Introduction to Algorithms: Lecture 4

Xue Chen
xuechen1989@ustc.edu.cn
2025 spring in

HW & Experiment

1. HW 2 & Experiment 1 are out
2. Office hours of Week 4 and Week 5 are in classroom 3A103

Outline

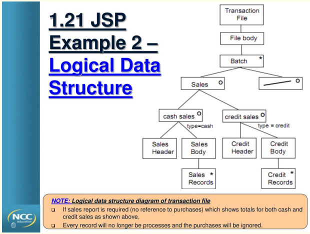
Data Structure

A data structure is a specific way to organize data such that these data can be used efficiently.

Data Structure

A data structure is a specific way to organize data such that these data can be used efficiently.

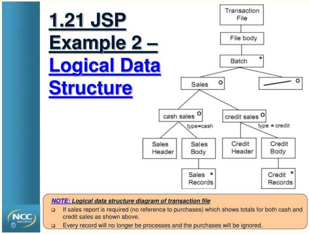
1. Many models: logical, mathematical, . . . ,

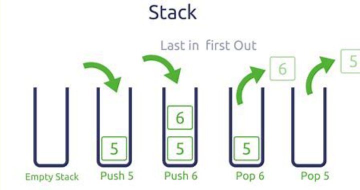


Data Structure

A data structure is a specific way to organize data such that these data can be used efficiently.

1. Many models: logical, mathematical, . . . ,



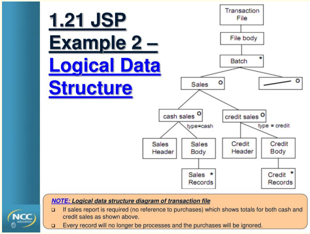


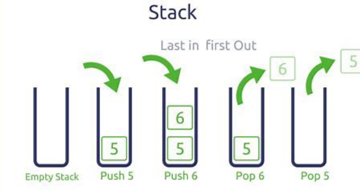
2. Different types: Linear (queues, stacks, linked lists, . . .), Non-Linear (trees,graphs,. . .)

Data Structure

A data structure is a specific way to organize data such that these data can be used efficiently.

1. Many models: logical, mathematical, . . . ,

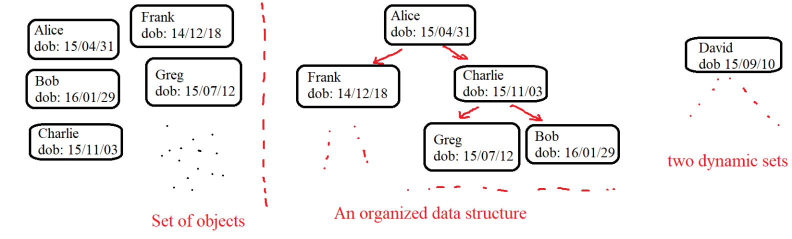




2. Different types: Linear (queues, stacks, linked lists, . . .), Non-Linear (trees,graphs,. . .)
3. Other properties: static vs dynamic, homogenous, . . .

Overview

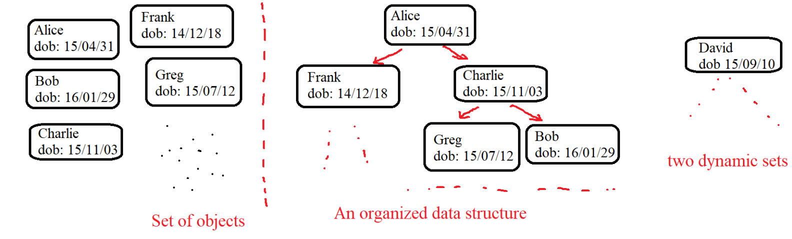
This course considers data structures as a way to represent finite dynamic sets (of various elements).



The goal is

Overview

This course considers data structures as a way to represent finite dynamic sets (of various elements).



The goal is

1. Maintain it efficiently
2. Answer queries like (1) who is the oldest kid? (2) How many kids born in August 2015? ...

Overview (II)

Each element has a unique ID/pointer (like “Alice”) and a key value $k \in \mathbb{Z}$ with a total order (like “dob: 15/04/31”).

Standard Operations on dynamic sets

Overview (II)

Each element has a unique ID/pointer (like “Alice”) and a key value $k \in \mathbb{Z}$ with a total order (like “dob: 15/04/31”).

Standard Operations on dynamic sets

1. $\textsc{Search}(S, k)$: Return a pointer x to an element in S with value k or NIL
2. $\textsc{Insert}(S, x)$: Insert the element pointed by x to S
3. $\textsc{Delete}(S, x)$: Delete element pointed by x from S

Overview (II)

Each element has a unique ID/pointer (like "Alice") and a key value $k \in \mathbb{Z}$ with a total order (like "dob: 15/04/31").

Standard Operations on dynamic sets

1. SEARCH(S, k): Return a pointer x to an element in S with value k or NIL
2. INSERT(S, x): Insert the element pointed by x to S
3. DELETE(S, x): Delete element pointed by x from S
4. MINIMUM(S): Return a point to element in S with the smallest key
5. SUCCESSOR(S, x): Given x, return the next element after x in S

Overview (II)

Each element has a unique ID/pointer (like "Alice") and a key value $k \in \mathbb{Z}$ with a total order (like "dob: 15/04/31").

Standard Operations on dynamic sets

1. $\textsc{Search}(S, k)$: Return a pointer x to an element in S with value k or NIL
2. $\textsc{Insert}(S, x)$: Insert the element pointed by x to S
3. $\textsc{Delete}(S, x)$: Delete element pointed by x from S
4. $\textsc{Minimum}(S)$: Return a point to element in S with the smallest key
5. $\textsc{Successor}(S, x)$: Given x, return the next element after x in S
6. $\textsc{Union}(S, T)$: Unites the two dynamic sets S and T
7. $\textsc{Count}(S, k_1, k_2)$: Given a total order and an interval $[k_1, k_2]$, return the number of elements in S with a key value k in $[k_1, k_2]$.

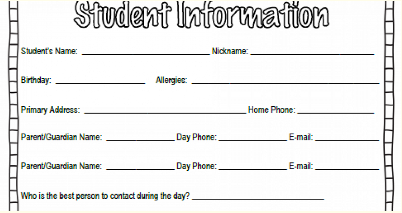
Outline

Introduction

Hash maintains a dynamic set S of keys for the dictionary problem

Example

Maintain all students information — enroll, graduate, search by id and name, ...


Student Information

Student's Name: ____________ Nickname: ____________

Birthday: ____________ Allergies: ____________

Primary Address: ____________ Home Phone: ____________

Parent/Guardian Name: ____________ Day Phone: ____________ E-mail: ____________

Parent/Guardian Name: ____________ Day Phone: ____________ E-mail: ____________

Who is the best person to contact during the day? ____________

Description

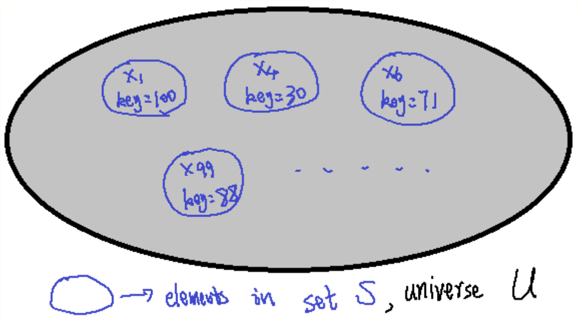
$x.key$ denotes the unique key of each element x — for convenience, only consider one key in $\mathbb{Z}$.

1. SEARCH(k): Find x in S with $x.key = k$
2. DELETE(x): Delete x in S
3. INSERT(x): Insert x in S

Description

$x.key$ denotes the unique key of each element x — for convenience, only consider one key in $\mathbb{Z}$.

1. SEARCH(k): Find x in S with $x.key = k$
2. DELETE(x): Delete x in S
3. INSERT(x): Insert x in S



Notations

1. U: the domain of keys
2. S: the dynamic set in U
3. m: the max elements in S

Think $|U| = 10^9$ and $m = 10^3$ or 10^6

Notations

1. U: the domain of keys
2. S: the dynamic set in U
3. m: the max elements in S

Think $|U| = 10^9$ and $m = 10^3$ or 10^6

Questions

(1) If we only want to support SEARCH, INSERT, DELETE in $O(1)$ time, what shall we do?
(2) If we only have $O(m)$ storage-space, what shall we do?

Notations

1. U: the domain of keys
2. S: the dynamic set in U
3. m: the max elements in S

Think $|U| = 10^9$ and $m = 10^3$ or 10^6

Questions

(1) If we only want to support SEARCH, INSERT, DELETE in $O(1)$ time, what shall we do?
(2) If we only have $O(m)$ storage-space, what shall we do?

1. Solution 1: Maintain $T : U \to$ node s.t. $T[x.key] = x$ for any x

Notations

1. U: the domain of keys
2. S: the dynamic set in U
3. m: the max elements in S

Think $|U| = 10^9$ and $m = 10^3$ or 10^6

Questions

(1) If we only want to support SEARCH, INSERT, DELETE in $O(1)$ time, what shall we do?
(2) If we only have $O(m)$ storage-space, what shall we do?

1. Solution 1: Maintain $T : U \rightarrow$ node s.t. $T[x.key] = x$ for any x
2. Solution 2: Maintain an array or a chain.

Hash

Hash tries to get the best parts of both — $O(1)$ time and $O(m)$ space.

Hash

Hash tries to get the best parts of both — $O(1)$ time and $O(m)$ space.

Solution

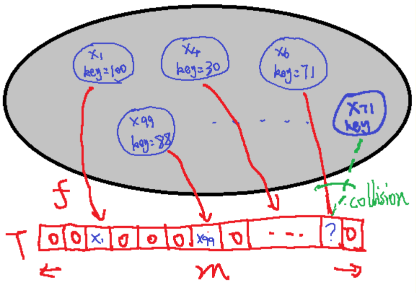
1. Prepare a hash function $f : U \to [m]$
   like $f_{a,b}(x) = (a \cdot x.key + b) \mod m$
2. Maintain a truth table $T : [m] \to$ node

Hash

Hash tries to get the best parts of both — $O(1)$ time and $O(m)$ space.

Solution

1. Prepare a hash function $f : U \to [m]$
   like $f_{a,b}(x) = (a \cdot x.key + b) \mod m$
2. Maintain a truth table $T : [m] \to$ node



Assume no collision, how to support SEARCH, INSERT, DELETE?

Handle Collisions (I)

Collisions are unavoidable unless $|S| \leqslant \sqrt{m}$ — birthday paradox.

Theorem

For a perfectly random hash h,

$$Pr_h\left[h(x_1), \ldots, h(x_k) \text{ have no collision}\right] \leqslant e^{-\binom{k}{2}/m}.$$

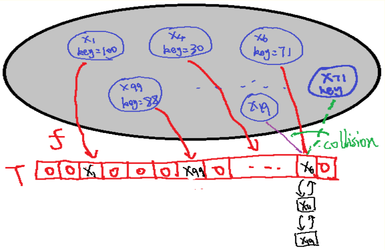
Handle Collisions (I)

Collisions are unavoidable unless $|S| \leqslant \sqrt{m}$ — birthday paradox.

Theorem

For a perfectly random hash h,

$$Pr_h\left[h(x_1),\ldots,h(x_k) \text{ have no collision}\right] \leqslant e^{-\binom{k}{2}/m}.$$



For each node x, introduce *x.left* and *x.right* to maintain the chain

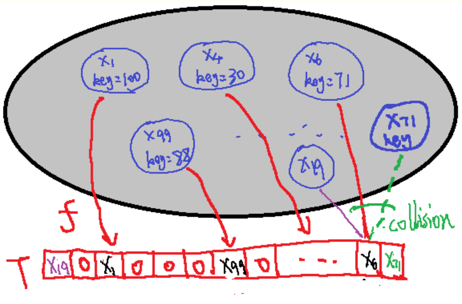
Solution 1

Maintain a chain for each position in T

Collisions 2

Solution 2

Put x into the next empty box in T — open-addressing method



Solution 3: Power of 2-choice

A surprising powerful idea

Prepare multiple hash functions, say h_1 and h_2, and put x into $h_1(x)$ or $h_2(x)$

1. Multiple choices hash
2. Always-Go-Left Hash
3. Cuckoo Hash: When $n < m/2$, at most one ball in every bin.

Summary

As a warm-up, Hash is the most fundamental data structure in CS

1. Collisions are unavoidable unless the table is huge.

Summary

As a warm-up, Hash is the most fundamental data structure in CS

1. Collisions are unavoidable unless the table is huge.
2. Solution 1, maintaining chains in T, is more time-efficient but waste $\Omega(1)$-frac of space.
3. Solution 2, open-addressing method, is more space efficient but slower.

Summary

As a warm-up, Hash is the most fundamental data structure in CS

1. Collisions are unavoidable unless the table is huge.
2. Solution 1, maintaining chains in T, is more time-efficient but waste $\Omega(1)$-frac of space.
3. Solution 2, open-addressing method, is more space efficient but slower.
4. Solution 3, multiple choices hash and cuckoo hash.

Summary

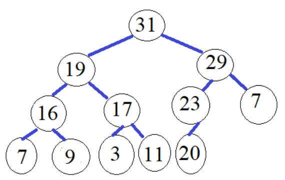
As a warm-up, Hash is the most fundamental data structure in CS

1. Collisions are unavoidable unless the table is huge.
2. Solution 1, maintaining chains in T, is more time-efficient but waste $\Omega(1)$-frac of space.
3. Solution 2, open-addressing method, is more space efficient but slower.
4. Solution 3, multiple choices hash and cuckoo hash.
5. More: memory-hierarchy hash, ...

Outline

Introduction

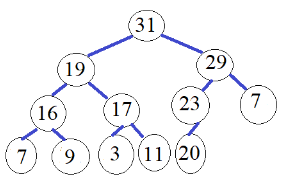
A heap is a nearly complete binary tree that supports MAXIMUM, INSERT, DELETE of a set with ordered keys in time $O(\log n)$.



Main Property

Introduction

A heap is a nearly complete binary tree that supports MAXIMUM, INSERT, DELETE of a set with ordered keys in time $O(\log n)$.

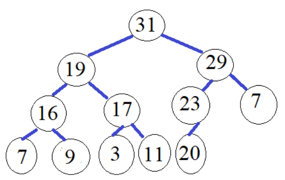


Main Property

1. For any node v (not root), parent[v].key $\geqslant$ v.key
   $\Rightarrow$ the maximal key value is at the root, called max-heap.
2. Nearly-complete & Almost-balanced: Most nodes (except $\leqslant 1$ node) have either 2 children (called v.left and v.right) or 0.

Introduction

A heap is a nearly complete binary tree that supports MAXIMUM, INSERT, DELETE of a set with ordered keys in time $O(\log n)$.



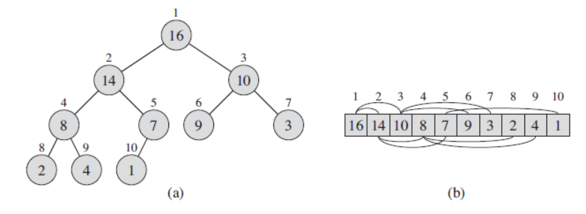
Main Property

1. For any node v (not root), parent[v].key $\geqslant$ v.key
   $\Rightarrow$ the maximal key value is at the root, called max-heap.
2. Nearly-complete & Almost-balanced: Most nodes (except $\leqslant 1$ node) have either 2 children (called v.left and v.right) or 0.

Remarks: (1) Only consider one dynamic set and $n :=$ its size. (2) Neglect corner cases in these slides.

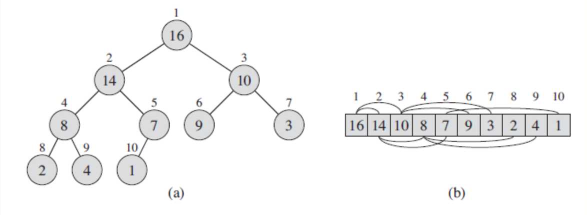
Details

Two ways to view the heap because it is nearly complete.



Details

Two ways to view the heap because it is nearly complete.



Implement it as an array

For each node with label i,

$$\begin{aligned} Key(i):&\quad A[i] \\ Parent(i):&\quad \lfloor i/2 \rfloor \\ Left(i):&\quad 2i \\ Right(i):&\quad 2i+1. \end{aligned}$$

Remark: $Left(i) = \emptyset$ if $2i > n$ and vice versa for $Right(i)$.

Maintaining the heap

Operations

The binary heap supports the following operations in $O(\log n)$ time except BUILD-MAX-HEAP in $O(n)$ time:

1. HEAP-MAXIMUM(A): *return* $A[1]$

Maintaining the heap

Operations

The binary heap supports the following operations in $O(\log n)$ time except BUILD-MAX-HEAP in $O(n)$ time:

1. HEAP-MAXIMUM(A): *return* $A[1]$
2. MAX-HEAPIFY(A, i): Decrease the value of $A[i]$ and adjust
3. HEAP-EXTRACT-MAX(A): Remove the largest element (root) in A
4. HEAP-INCREASE-KEY(A, i): Increase the value of $A[i]$ and adjust

Maintaining the heap

Operations

The binary heap supports the following operations in $O(\log n)$ time except BUILD-MAX-HEAP in $O(n)$ time:

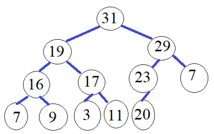
1. HEAP-MAXIMUM(A): *return* $A[1]$
2. MAX-HEAPIFY(A, i): Decrease the value of $A[i]$ and adjust
3. HEAP-EXTRACT-MAX(A): Remove the largest element (root) in A
4. HEAP-INCREASE-KEY(A, i): Increase the value of $A[i]$ and adjust
5. MAX-HEAP-INSERT(A, k): Insert a new element with key value k
6. BUILD-MAX-HEAP(A): Build array A as a heap.

Maintaining the heap

Operations

The binary heap supports the following operations in $O(\log n)$ time except BUILD-MAX-HEAP in $O(n)$ time:

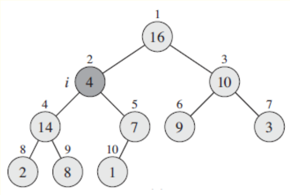
1. HEAP-MAXIMUM(A): *return* $A[1]$
2. MAX-HEAPIFY(A, i): Decrease the value of $A[i]$ and adjust
3. HEAP-EXTRACT-MAX(A): Remove the largest element (root) in A
4. HEAP-INCREASE-KEY(A, i): Increase the value of $A[i]$ and adjust
5. MAX-HEAP-INSERT(A, k): Insert a new element with key value k
6. BUILD-MAX-HEAP(A): Build array A as a heap.



Key property: the height h is $\leqslant \lceil \log_2 n \rceil + 1$.

MAX-HEAPIFY

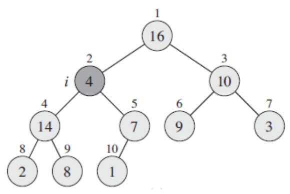
Task: After decreasing $A[i]$, maintain A as a heap.



Example: Adjust $A[2]$ to maintain the properties of a heap

MAX-HEAPIFY

Task: After decreasing $A[i]$, maintain A as a heap.



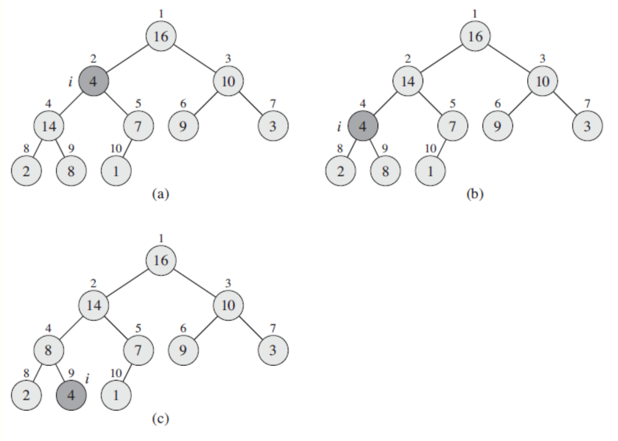
Example: Adjust $A[2]$ to maintain the properties of a heap

Recall two main properties: (1) $A[i] \geqslant A[2i]$ & $A[2i+1]$ (2) a nearly complete binary tree

---

procedure MAX-HEAPIFY(A, i)
  while $A[i] \leqslant A[2i]$ or $A[i] \leqslant A[2i+1]$ **do**
    $largest = \arg\max\{A[2i], A[2i+1]\}$
    Exchange $A[i]$ with $A[largest]$
    $i = largest$

Example run of MAX-HEAPIFY



Example: Adjust $A[2]$ to maintain the properties of a heap

BUILD-MAX-HEAP

Task: Input $A[1], \ldots, A[n]$, adjust them to make it a heap.

BUILD-MAX-HEAP

Task: Input $A[1], \ldots, A[n]$, adjust them to make it a heap.

---

procedure BUILD-MAX-HEAP(A)
  for $i = [n/2], \ldots, 1$ **do**
    MAX-HEAPIFY(A, i)

BUILD-MAX-HEAP

Task: Input $A[1], \ldots, A[n]$, adjust them to make it a heap.

---

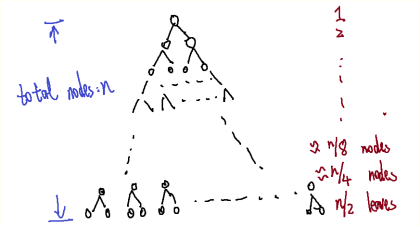
procedure BUILD-MAX-HEAP(A)
  for $i = [n/2], \ldots, 1$ **do**
    MAX-HEAPIFY(A, i)

Remarks

1. The loop i has to go from $[n/2]$ to 1. Otherwise it violates 1st property.
2. Question: Running time?

Running time of BUILD-MAX-HEAP

While the total height is $\log_2 n$, only 1 node (root) will have $\log_2 n$ exchanges possibly; only 2 nodes will have $\log_2 n - 1$ exchanges possibly, ...



$$1 \cdot \log_2 n + 2 \cdot (\log_2 n - 1) + \ldots + n/4 \cdot 1 = O(n).$$

HEAP-EXTRACT-MAX

Task: Remove $A[1]$ from A — decrease n and maintain A as a heap

HEAP-EXTRACT-MAX

Task: Remove $A[1]$ from A — decrease n and maintain A as a heap

procedure HEAP-EXTRACT-MAX(A)

$\quad \mathsf{max} = A[1]$

$\quad A[1] = A[n]$

$\quad n = n - 1$

$\quad$ MAX-HEAPIFY(A, 1)

$\quad$ Return max

HEAP-EXTRACT-MAX

Task: Remove $A[1]$ from A — decrease n and maintain A as a heap

---

procedure HEAP-EXTRACT-MAX(A)
  max $= A[1]$
  $A[1] = A[n]$
  $n = n - 1$
  MAX-HEAPIFY(A, 1)
  Return max

Question

Use BUILD-MAX-HEAP and HEAP-EXTRACT-MAX to design a sort algorithm?

HEAP-INCREASE-KEY

Task: After increasing $A[i]$, maintain A as a heap

HEAP-INCREASE-KEY

Task: After increasing $A[i]$, maintain A as a heap

---

procedure HEAP-INCREASE-KEY(A, i, key)
    $A[i] = key$
    while $i > 1$ and $A[parent(i)] < A[i]$ **do**
        Exchange $A[i]$ with $A[parent(i)]$
        $i = Parent(i)$

HEAP-INCREASE-KEY

Task: After increasing $A[i]$, maintain A as a heap

---

procedure HEAP-INCREASE-KEY(A, i, key)

  $A[i] = key$

  while $i > 1$ and $A[parent(i)] < A[i]$ **do**

    Exchange $A[i]$ with $A[parent(i)]$

    $i = Parent(i)$

Question

How to delete an arbitrary element?

MAX-HEAP-INSERT

Task: Insert a new element with value *key*

---

procedure HEAP-INCREASE-KEY(A, *key*)

  $n = n + 1$

  HEAP-INCREASE-KEY(A, n, *key*)

Summary of Heap

1. Heap is a priority queue that keeps the largest element as the root
2. An almost complete binary tree

Summary of Heap

1. Heap is a priority queue that keeps the largest element as the root
2. An almost complete binary tree
3. Easy to implement and works well in practice
4. Support many operations in $O(\log_2 n)$-time

Summary of Heap

1. Heap is a priority queue that keeps the largest element as the root
2. An almost complete binary tree
3. Easy to implement and works well in practice
4. Support many operations in $O(\log_2 n)$-time
5. While heap does not support SEARCH(KEY), one could combine it with Hash
6. But neither of them could find the kth largest in $o(n)$ time.

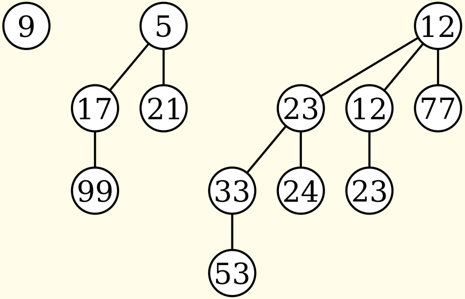
Extensions

Support UNION in $o(n)$ time

Binomial heap and Fibonacci Heap

Basic idea: Allow each node to have more children.

While they are faster and more powerful, complicated to implement.



Questions?