Disjoint Sets

CS 140 HMC - Jim Boerkoel

Today's Goals

- Admin
- Review of ADTs
- Disjoint Sets
- Red-Black Trees
  - Insertion / Deletion

Admin

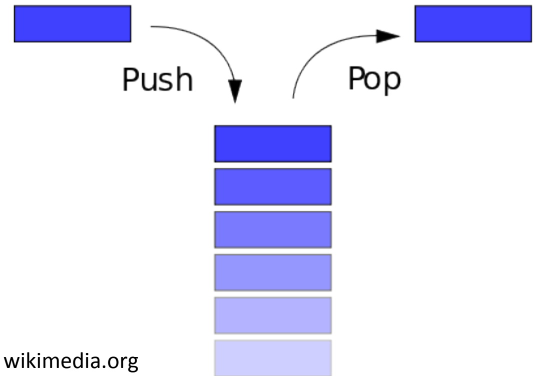
- Homework 5a out today
- Midterm out Thursday
  - 2 hours
  - Must take and turn in by Tuesday at 2:45pm
  - No homework / no office hours
  - Distributed at review session (see below)
  - In class session?
- Joint Pomona / Mudd Review Session on Thursday
  - Edmunds 101, 2:45 (right before colloquium)
  - Will cover topics / example problems suggested in homework submissions
  - Hand out exam
- Final
  - Take-home; time-shiftable
  - Details later…

Abstract Data Type (ADT)

- An abstract model of data object
- Characterize behavior by the operations that must be supported
- Examples:
  - Dynamic Sets (PS3A)
  - Stacks / Queues (PS3BQ2)
  - Priority Queues (PS3BQ1)
  - Mergeable Heaps (Last time)
  - Disjoint Sets (Today)
- Key idea: Implementation is independent
  - Examples: Dynamic set as (singly- / doubly-) (un)sorted linked list, priority queue implemented as binary max heap, mergeable heap as binary min heap, as binomial min heap, and as linked list

Stack ADT

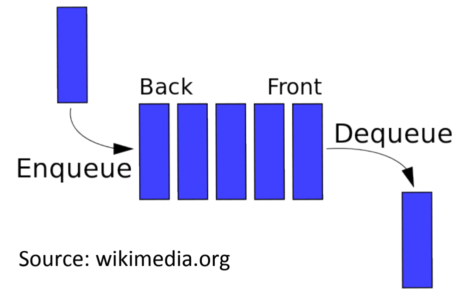
- Defined by operations:
  - Empty
  - Push
  - Pop / Peek
  - LIFO behavior



Source: wikimedia.org

Queue ADT

- Defined by operations:
  - Enqueue
  - Dequeue / Peek
  - FIFO behavior



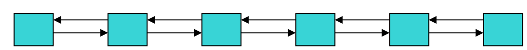
Source: wikimedia.org

Mergeable Heap

A mergeable heap is any data structure that supports the following five operations, in which each element has a key:

- **MakeHeap()** - creates and returns a new heap containing no elements.
- **BuildHeap()** - creates and returns a new heap from *n* elements.
- **Insert(H,x)** - inserts element x, whose key has already been filled in, into heap H.
- **Minimum(H)** - returns a pointer to the element in heap H whose key is minimum.
- **Extract-Min(H)** - deletes the element from heap H whose key is minimum, returning a pointer to the element.
- **Union(H_1, H_2)** - creates and returns a new heap that contains all the elements of heaps H1 and H2. Heaps H1 and H2 are "destroyed" by this operation.
- ALSO: Decrease-Element, Delete

Linked-list heap

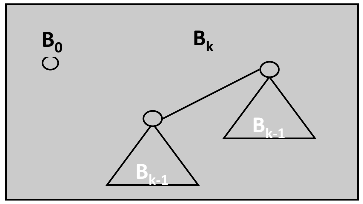


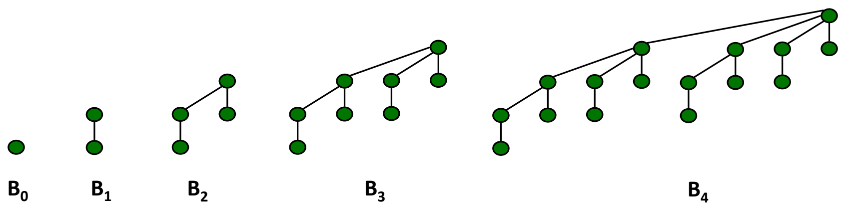
- Store the elements in a doubly linked list
- Insert: add to the end/beginning
- Max: search through the linked list
- Extract-Max: search and delete
- Increase: increase value
- Union: concatenate linked lists

Adapted from:
Kevin Wayne

Binomial Tree

B_k: a binomial tree B_{k-1} with the addition of a left child with another binomial tree B_{k-1}



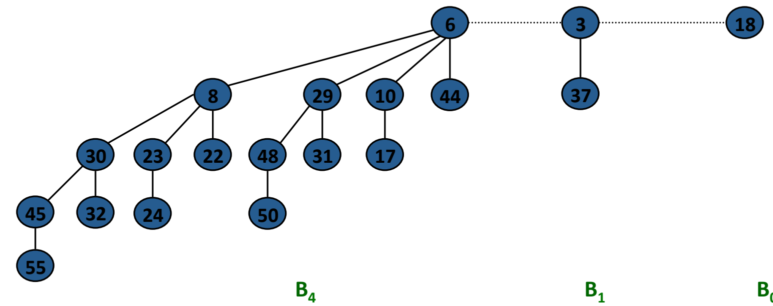


Binomial Heap

- Binomial heap Vuillemin, 1978.

Sequence of binomial trees that satisfy binomial heap property:

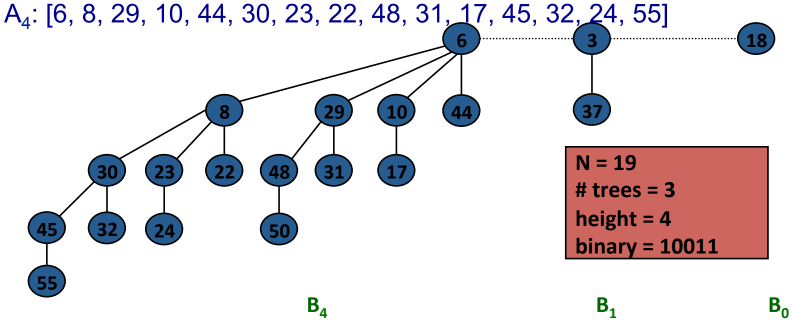
- each tree is min-heap ordered
- top level: full or empty binomial tree of order/rank k
- which are empty or full is based on the number of elements



Binomial Heap

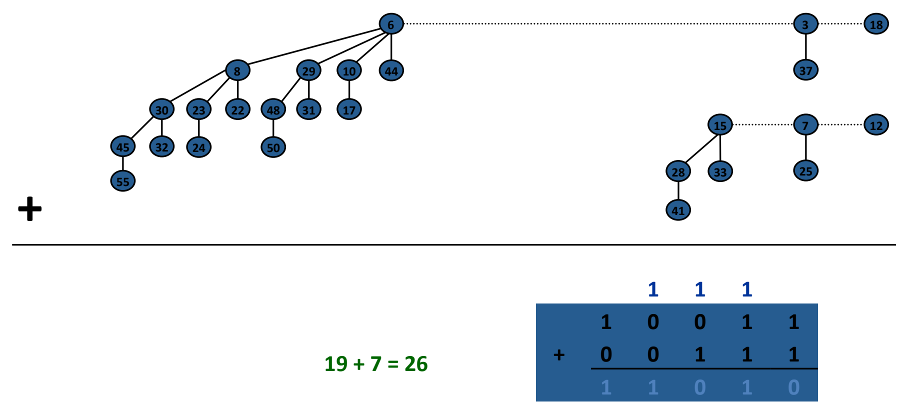
Represented as a series of arrays (each representing a min-heap)

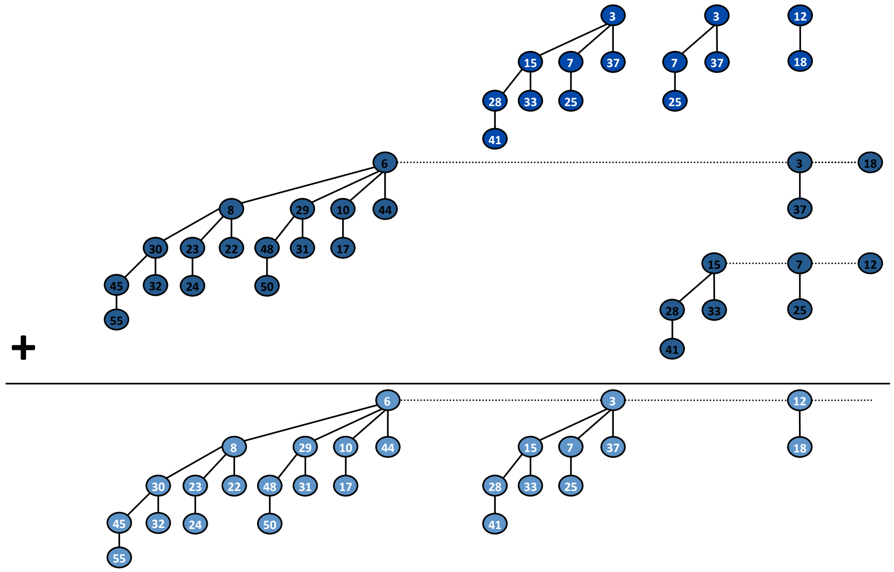
A_0: [18]
A_1: [3, 7]
A_2: empty
A_3: empty
A_4: [6, 8, 29, 10, 44, 30, 23, 22, 48, 31, 17, 45, 32, 24, 55]



Binomial Heap: Union

Go through each tree size starting at 0 and merge as we go





Heaps

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Linked-list
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$	Θ(n)
INSERT	$\Theta(\log n)$	$O(\log n)$	Θ(1)
MAXIMUM	$\Theta(1)$	$O(\log n)$	Θ(n)
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(\log n)$	Θ(n)
UNION	$\Theta(n)$	$\Theta(\log n)$	Θ(1)
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(\log n)$	Θ(1)
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	Θ(1)

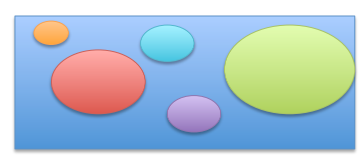
(adapted from Figure 19.1, pg. 456 [1])

Disjoint-Set Data Structure

- Also known as "union-find"
- **Idea:** Maintain collection $\Sigma = \{S_1, \ldots, S_K\}$ of disjoint (non-overlapping), dynamic (changing over time) sets.
- Each set is identified by a *representative*, which is some member of the set
  - Doesn't matter which member is representative as long as representative is consistent (doesn't change unless the set does).

The Disjoint Set ADT

- Operations:
  - Make-Set(x): make a new set $S_i = \{x\}$, and add S_i to Σ
  - Union(x,y): if $x \in S_x, y \in S_y$ then $\Sigma = \Sigma - S_x - S_y \cup \{S_x \cup S_y\}$
    - Representative of new set is any member of $S_x \cup S_y$
    - Destroys S_x and S_y
  - Find-Set(x): return representative of set containing *x*



Example sequence of operations

Operation	Σ
Make-Set(x)	{x}
Make-Set(y)	{x}, {y}
Make-Set(z)	{x}, {y},{z}
Union(x,y)	{x,y},{z}
Find-Set(x)	Returns: {x,y}
	(Representative underlined)

The Disjoint Set ADT

- Analysis in terms of:
  - n = # of elements = # of Make-Set operations
  - m = total # of operations
- Worksheet:
  1. m is <, ≤, =, ≥, or > n. Why?
  2. # of Union operations is ≥ ___________ Why?

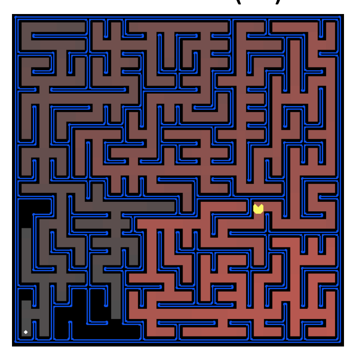
Example sequence of operations

Operation	Σ
Make-Set(x)	{x}
Make-Set(y)	{x}, {y}
Make-Set(z)	{x}, {y},{z}
Union(x,y)	{x,y},{z}
Find-Set(x)	Returns: {x,y}
	(Representative underlined)

For analysis, typically assume first *n* operations are Make-Set

Problem

I need help generating mazes for my Pacman projects in CS-151 (AI)



Maze-Builder

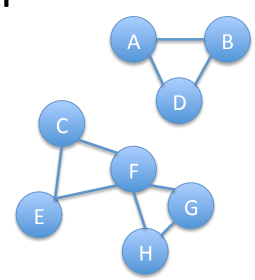
```
INPUT: a k by l grid of k*l = m squares, separated by walls
OUTPUT: a feasible maze

    for each of the m squares:
            Make-Set(square)

    let W be all the interior walls

    while |W| > 0
            pick a random wall in W and remove from W
            let x and y be the squares on either side of the wall

            if Find-Set(x) != Find-Set(y) // not in the same set
                remove that wall in the maze (connecting the two sets of squares)
                Union(set(x), set(y))

    select two random edge walls and remove them,
          label one as entrance and one as exit
```

Credit: Dave Kauchak

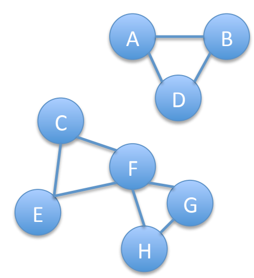
Maze-Demo

Applications: Detecting Disjoint Graphs

- A graph G = <*V*,*E*> is composed of
  - *V*: a set of vertices
  - *E*: a set of edges where edge {*u*,*v*} connects vertices *u* and *v*
- *Questions: Is it connected? Is there a path between vertex* x *and vertex* y?



Connected-Components



CONNECTED-COMPONENTS(G)
  for each vertex $v \in G.V$
      MAKE-SET(v)
  for each edge $(u, v) \in G.E$
      if FIND-SET$(u) \neq$ FIND-SET(v)
          UNION(u, v)

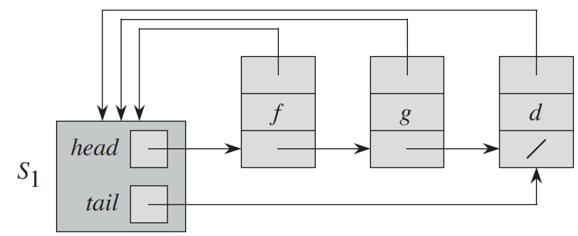
SAME-COMPONENT(u, v)
  if FIND-SET(u) == FIND-SET(v)
      return TRUE
  else return FALSE

Implementations of Disjoint Sets

- We will explore two implementations of disjoint set operations
  - Linked lists
  - Disjoint forests

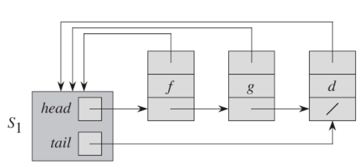
A linked list implementation

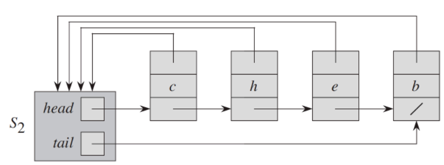
- Represent each set as a singly linked list with attributes:
  - Head: the first element in the list; representative
  - Tail: the last element in the list
- Each object in the list has attributes for:
  - Set member (the element itself)
  - Pointer to the set object (i.e., to the representative)
  - Next (the link)



Operations

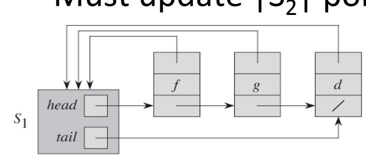
- Make-Set: create a singleton list
- Find-Set: follow pointer back to list object, then follow head pointer back to representative
- Union(x,y): append *y*'s list onto end of *x*'s list

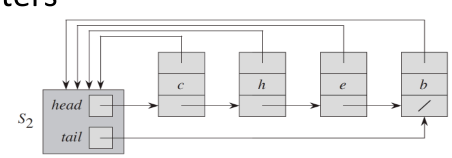


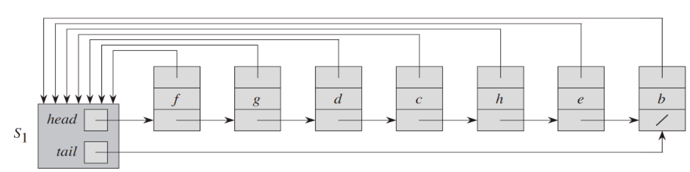


Linked-list Union

- How expensive is it?
- Must update $|S_2|$ pointers







Amortized Cost?

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n-1$
	$\Theta(n^2)$ total

Amortized time per operation $= \Theta(n)$.

Weighted-union heuristic

- Always append smaller list to larger list
- A single union of two sets containing n/2 members can still take Ω(n) time

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	~~2~~
UNION(x_4, x_3)	~~3~~
⋮	⋮
UNION(x_n, x_{n-1})	~~$n-1$~~

Weighted-union heuristic

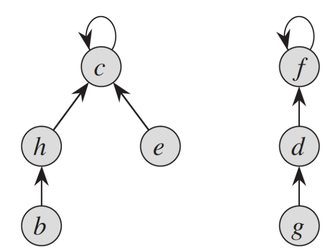
Theorem: With weighted union, a sequence of *m* operations on *n* elements takes *O*(*m*+*n* log *n*) time.

- How many times can a member's representative pointer be updated?

times updated	size of resulting set
	≥ 2

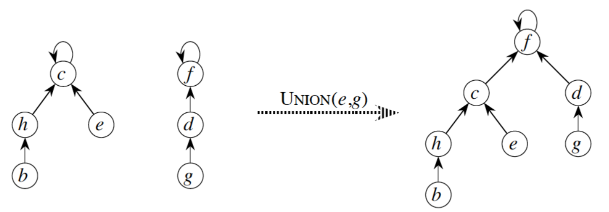
A Disjoint-set forest

- Forest of trees
  - 1 tree per set; root is representative
  - Each node points only to its parent



Disjoint-set Forest Operations

- Make-set: make a single-node tree *Cost?*
- Union: Make one root a child of the other *Cost?*



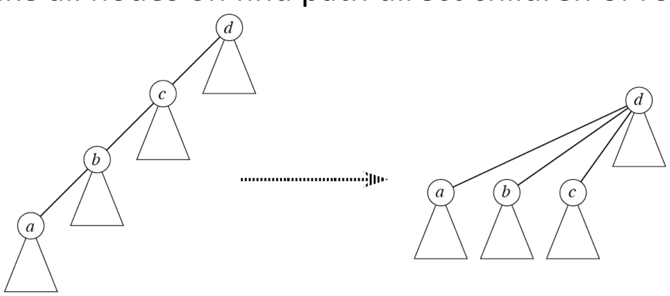
- Find-Set: follow pointers to root *Cost?*

Union-by-rank Heuristic

- Make the root of the smaller tree (fewer nodes) a child of the root of the larger tree
  - Don't actually use *size*
  - Use *rank*---upper-bound on the height of node
  - Rank maintained as an additional node attribute (with parent)
  - Make the root with the smaller rank into a child of the root with the larger rank
- Improves analysis to:
- Is it a tight bound? Show Ω in PS5A

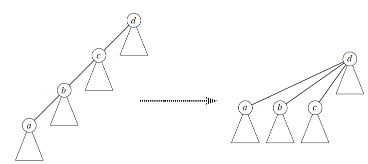
Path compression

- **Find-Path** = nodes visited during Find-Set on the trip to root
- Make all nodes on find path direct children of root



Implementation

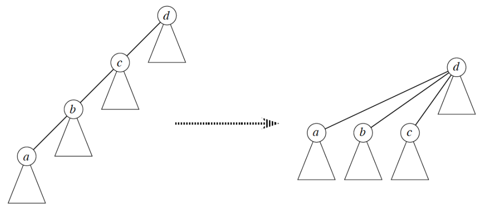
```
MAKE-SET(x)
  x.p = x
  x.rank = 0

UNION(x, y)
  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
  if x.rank > y.rank
      y.p = x
  else x.p = y
      // If equal ranks, choose y as parent and increment its rank.
      if x.rank == y.rank
          y.rank = y.rank + 1

FIND-SET(x)
  if x ≠ x.p
      x.p = FIND-SET(x.p)
  return x.p
```



Disjoint-set Forest Analysis

If use both union by rank and path compression, $O(m\,\alpha(n))$.

n	$\alpha(n)$
0–2	0
3	1
4–7	2
8–2047	3
2048–$A_4(1)$	4



$A_4(1) \gg 10^{80} \approx$ # of atoms