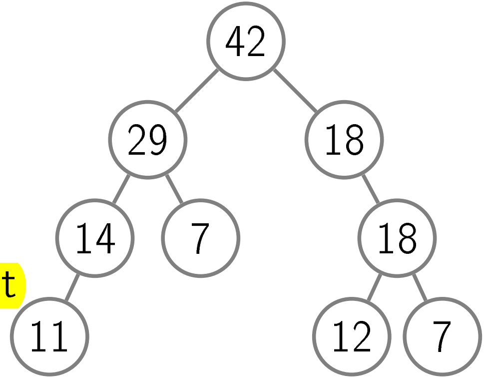
ChangePriority



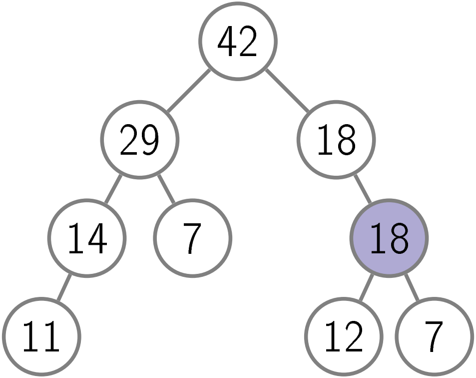
running time: O(tree height)

Remove

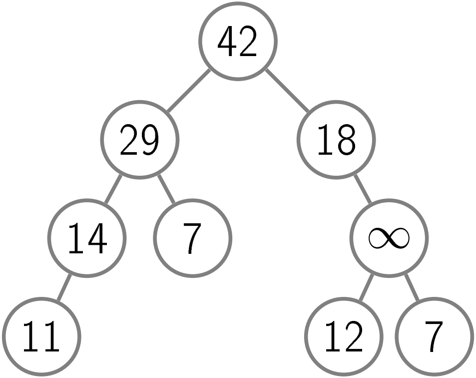
change the priority of the element to ∞, let it sift up, and then extract maximum



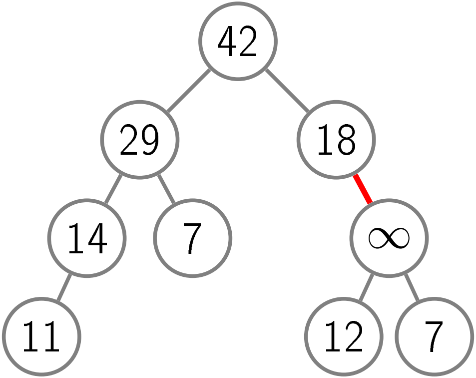
Remove



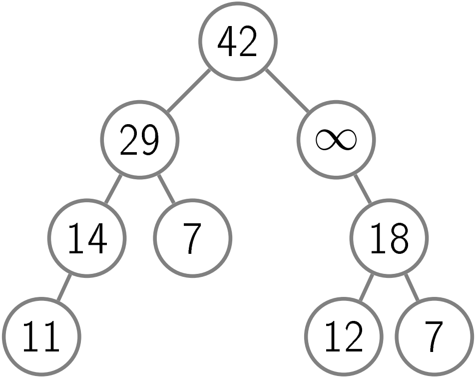
Remove



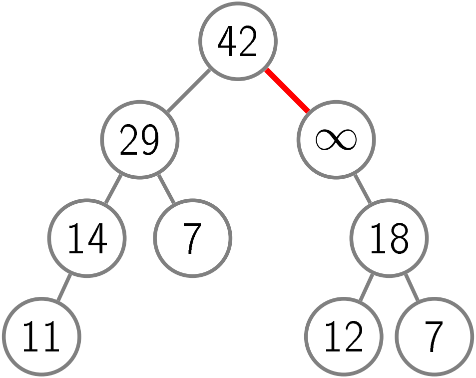
Remove



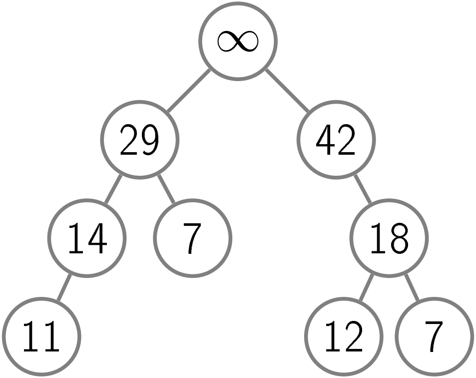
Remove



Remove



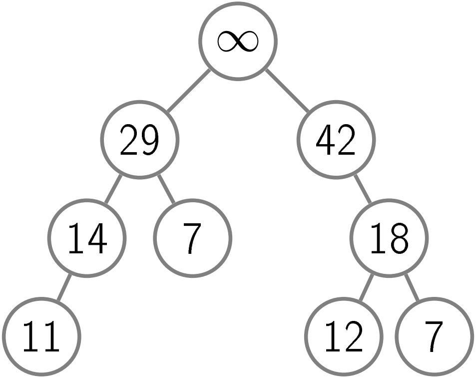
Remove



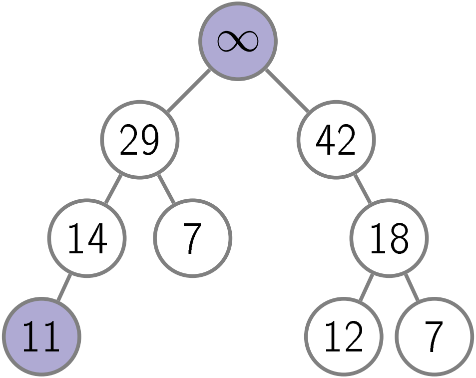
Remove

now, call

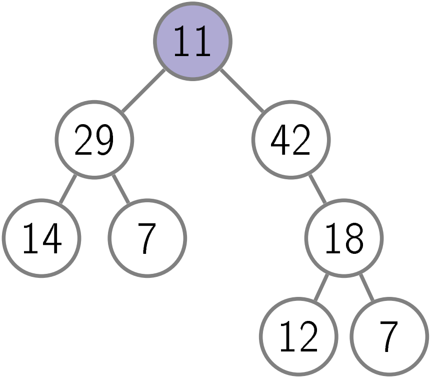
`ExtractMax()`



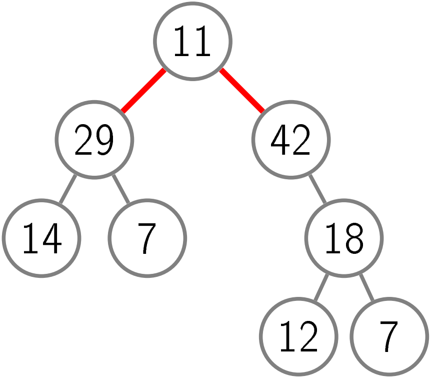
Remove



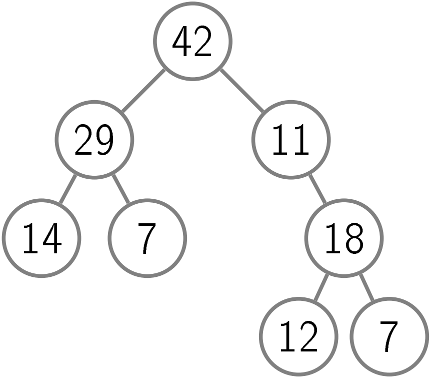
Remove



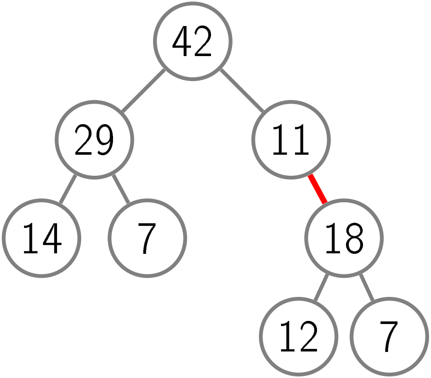
Remove



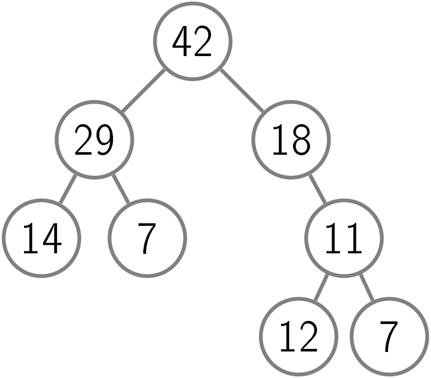
Remove



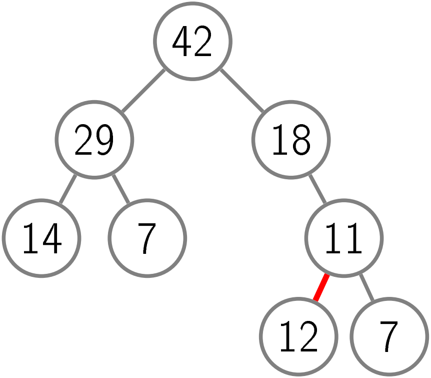
Remove



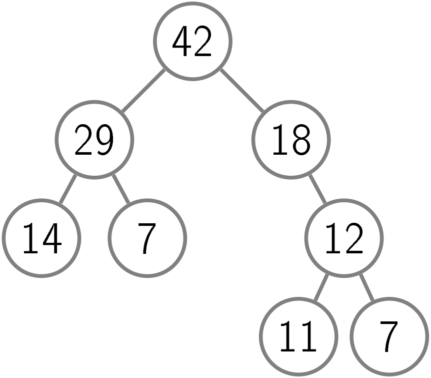
Remove



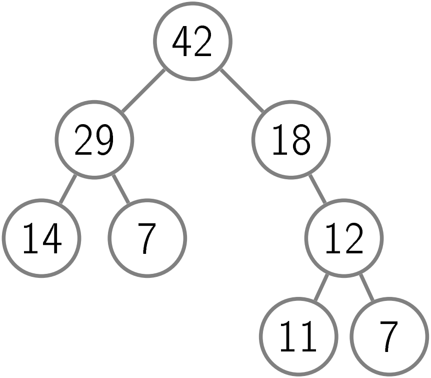
Remove



Remove



Remove



running time: $O(\text{tree height})$

Summary

- `GetMax` works in time $O(1)$, all other operations work in time $O(\text{tree height})$
- we definitely want a tree to be shallow

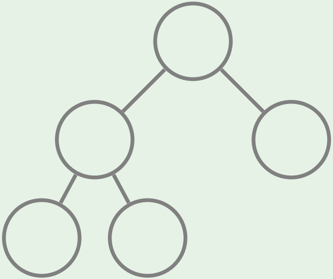
Outline

How to Keep a Tree Shallow?

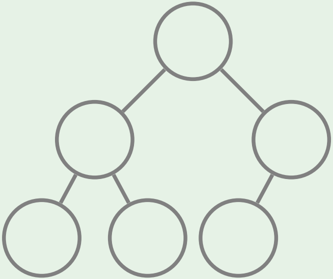
Definition

A binary tree is complete if all its levels are filled except possibly the last one which is filled from left to right.

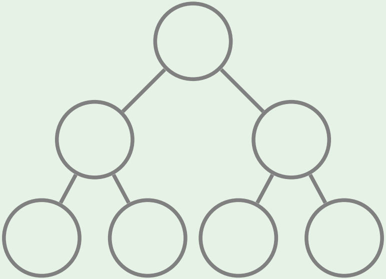
Example: complete binary tree



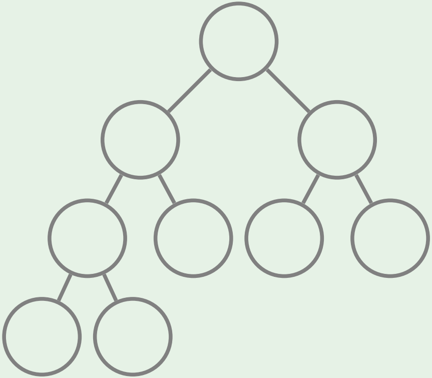
Example: complete binary tree



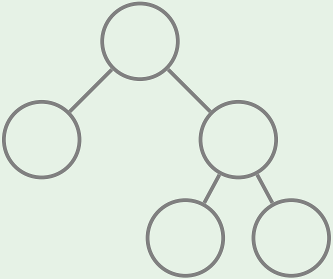
Example: complete binary tree



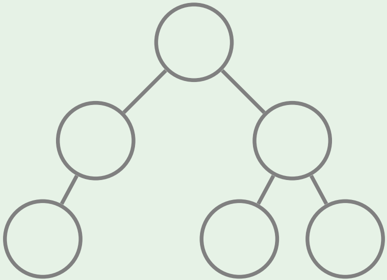
Example: complete binary tree



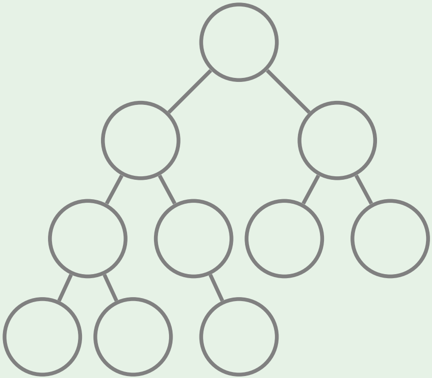
Example: not complete binary tree



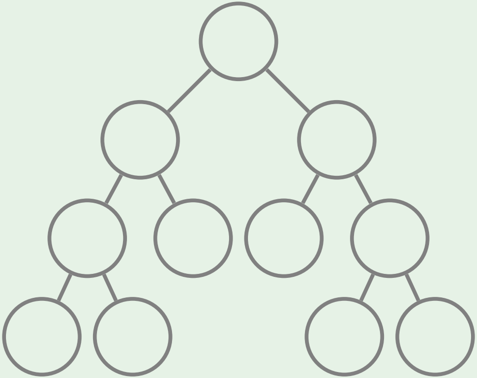
Example: not complete binary tree



Example: not complete binary tree



Example: not complete binary tree



First Advantage: Low Height

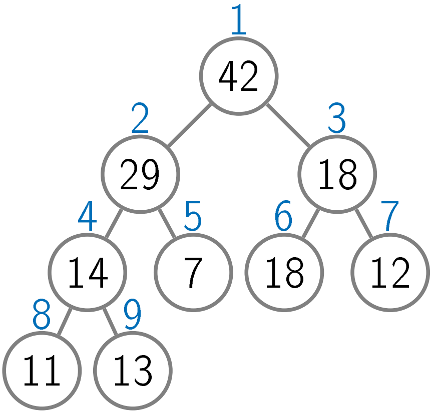
Lemma

A complete binary tree with n nodes has height at most $O(\log n)$.

Proof

- Complete the last level to get a full binary tree on $n' \geq n$ nodes and the same number of levels ℓ.
- Note that $n' \leq 2n$.
- Then $n' = 2^\ell - 1$ and hence $\ell = \log_2(n' + 1) \leq \log_2(2n + 1) = O(\log n)$. $\square$

Second Advantage: Store as Array



$\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$

$\text{leftchild}(i) = 2i$

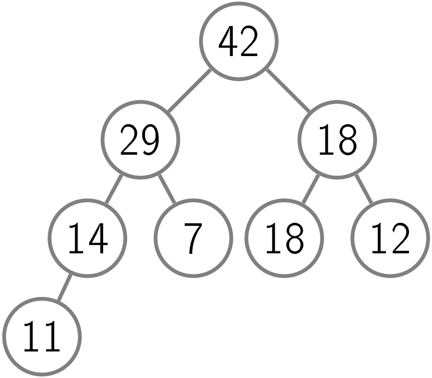
$\text{rightchild}(i) = 2i + 1$

1	2	3	4	5	6	7	8	9
42	29	18	14	7	18	12	11	5

- What do we pay for these advantages?
- We need to keep the tree complete.
- Which binary heap operations modify the shape of the tree?
- Only `Insert` and `ExtractMax` (`Remove` changes the shape by calling `ExtractMax`).

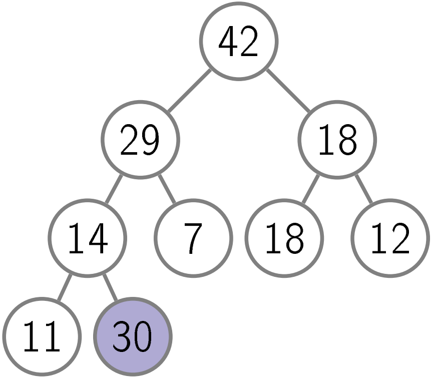
Keeping the Tree Complete

to insert an element, insert it as a leaf in the leftmost vacant position in the last level and let it sift up



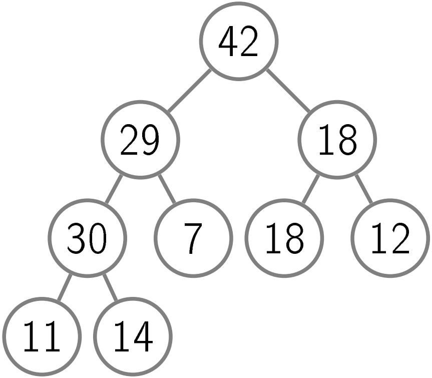
Keeping the Tree Complete

to insert an element, insert it as a leaf in the leftmost vacant position in the last level and let it sift up



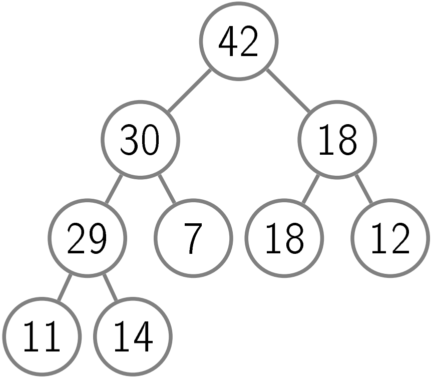
Keeping the Tree Complete

to insert an element, insert it as a leaf in the leftmost vacant position in the last level and let it sift up



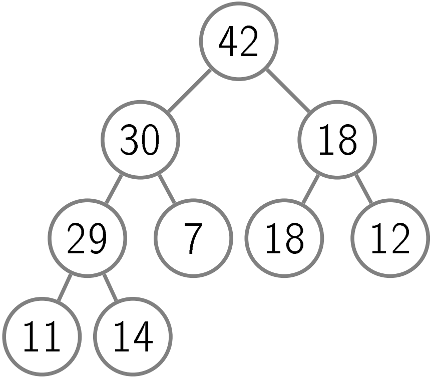
Keeping the Tree Complete

to insert an element, insert it as a leaf in the leftmost vacant position in the last level and let it sift up



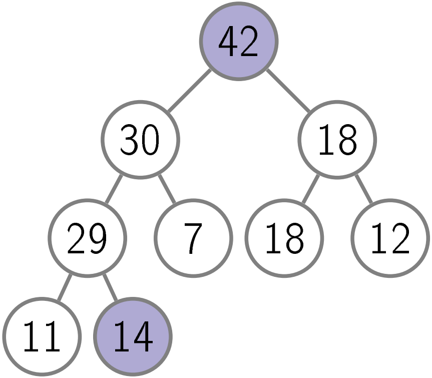
Keeping the Tree Complete

to extract the maximum value, replace the root by the last leaf and let it sift down



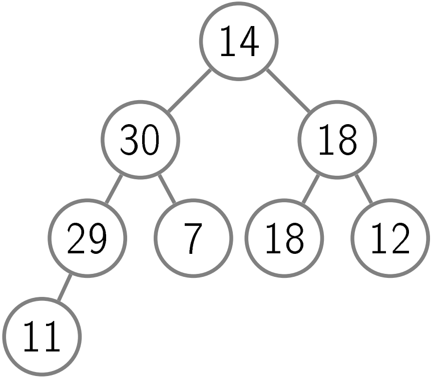
Keeping the Tree Complete

to extract the maximum value, replace the root by the last leaf and let it sift down



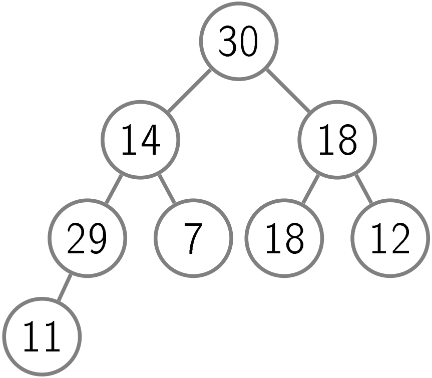
Keeping the Tree Complete

to extract the maximum value, replace the root by the last leaf and let it sift down



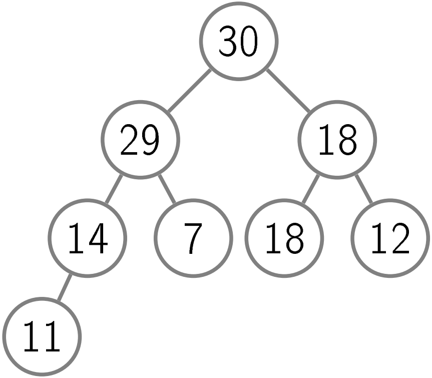
Keeping the Tree Complete

to extract the maximum value, replace the root by the last leaf and let it sift down



Keeping the Tree Complete

to extract the maximum value, replace the root by the last leaf and let it sift down

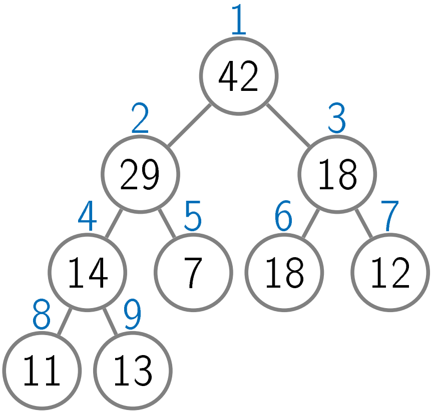


Outline

General Setting

- *maxSize* is the maximum number of elements in the heap
- *size* is the size of the heap
- $H[1 \dots maxSize]$ is an array of length *maxSize* where the heap occupies the first *size* elements

Example



$size = 9$

$maxSize = 13$

	1	2	3	4	5	6	7	8	9	10	11	12	13
H	42	29	18	14	7	18	12	11	5	30	29	2	8

Parent(i)

return $\lfloor \frac{i}{2} \rfloor$

LeftChild(i)

return $2i$

RightChild(i)

return $2i + 1$

SiftUp(i)

```
while i > 1 and H[Parent(i)] < H[i]:
   swap H[Parent(i)] and H[i]
   i ← Parent(i)
```

`SiftDown`(i)

```
maxIndex ← i
ℓ ← LeftChild(i)
if ℓ ≤ size and H[ℓ] > H[maxIndex]:
    maxIndex ← ℓ
r ← RightChild(i)
if r ≤ size and H[r] > H[maxIndex]:
    maxIndex ← r
if i ≠ maxIndex:
    swap H[i] and H[maxIndex]
    SiftDown(maxIndex)
```

Insert(p)

```
if size = maxSize:
  return ERROR
size ← size + 1
H[size] ← p
SiftUp(size)
```

ExtractMax()

$result \leftarrow H[1]$

$H[1] \leftarrow H[size]$

$size \leftarrow size - 1$

`SiftDown(1)`

`return` $result$

Remove(i)

$H[i] \leftarrow \infty$

SiftUp(i)

ExtractMax()

ChangePriority(i, p)

```
oldp ← H[i]
H[i] ← p
if p > oldp:
  SiftUp(i)
else:
  SiftDown(i)
```

Summary

The resulting implementation is

- fast: all operations work in time $O(\log n)$ (`GetMax` even works in $O(1)$)
- space efficient: we store an array of priorities; parent-child connections are not stored, but are computed on the fly
- easy to implement: all operations are implemented in just a few lines of code

Outline

Sort Using Priority Queues

HeapSort($A[1 \dots n]$)

```
create an empty priority queue
for i from 1 to n:
  Insert(A[i])
for i from n downto 1:
  A[i] ← ExtractMax()
```

- The resulting algorithms is comparison-based and has running time $O(n \log n)$ (hence, asymptotically optimal!).
- Natural generalization of selection sort: instead of simply scanning the rest of the array to find the maximum value, use a smart data structure.
- Not in-place: uses additional space to store the priority queue.

This lesson

In-place heap sort algorithm. For this, we will first turn a given array into a heap by permuting its elements.

Turn Array into a Heap

BuildHeap($A[1 \dots n]$)

$size \leftarrow n$

for i from $\lfloor n/2 \rfloor$ downto 1:

  SiftDown(i)

larger and larger subtrees

- We repair the heap property going from bottom to top.
- Initially, the heap property is satisfied in all the leaves (i.e., subtrees of depth 0).
- We then start repairing the heap property in all subtrees of depth 1.
- When we reach the root, the heap property is satisfied in the whole tree.
- Online visualization
- Running time: $O(n \log n)$

In-place Heap Sort

HeapSort($A[1 \dots n]$)

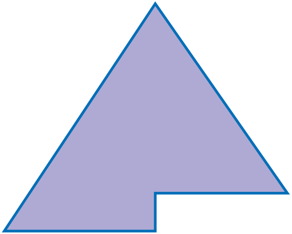
```
BuildHeap(A)                          {size = n}
repeat (n − 1) times:
    swap A[1] and A[size]
    size ← size − 1
    SiftDown(1)
```

ExtractMax()

Building Running Time

- The running time of `BuildHeap` is $O(n \log n)$ since we call `SiftDown` for $O(n)$ nodes.
- If a node is already close to the leaves, then sifting it down is fast.
- We have many such nodes!
- Was our estimate of the running time of `BuildHeap` too pessimistic?

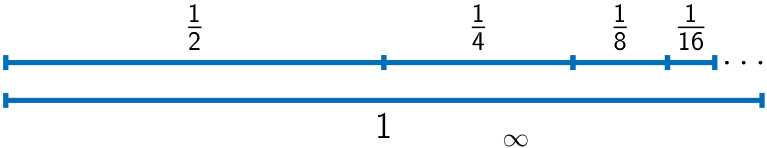
Building Running Time



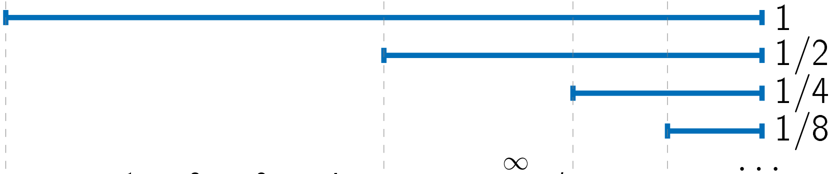
# nodes	$T(\texttt{SiftDown})$
1	$\log_2 n$
2	
$\vdots$	$\vdots$
$\leq n/4$	2
$\leq n/2$	1

$$\boxed{\begin{aligned} T(\texttt{BuildHeap}) &\leq \frac{n}{2}\cdot 1 + \frac{n}{4}\cdot 2 + \frac{n}{8}\cdot 3 + \dots \\ &\leq n \cdot \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n \end{aligned}}$$

Estimating the Sum



$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots = \sum_{k=1}^{\infty} \frac{1}{2^k} = 1$$



$$\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \ldots = \sum_{k=1}^{\infty} \frac{k}{2^k} = 2$$

Partial sorting

Input: An array $A[1 \dots n]$, an integer $1 \leq k \leq n$.

Output: The last k elements of a sorted version of A.

Can be solved in $O(n)$ if $k = O(\frac{n}{\log n})$!

```
PartialSorting(A[1...n], k)

BuildHeap(A)
for i from 1 to k:
    ExtractMax()
```

Running time: $O(n + k \log n)$

Summary

Heap sort is a time and space efficient comparison-based algorithm: has running time $O(n \log n)$, uses no additional space.

Outline

0-based Arrays

Parent(i)

return $\left\lfloor \frac{i-1}{2} \right\rfloor$

LeftChild(i)

return $2i + 1$

RightChild(i)

return $2i + 2$

Binary Min-Heap

Definition

Binary min-heap is a binary tree (each node has zero, one, or two children) where the value of each node is at most the values of its children.

Can be implemented similarly.

d-ary Heap

- In a d-ary heap nodes on all levels except for possibly the last one have exactly d children.
- The height of such a tree is about $\log_d n$.
- The running time of `SiftUp` is $O(\log_d n)$.
- The running time of `SiftDown` is $O(d \log_d n)$: on each level, we find the largest value among d children.

Summary

- Priority queue supports two main operations: `Insert` and `ExtractMax`.
- In an array/list implementation one operation is very fast ($O(1)$) but the other one is very slow ($O(n)$).
- Binary heap gives an implementation where both operations take $O(\log n)$ time.
- Can be made also space efficient.

Disjoint Sets: Naive Implementations

Alexander S. Kulikov

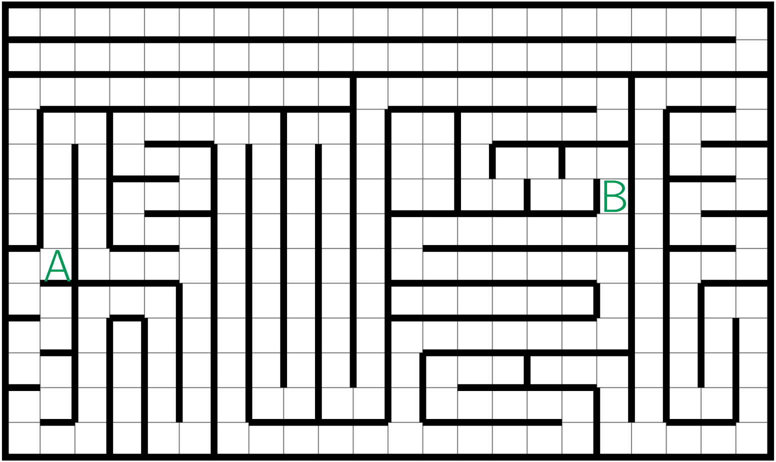
Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Data Structures
Data Structures and Algorithms

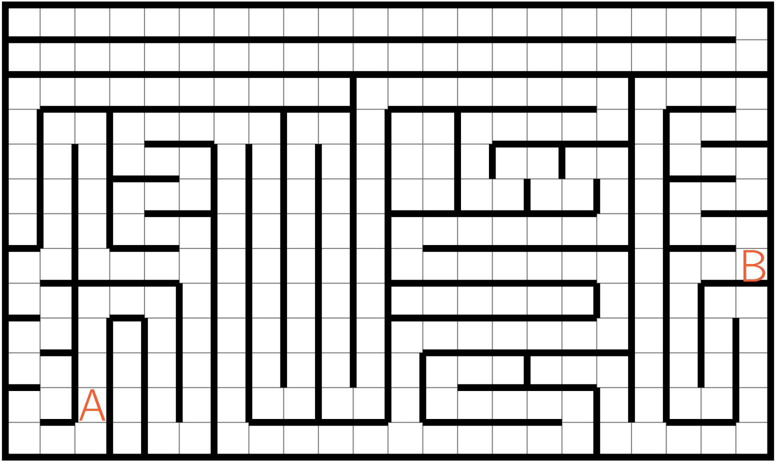
Outline

1. Overview
2. Naive Implementations

Maze: Is B Reachable from A?



Maze: Is B Reachable from A?



Definition

A disjoint-set data structure supports the following operations:

- `MakeSet`(x) creates a singleton set $\{x\}$
- `Find`(x) returns ID of the set containing x:
  - if x and y lie in the same set, then $\texttt{Find}(x) = \texttt{Find}(y)$
  - otherwise, $\texttt{Find}(x) \neq \texttt{Find}(y)$
- `Union`(x, y) merges two sets containing x and y

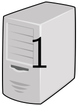
Preprocess(*maze*)

```
for each cell c in maze:
  MakeSet(c)
for each cell c in maze:
  for each neighbor n of c:
    Union(c, n)
```

IsReachable(A, B)

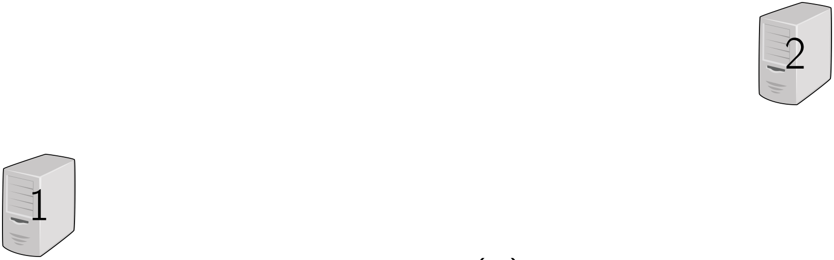
```
return Find(A) = Find(B)
```

Building a Network



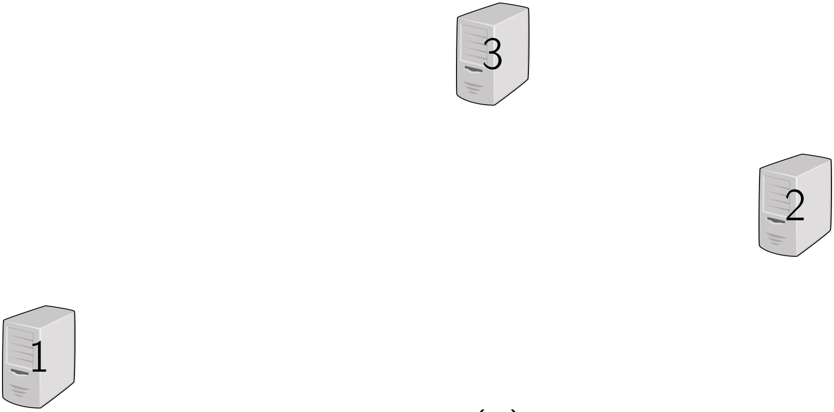
MakeSet(1)

Building a Network



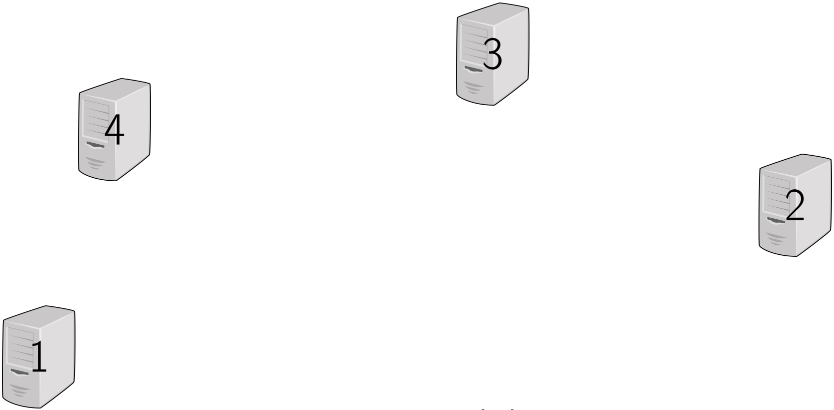
`MakeSet(2)`

Building a Network



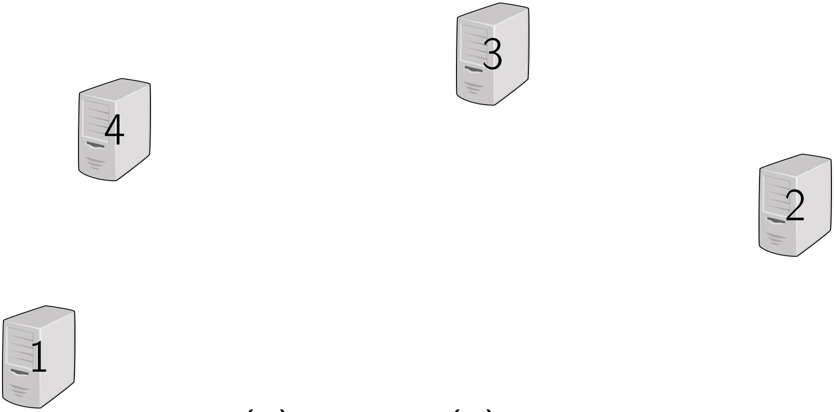
`MakeSet(3)`

Building a Network



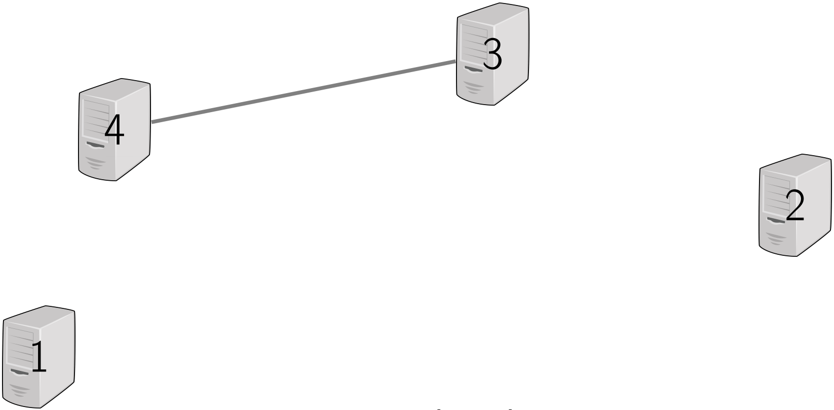
`MakeSet(4)`

Building a Network



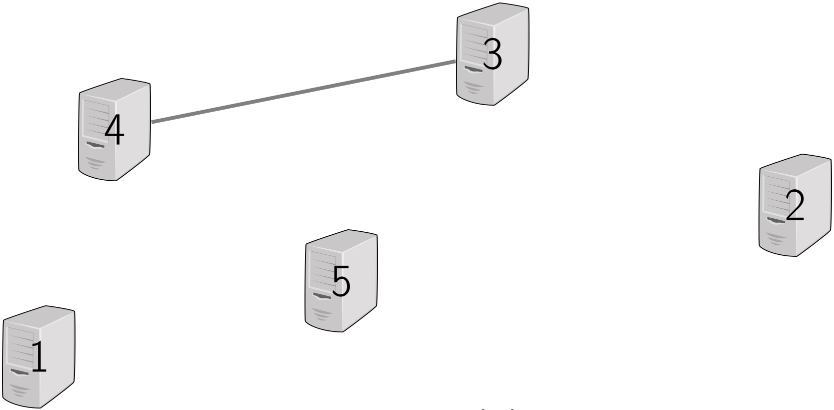
$$\texttt{Find}(1) = \texttt{Find}(2) \to \texttt{False}$$

Building a Network



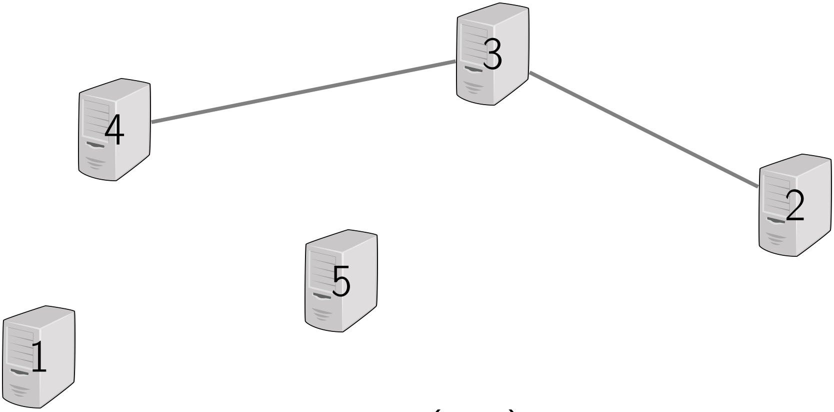
`Union(3,4)`

Building a Network



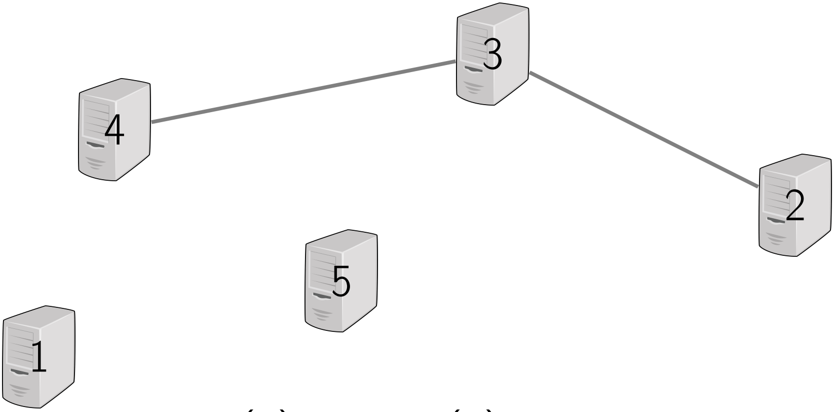
`MakeSet(5)`

Building a Network



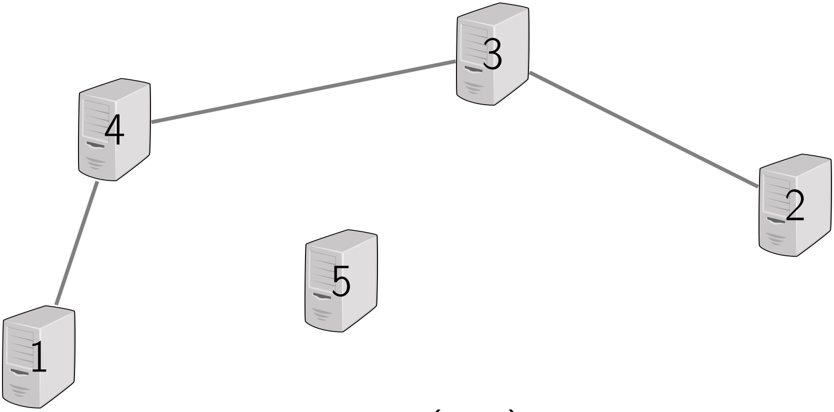
`Union(3,2)`

Building a Network



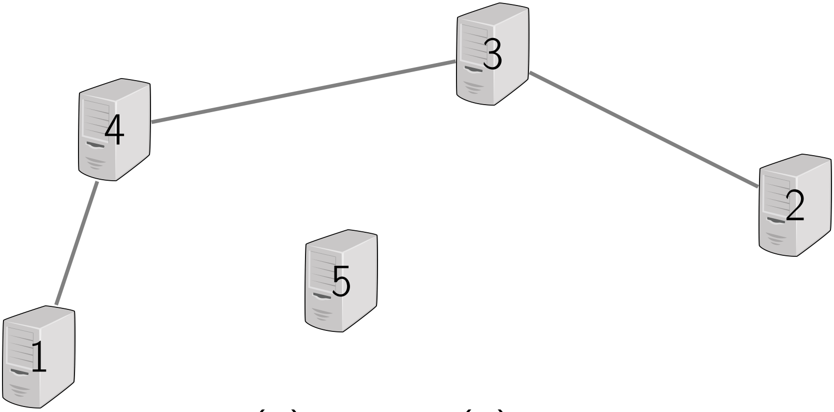
$$\texttt{Find}(1) = \texttt{Find}(2) \to \texttt{False}$$

Building a Network



`Union(1,4)`

Building a Network



$\texttt{Find}(1) = \texttt{Find}(2) \rightarrow \texttt{True}$

Outline

1. Overview
2. Naive Implementations

For simplicity, we assume that our n objects are just integers $1, 2, \ldots, n$.

Using the Smallest Element as ID

- Use the smallest element of a set as its ID
- Use array $\texttt{smallest}[1 \dots n]$: $\texttt{smallest}[i]$ stores the smallest element in the set i belongs to

Example

$$\{9, 3, 2, 4, 7\} \quad \{5\} \quad \{6, 1, 8\}$$

	1	2	3	4	5	6	7	8	9
`smallest`	1	2	2	2	5	1	2	1	2

```
MakeSet(i)
smallest[i] ← i
```

```
Find(i)
return smallest[i]
```

Running time: $O(1)$

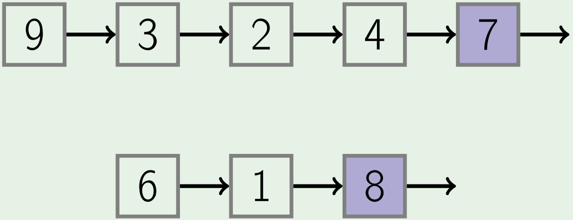
$\texttt{Union}(i, j)$

```
i_id ← Find(i)
j_id ← Find(j)
if i_id = j_id:
  return
m ← min(i_id, j_id)
for k from 1 to n:
  if smallest[k] in {i_id, j_id}:
    smallest[k] ← m
```

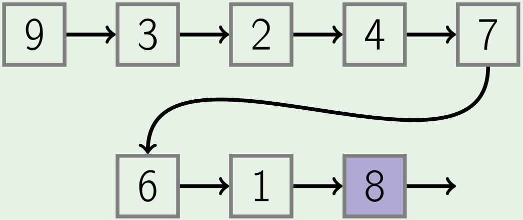
Running time: $O(n)$

- Current bottleneck: `Union`
- What basic data structure allows for efficient merging?
- Linked list!
- Idea: represent a set as a linked list, use the list tail as ID of the set

Example: merging two lists

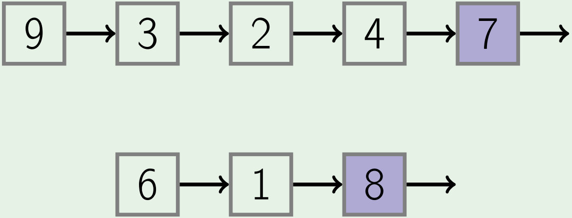


Example: merging two lists

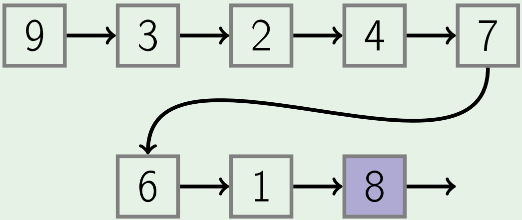


- Pros:
  - Running time of `Union` is $O(1)$
  - Well-defined ID
- Cons:
  - Running time of `Find` is $O(n)$ as we need to traverse the list to find its tail
  - `Union`(x, y) works in time $O(1)$ only if we can get the tail of the list of x and the head of the list of y in constant time!

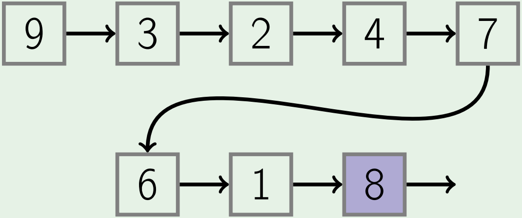
Example: merging two lists



Example: merging two lists

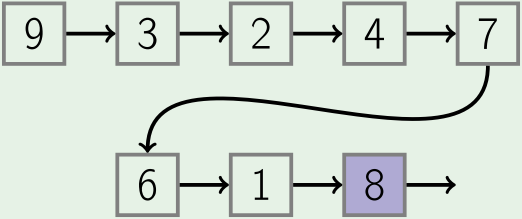


Example: merging two lists



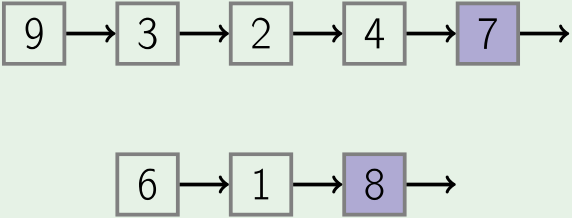
`Find`(9) goes through all elements

Example: merging two lists

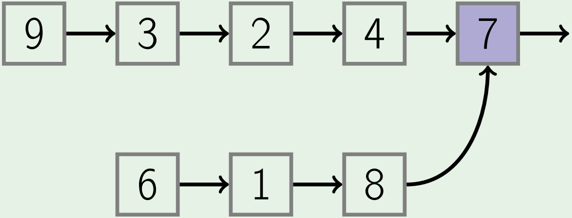


can we merge in a different way?

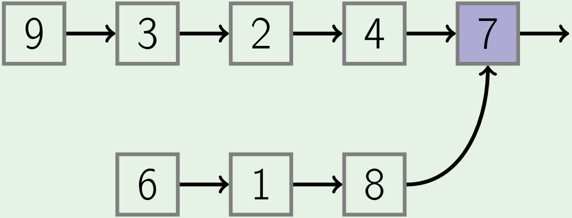
Example: merging two lists



Example: merging two lists

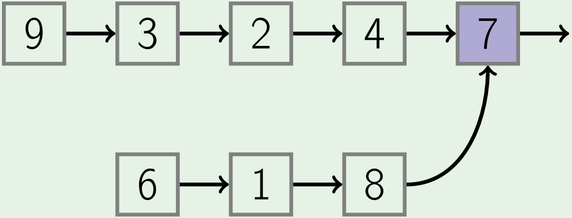


Example: merging two lists



instead of a list we get a tree

Example: merging two lists



we'll see that representing sets as trees gives a very efficient implementation: nearly constant amortized time for all operations

Disjoint Sets: Efficient Implementations

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

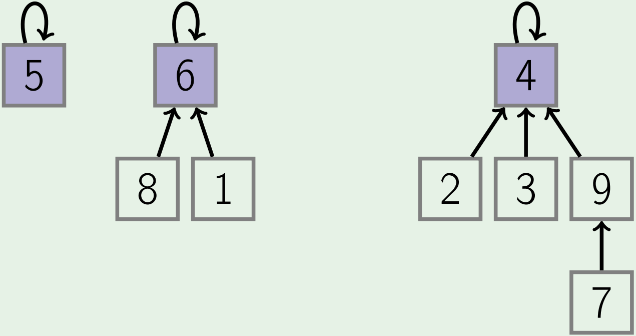
Data Structures
Data Structures and Algorithms

Outline

1. Trees
2. Union by Rank
3. Path Compression
4. Analysis

- Represent each set as a rooted tree
- ID of a set is the root of the tree
- Use array `parent`$[1 \ldots n]$: `parent`$[i]$ is the parent of i, or i if it is the root

Example



	1	2	3	4	5	6	7	8	9
parent	6	4	4	4	5	6	9	6	4

```
MakeSet(i)
```

$\texttt{parent}[i] \leftarrow i$

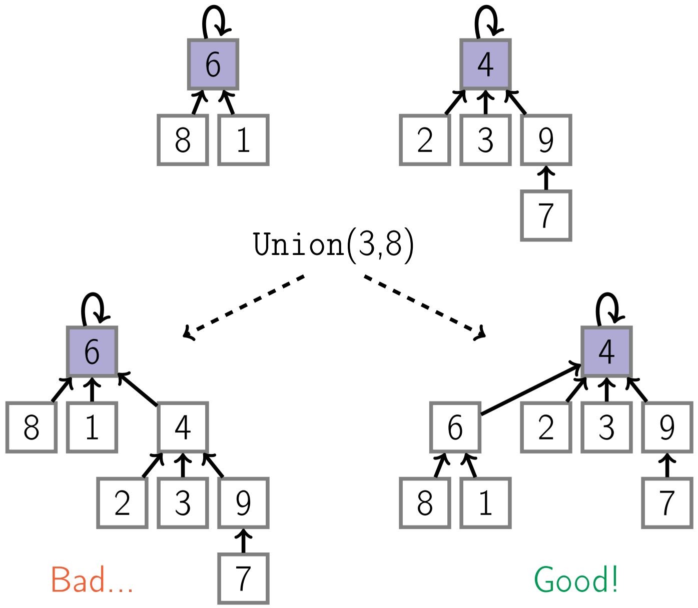
Running time: $O(1)$

```
Find(i)
```

$\texttt{while } i \neq \texttt{parent}[i]$:
  $i \leftarrow \texttt{parent}[i]$
$\texttt{return } i$

Running time: $O(\text{tree height})$

- How to merge two trees?
- Hang one of the trees under the root of the other one
- Which one to hang?
- A shorter one, since we would like to keep the trees shallow


6
8
1
4
2
3
9
7
Union(3,8)
6
8
1
4
2
3
9
7
Bad...
4
6
2
3
9
8
1
7
Good!

Outline

- When merging two trees we hang a shorter one under the root of a taller one
- To quickly find a height of a tree, we will keep the height of each subtree in an array `rank`$[1 \dots n]$: `rank`$[i]$ is the height of the subtree whose root is i
- (The reason we call it `rank`, but not `height` will become clear later)
- Hanging a shorter tree under a taller one is called a union by rank heuristic

MakeSet(i)

```
parent[i] ← i
rank[i] ← 0
```

Find(i)

```
while i ≠ parent[i]:
    i ← parent[i]
return i
```

Union(i, j)

```
i_id ← Find(i)
j_id ← Find(j)
if i_id = j_id:
   return
if rank[i_id] > rank[j_id]:
   parent[j_id] ← i_id
else:
   parent[i_id] ← j_id
   if rank[i_id] = rank[j_id]:
      rank[j_id] ← rank[j_id] + 1
```

Example

Query:

`MakeSet(1)`

`MakeSet(2)`

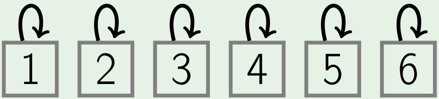
`...`

`MakeSet(6)`

	1	2	3	4	5	6
parent						
rank						

Example

Query:

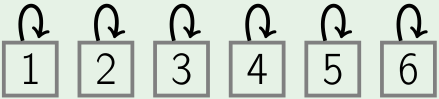


	1	2	3	4	5	6
parent	1	2	3	4	5	6
rank	0	0	0	0	0	0

Example

Query:

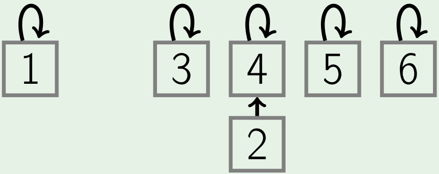
`Union`$(2, 4)$



	1	2	3	4	5	6
`parent`	1	2	3	4	5	6
`rank`	0	0	0	0	0	0

Example

Query:

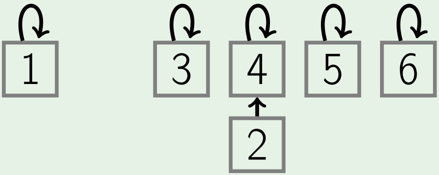


	1	2	3	4	5	6
`parent`	1	4	3	4	5	6
`rank`	0	0	0	1	0	0

Example

Query:

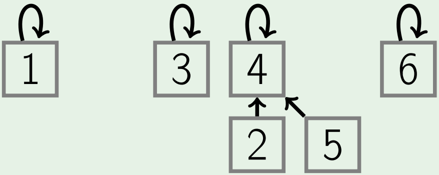
`Union(5, 2)`



	1	2	3	4	5	6
parent	1	4	3	4	5	6
rank	0	0	0	1	0	0

Example

Query:

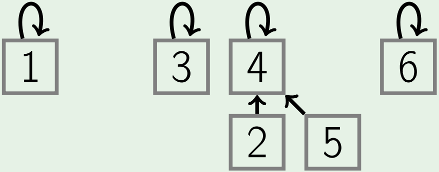


	1	2	3	4	5	6
`parent`	1	4	3	4	4	6
`rank`	0	0	0	1	0	0

Example

Query:

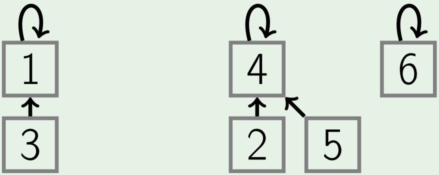
`Union`$(3, 1)$



	1	2	3	4	5	6
`parent`	1	4	3	4	4	6
`rank`	0	0	0	1	0	0

Example

Query:

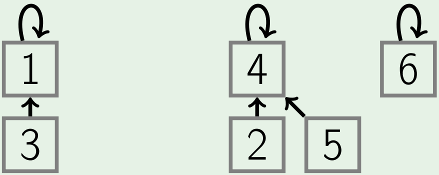


	1	2	3	4	5	6
`parent`	1	4	1	4	4	6
`rank`	1	0	0	1	0	0

Example

Query:

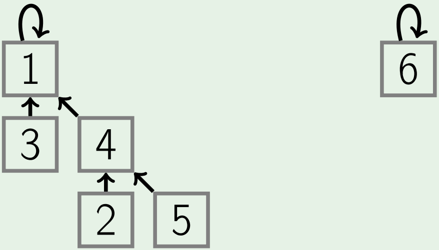
Union(2, 3)



	1	2	3	4	5	6
parent	1	4	1	4	4	6
rank	1	0	0	1	0	0

Example

Query:

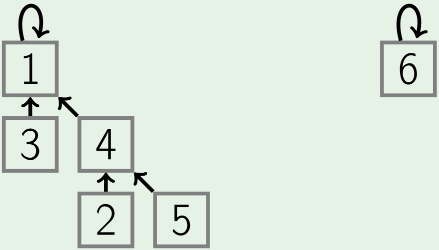


	1	2	3	4	5	6
`parent`	1	4	1	1	4	6
`rank`	2	0	0	1	0	0

Example

Query:

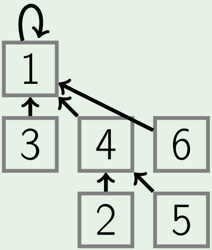
`Union(2, 6)`



	1	2	3	4	5	6
`parent`	1	4	1	1	4	6
`rank`	2	0	0	1	0	0

Example

Query:



	1	2	3	4	5	6
`parent`	1	4	1	1	4	1
`rank`	2	0	0	1	0	0

Important property: for any node i, rank[i] is equal to the height of the tree rooted at i

Lemma

The height of any tree in the forest is at most $\log_2 n$.

Follows from the following lemma.

Lemma

Any tree of height k in the forest has at least 2^k nodes.

Proof

Induction on k.

- Base: initially, a tree has height 0 and one node: $2^0 = 1$.
- Step: a tree of height k results from merging two trees of height $k - 1$. By induction hypothesis, each of two trees has at least 2^{k-1} nodes, hence the resulting tree contains at least 2^k nodes. □

Summary

The union by rank heuristic guarantees that `Union` and `Find` work in time $O(\log n)$.

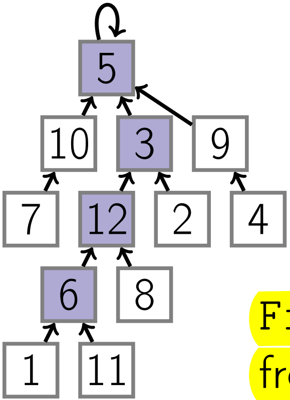
Next part

We'll discover another heuristic that improves the running time to nearly constant!

Outline

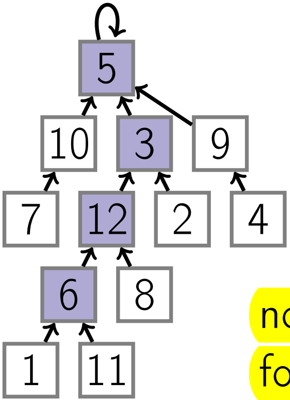
1. Trees
2. Union by Rank
3. Path Compression
4. Analysis

Path Compression: Intuition



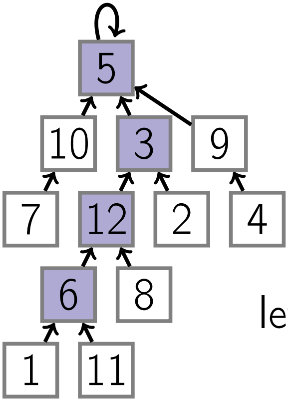
`Find`(6) traverses the path from 6 to the root

Path Compression: Intuition



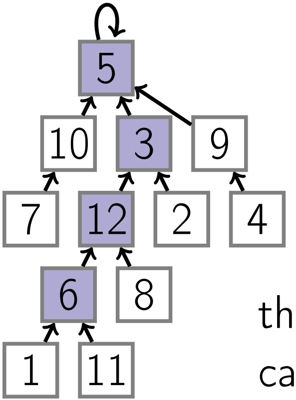
not only it finds the root for 6, it does so for all the nodes on this path

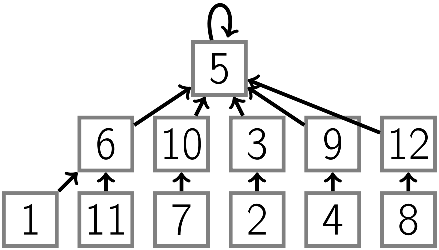
Path Compression: Intuition



let's not lose this useful info

Path Compression: Intuition





the resulting heuristic is called path compression

```
Find(i)

if i ≠ parent[i]:
   parent[i] ← Find(parent[i])
return parent[i]
```

Definition

The iterated logarithm of n, $\log^* n$, is the number of times the logarithm function needs to be applied to n before the result is less or equal than 1.

Example

n	$\log^* n$
$n = 1$	0
$n = 2$	1
$n \in \{3, 4\}$	2
$n \in \{5, 6, \ldots, 16\}$	3
$n \in \{17, \ldots, 65536\}$	4
$n \in \{65537, \ldots, 2^{65536}\}$	5

Lemma

Assume that initially the data structure is empty. We make a sequence of m operations including n calls to `MakeSet`. Then the total running time is $O(m \log^* n)$.

In other words

The amortized time of a single operation is $O(\log^* n)$.

Nearly constant!

$\boxed{\text{For practical values of } n,\ \log^* n \leq 5.}$

Outline

1. Trees
2. Union by Rank
3. Path Compression
4. Analysis

Goal

Prove that when both union by rank heuristic and path compression heuristic are used, the average running time of each operation is nearly constant.

Height $\leq$ Rank

- When using path compression, $\texttt{rank}[i]$ is no longer equal to the height of the subtree rooted at i
- Still, the height of the subtree rooted at i is at most $\texttt{rank}[i]$
- And it is still true that a root node of rank k has at least 2^k nodes in its subtree: a root node is not affected by path compression

Important Properties

1. There are at most $\frac{n}{2^k}$ nodes of rank k
2. For any node i, $\texttt{rank}[i] < \texttt{rank}[\texttt{parent}[i]]$
3. Once an internal node, always an internal node

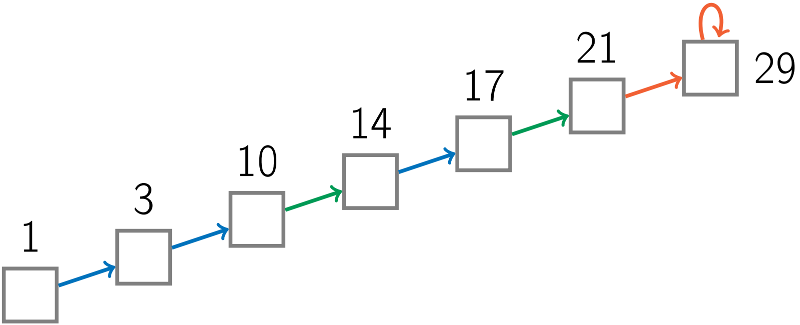
$T(\text{all calls to } \texttt{Find}) =$

$\#(i \to j) =$

$\#(i \to j : j \text{ is a root}) +$

$\#(i \to j : \log^*(\texttt{rank}[i]) < \log^*(\texttt{rank}[j])) +$

$\#(i \to j : \log^*(\texttt{rank}[i]) = \log^*(\texttt{rank}[j]))$



Claim

$\#(i \to j : j \text{ is a root}) \leq O(m)$

Proof

There are at most m calls to `Find`. $\square$

Claim

$$\#(i \to j : \log^*(\texttt{rank}[i]) < \log^*(\texttt{rank}[j])) \\ \leq O(m \log^* n)$$

Proof

There are at most $\log^* n$ different values for $\log^*(\texttt{rank})$. $\square$

Claim

$$\#(i \to j\colon\ \log^*(\mathtt{rank}[i]) = \log^*(\mathtt{rank}[j])) \leq O(n \log^* n)$$

Proof

- assume $\texttt{rank}[i] \in \{k+1, \ldots, 2^k\}$
- the number of nodes with rank lying in this interval is at most

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}$$

- after a call to $\texttt{Find}(i)$, the node i is adopted by a new parent of strictly larger rank
- after at most 2^k calls to $\texttt{Find}(i)$, the parent of i will have rank from a different interval

Proof (Continued)

- there are at most $\frac{n}{2^k}$ nodes with rank in $\{k+1, \dots, 2^k\}$
- each of them contributes at most 2^k
- the contribution of all the nodes with rank from this interval is at most $O(n)$
- the number of different intervals is $\log^* n$
- thus, the contribution of all nodes is $O(n \log^* n)$ $\square$

Summary

- Represent each set as a rooted tree
- Use the root of the set as its ID
- Union by rank heuristic: hang a shorter tree under the root of a taller one
- Path compression heuristic: when finding the root of a tree for a particular node, reattach each node from the traversed path to the root
- Amortized running time: $O(\log^* n)$ (constant for practical values of n)