Graph Primitives

Dijkstra's Algorithm: The Basics

Design and Analysis of Algorithms I

Single-Source Shortest Paths

Input: directed graph G=(V, E). (m=|E|, n=|V|)

- each edge has non negative length l_e
- source vertex s

Output: for each $v \in V$, compute

L(v) := length of a shortest s-v path in G

Length of path
= sum of edge lengths

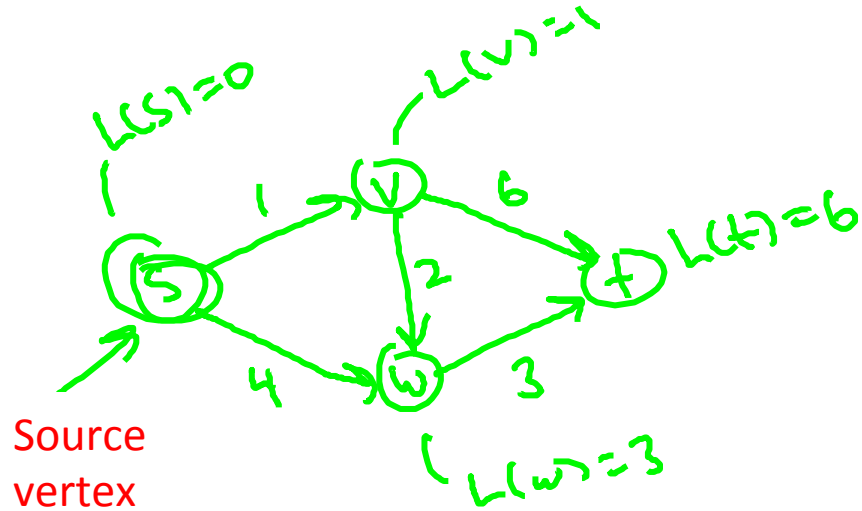
Path length = 6

Assumption:

1. [for convenience] $\forall v \in V, \exists s \Rightarrow v$ path
2. [important] $le \geq 0 \ \forall e \in E$

One of the following is the list of shortest-path distances for the nodes s,v,w,t, respectively. Which is it?

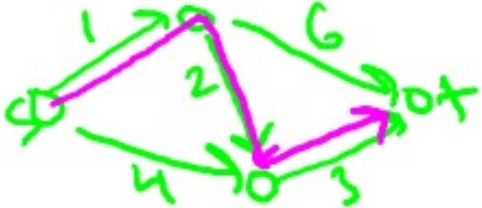
- 0,1,2,3
- 0,1,4,7
- 0,1,4,6
- 0,1,3,6



Why Another Shortest-Path Algorithm?

Question: doesn't BFS already compute shortest paths in linear time?

Answer: yes, IF $l_e = 1$ for every edge e.



Question: why not just replace each edge e by directed path of l_e unit length edges:

Answer: blows up graph too much

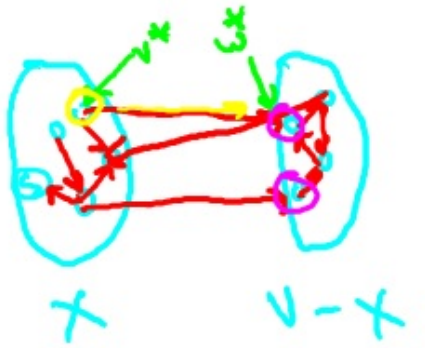
Solution: Dijkstra's shortest path algorithm.

Dijkstra's Algorithm

This array only to help explanation!

Initialize:
- X = [s] [vertices processed so far]
- A[s] = 0 [computed shortest path distances]
- B[s] = empty path [computed shortest paths]

Main Loop
- while X ‡ V:



-need to grow x by one node

Main Loop cont'd:
- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$

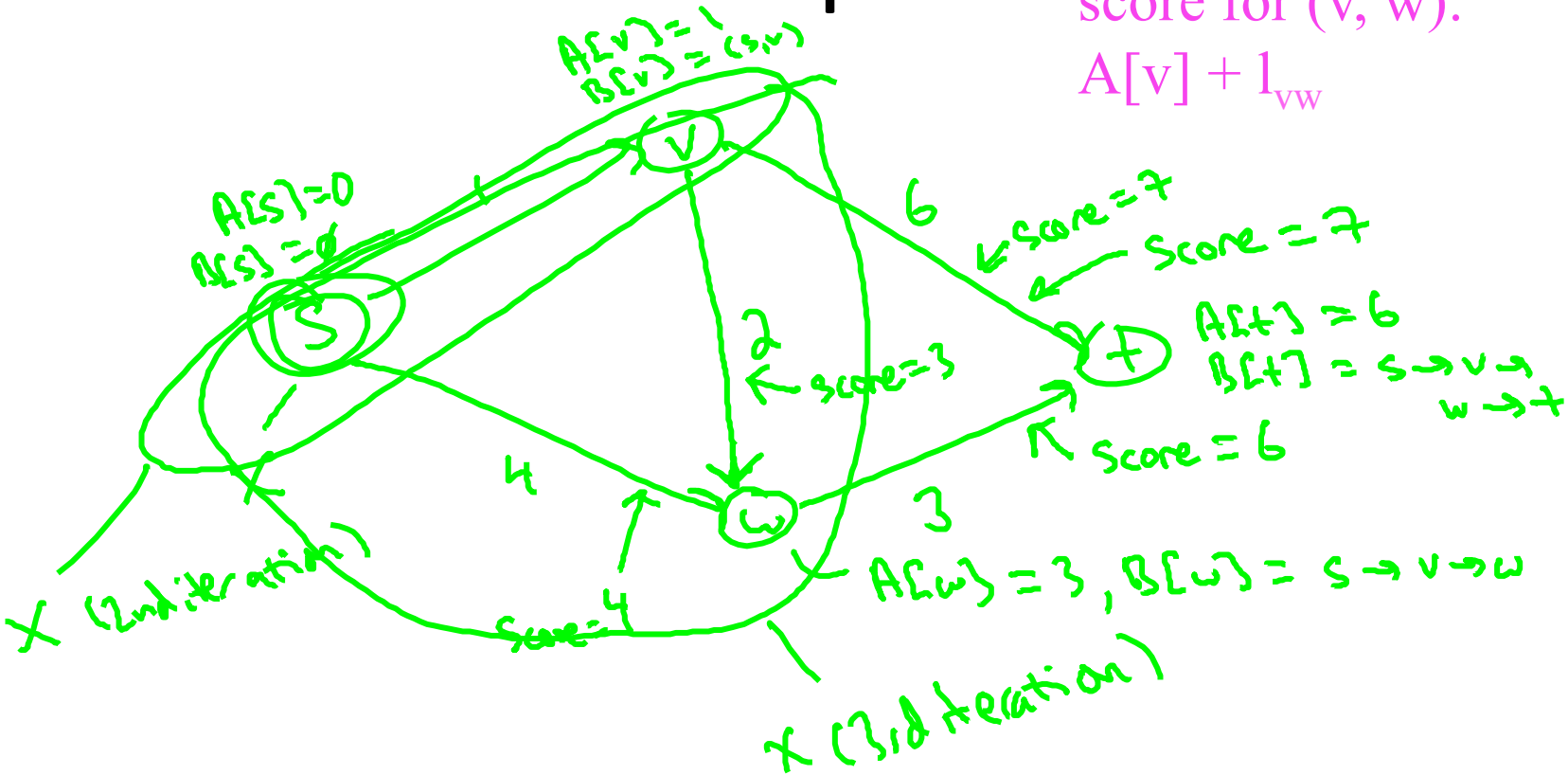
[call it (v*, w*)]

Already computed in earlier iteration

- add w* to X
- set $A[w^*] := A[v^*] + l_{v^*w^*}$
- set $B[w^*] := B[v^*]u(v^*, w^*)$

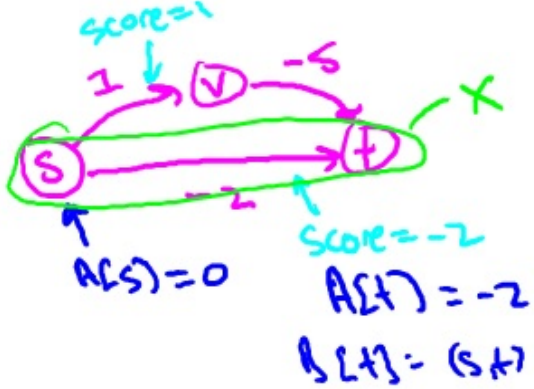
Example

Dijkstra's greedy score for (v, w): $A[v] + l_{vw}$



Non-Example

Question: why not reduce computing shortest paths with negative edge lengths to the same problem with non negative lengths? (by adding large constant to edge lengths)



Problem: doesn't preserve shortest paths !

Also: Dijkstra's algorithm incorrect on this graph !
(computes shortest s-t distance to be -2 rather than -4)

Graph Primitives

Dijkstra's Algorithm: The Basics

Design and Analysis of Algorithms I

Single-Source Shortest Paths

Input: directed graph G=(V, E). (m=|E|, n=|V|)

- each edge has non negative length l_e
- source vertex s

Output: for each $v \in V$, compute

L(v) := length of a shortest s-v path in G

Length of path
= sum of edge lengths

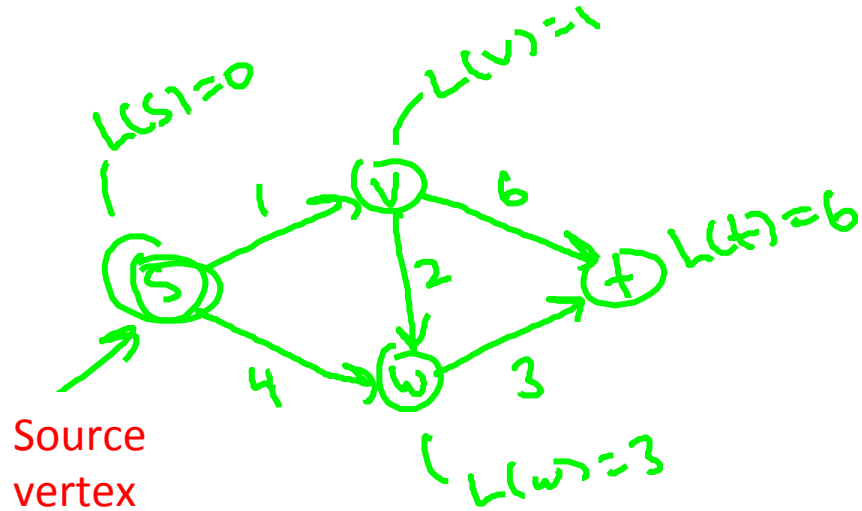
Path length = 6

Assumption:

1. [for convenience] $\forall v \in V, \exists s \Rightarrow v$ path
2. [important] $le \geq 0 \ \forall e \in E$

One of the following is the list of shortest-path distances for the nodes s,v,w,t, respectively. Which is it?

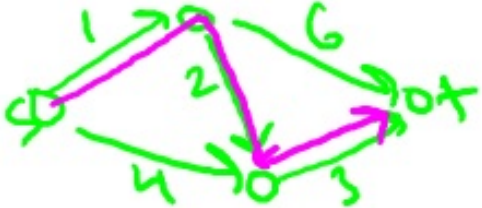
- 0,1,2,3
- 0,1,4,7
- 0,1,4,6
- 0,1,3,6



Why Another Shortest-Path Algorithm?

Question: doesn't BFS already compute shortest paths in linear time?

Answer: yes, IF $l_e = 1$ for every edge e.



Question: why not just replace each edge e by directed path of l_e unit length edges:

Answer: blows up graph too much

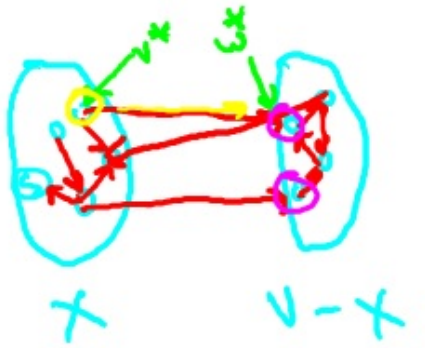
Solution: Dijkstra's shortest path algorithm.

Dijkstra's Algorithm

This array only to help explanation!

Initialize:
- X = [s] [vertices processed so far]
- A[s] = 0 [computed shortest path distances]
- B[s] = empty path [computed shortest paths]

Main Loop
- while X ‡ V:



-need to grow x by one node

Main Loop cont'd:
- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$

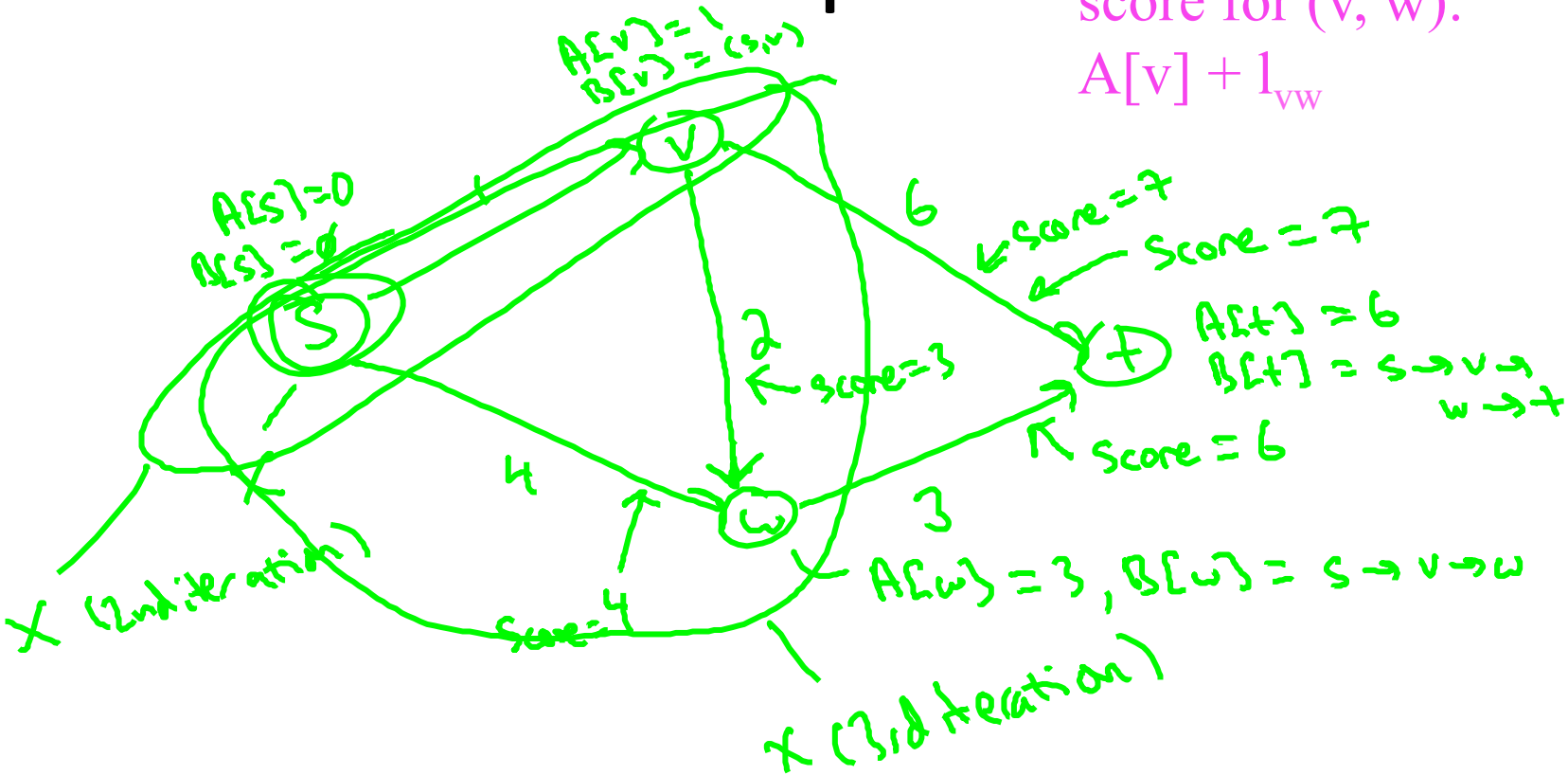
[call it (v*, w*)]

Already computed in earlier iteration

- add w* to X
- set $A[w^*] := A[v^*] + l_{v^*w^*}$
- set $B[w^*] := B[v^*]u(v^*, w^*)$

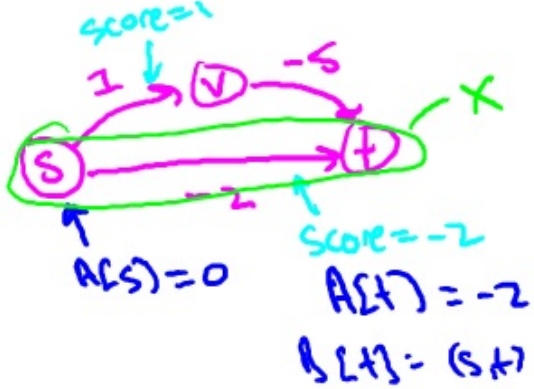
Example

Dijkstra's greedy score for (v, w): $A[v] + l_{vw}$



Non-Example

Question: why not reduce computing shortest paths with negative edge lengths to the same problem with non negative lengths? (by adding large constant to edge lengths)



Problem: doesn't preserve shortest paths !

Also: Dijkstra's algorithm incorrect on this graph !
(computes shortest s-t distance to be -2 rather than -4)

Graph Primitives

Dijkstra's Algorithm: Why It Works

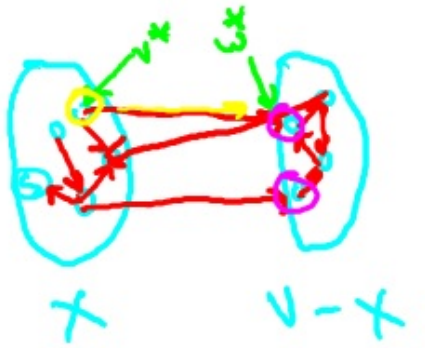
Design and Analysis of Algorithms I

Dijkstra's Algorithm

This array only to help explanation!

Initialize:
- X = [s] [vertices processed so far]
- A[s] = 0 [computed shortest path distances]
- B[s] = empty path [computed shortest paths]

Main Loop
- while X ‡ V:



-need to grow x by one node

Main Loop cont'd:
- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$

[call it (v*, w*)]

Already computed in earlier iteration

- add w* to X
- set $A[w^*] := A[v^*] + l_{v^*w^*}$
- set $B[w^*] := B[v^*]u(v^*, w^*)$

Correctness Claim

Theorem [Dijkstra] For every directed graph with nonnegative edge lengths, Dijkstra's algorithm correctly computes all shortest-path distances.

$$[i.e.,\ A[v] = L(v)\ \forall v \in V]$$

what algorithm computes

True shortest distance from s to v

Proof: by induction on the number of iterations.

Base Case: $A[s] = L[s] = 0$ (correct)

Proof

Inductive Step:

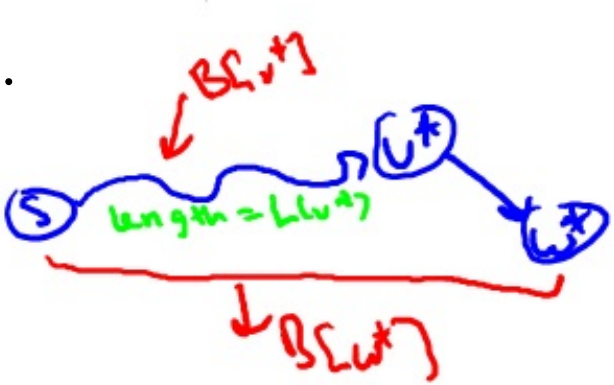
Inductive Hypothesis: all previous iterations correct (i.e., A[v] = L(v) and B[v] is a true shortest s-v path in G, for all v already in X).

In current iteration: (v* in X, w* not in X)

We pick an edge (v*, w*) and we add w* to X.

We set B[w*] = B[v*] u(v*, w*)

(B[v*] has length L(v*); B[w*] has length L(v*) + $l_{v^*w^*}$)



Also: $A[w^*] = A[v^*] + l_{v^*w^*} = L(v^*) + l_{v^*w^*}$ (L(v*) by I.H)

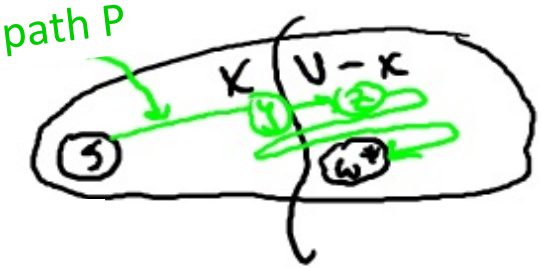
Proof (con'd)

Upshot: in current iteration, we set:

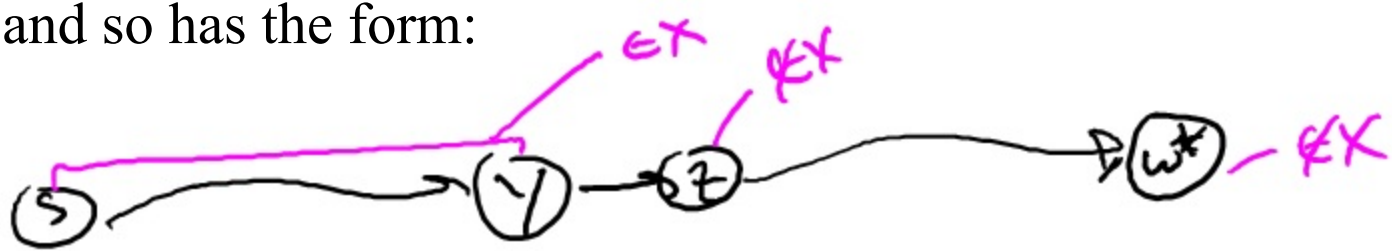
1. A[w*] = L(v*) + $l_{v^*w^*}$
2. B[w*] = an s -> w* path with length (L(v*) + $l_{v^*w^*}$)

To finish proof: need to show that *every* s-w* path has length >= L(v*) +$l_{v^*w^*}$ (if so, our path is the shortest!)

So: Let P= any s->w* path. Must "cross the frontier":

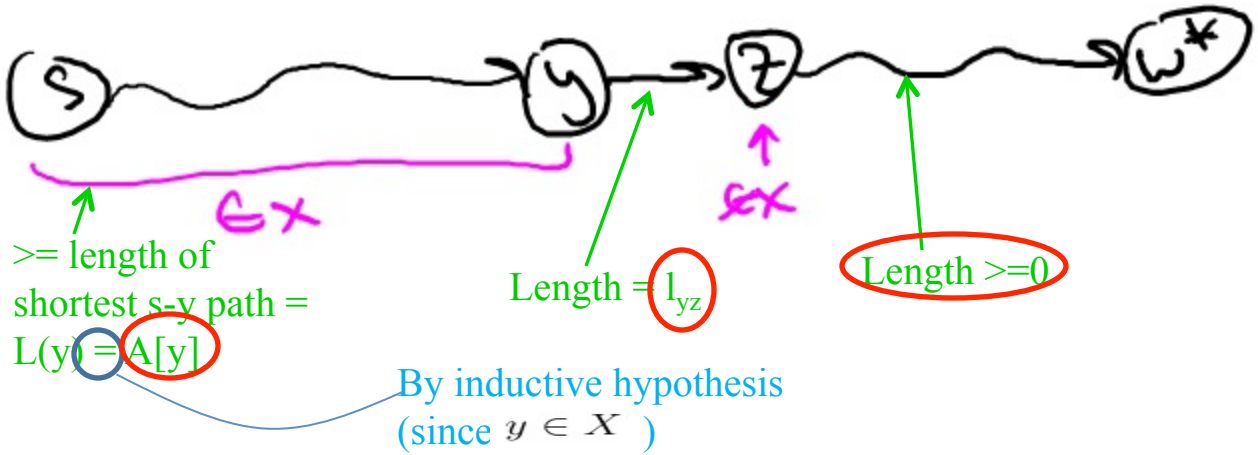


and so has the form:



Proof (con'd)

So: every s->w* path P has to have the form



(Since no negative edges !)

Total length of path P: at least A[y] + C_{yz}

-> by Dijkstra's greedy criterion, $A[v^*] + l_{v^*w^*} \leq A[y] + l_{yz} \leq$ length of P

length of our path !

($y \in X$, $z \notin X$)

Q.E.D.

Graph Primitives

Dijkstra's Algorithm: Fast Implementation

Design and Analysis of Algorithms I

Single-Source Shortest Paths

Input: directed graph G=(V, E). (m=|E|, n=|V|)

- each edge has non negative length l_e
- source vertex s

Length of path
= sum of edge lengths

Path length = 6

Output: for each $v \in V$,compute
L(v) := length of a shortest s-v path in G

Assumption:

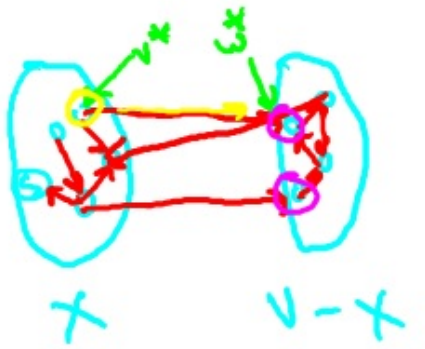
1. [for convenience] $\forall v \in V, \exists s \Rightarrow v$ path
2. [important] $le \geq 0 \ \forall e \in E$

Dijkstra's Algorithm

This array only to help explanation!

Initialize:
- X = [s] [vertices processed so far]
- A[s] = 0 [computed shortest path distances]
- ~~B[s] = empty path [computed shortest paths]~~

Main Loop
- while X≠V:



-need to grow x by one node

Main Loop cont'd:
- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$

[call it (v*, w*)]

Already computed in earlier iteration

- add w* to X
- set $A[w^*] := A[v^*] + l_{v^*w^*}$
- ~~set $B[w^*] := B[v^*]a(v^*, w^*)$~~

Which of the following running times seems to best describe a "naïve" implementation of Dijkstra's algorithm?

- ○ $\theta(m+n)$
- ○ $\theta(m \log n)$
- ○ $\theta(n^2)$
- ○ $\theta(mn)$

- (n-1) iterations of while loop
- $\theta(m)$ work per iteration

[$\theta(1)$ work per edge]

CAN WE DO BETTER?

Heap Operations

Recall: raison d'être of heap = perform Insert, Extract-Min in O(log n) time.
[rest of video assumes familiarity with heaps]

Height ~ $\log_2 n$

- conceptually, a perfectly balanced binary tree
- Heap property: at every node, key <= children's keys
- extract-min by swapping up last leaf, bubbling down
- insert via bubbling up

Also: will need ability to delete from middle of heap. (bubble up or down as needed)

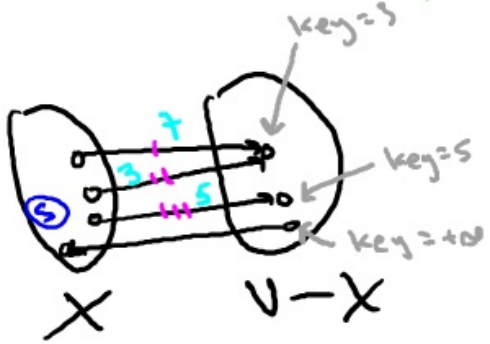
Two Invariants

Dijkstra's greedy score of (v, w) : $A[v] + l_{vw}$

Invariant # 1: elements in heap = vertices of V-X.

Invariant #2: for $v \notin X$ Key[v] = smallest Dijstra greedy score of an edge (u, v) in E with v in X

(of $+\infty$ if no such edges exist)

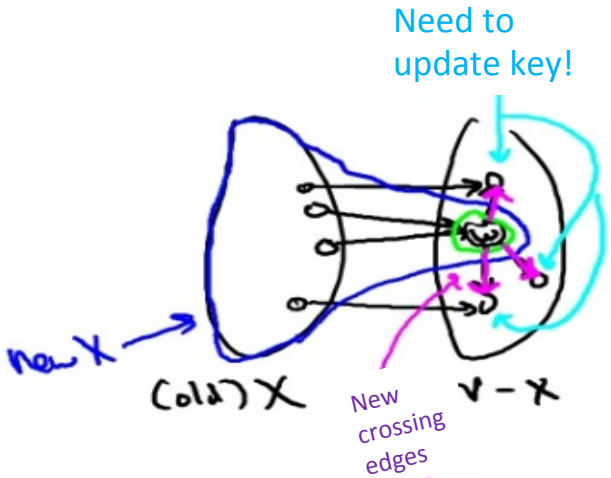


Point: by invariants, Extract-Min yields correct vertex w* to add to X next.

(and we set A[w*] to key[w*])

Maintaining the Invariants

To maintain Invariant #2: [i.e., that $\forall v \notin X$ Key[v] = smallest Dijkstra greedy score of edge (u,v) with u in X]



When w extracted from heap (i.e., added to X)

- for each edge (w,v) in E:
  - if v in V-X (i.e., in heap)
    - delete v from heap
    - recompute key[v] = min{key[v] , A[w] + l_{wv} }
    - re-Insert v into heap

Key update

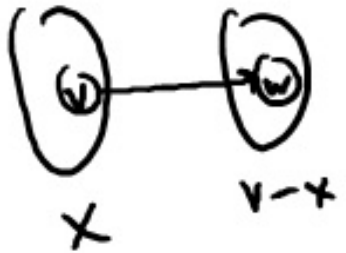
Greedy score of (w,v)

Running Time Analysis

You check: dominated by heap operations. (O(log(n)) each)

- (n-1) Extract mins
- each edge (v,w) triggers at most one Delete/Insert combo

(if v added to X first)



So: # of heap operations in O(n+m) = O(m)

So: running time = O(m log(n)) (like sorting)

Since graph is weakly connected

Data Structures

Introduction

Design and Analysis
of Algorithms I

Data Structures

Point : organize data so that it can be accessed quickly and usefully.

Examples : lists, stacks, queues, heaps, search trees, hashtables, bloom filters, union-find, etc.

Why so Many ? : different data structures support different sets of operations => suitable for different types of tasks.

Rule of Thumb : choose the “minimal” data structure that supports all the operations that you need.

Tim Roughgarden

Taking It To The Next Level

Level 0 - “what’s a data structure ?”

Level 1 - cocktail party-level literacy

Level 2 - “this problem calls out for a heap”

Level 3 - “I only use data structures that I create myself”

Tim Roughgarden

Data Structures

Heaps and Their Applications

Design and Analysis of Algorithms I

Heap: Supported Operations

- A container for objects that have keys
- Employer records, network edges, events, etc.

Insert: add a new object to a heap.

Running time : O(log(n))

Equally well, EXTRACT MAX

Extract-Min: remove an object in heap with a minimum key value. [ties broken arbitrarily]

Running time : O(log n) [n = # of objects in heap]

Also : HEAPIFY (n batched Inserts in O(n) time), DELETE(O(log(n)) time)

Application: Sorting

Canonical use of heap : fast way to do repeated minimum computations.

Example : SelectionSort ~$\theta(n)$ linear scans, $\theta(n^2)$ runtime on array of length n

Heap Sort : 1.) insert all n array elements into a heap
2.) Extract-Min to pluck out elements in sorted order

Running Time = 2n heap operations = O(nlog(n)) time.
=> optimal for a "comparison-based" sorting algorithm!

Application: Event Manager

"Priority Queue" – synonym for a heap.

Example : simulation (e.g., for a video game)

-Objects = event records [Action/update to occur at given time in the future]

- Key = time event scheduled to occur

- Extract-Min => yields the next scheduled event

Application: Median Maintenence

I give you : a sequence x1,…,xn of numbers, one-by-one.

You tell me : at each time step i, the median of {x1,….,xi}.

Constraint : use O(log(i)) time at each step i.

Solution : maintain heaps H_{Low} : supports Extract Max
H_{High} : supports Extract Min

Key Idea : maintain invariant that ~ i/2 smallest (largest) elements in H_{Low} (H_{High})

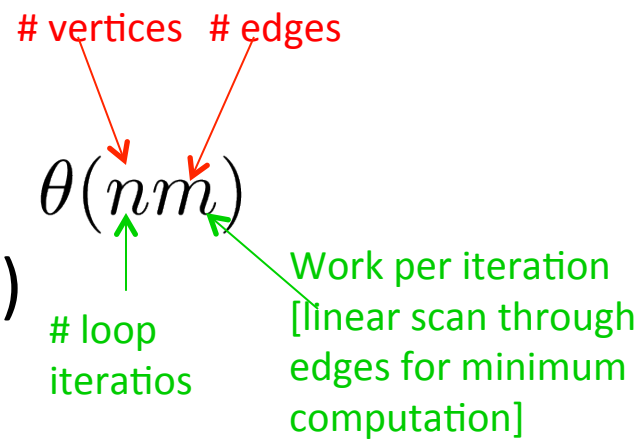
You Check : 1.) can maintain invariant with O(log(i)) work
2.) given invariant, can compute median in O(log(i)) work

Application: Speeding Up Dijkstra

Dijkstra's Shortest-Path Algorithm

-Naïve implementation => runtime = $\theta(nm)$

- with heaps => runtime = O(m log(n))



Data Structures

Heaps: Some Implementation Details

Design and Analysis of Algorithms I

Heap: Supported Operations

- A container for objects that have keys
- Employer records, network edges, events, etc.

Insert: add a new object to a heap.

Running time : O(log(n))

Equally well, EXTRACT MAX

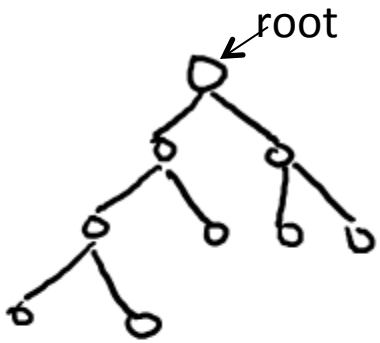
Extract-Min: remove an object in heap with a minimum key value. [ties broken arbitrarily]

Running time : O(log n) [n = # of objects in heap]

Also : HEAPIFY (n batched Inserts in O(n) time), DELETE(O(log(n)) time)

The Heap Property

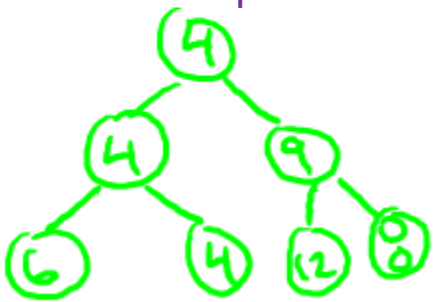
Conceptually : think of a heap as a tree.
-rooted, binary, as complete as possible



Heap Property: at every node x,
Key[x] <= all keys of x's children

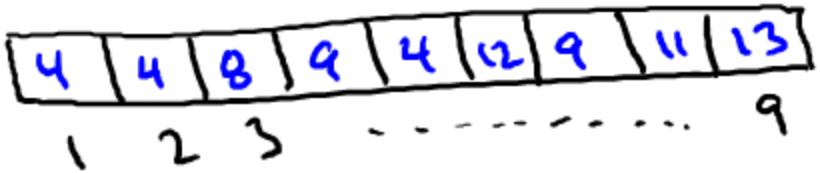
alternatively

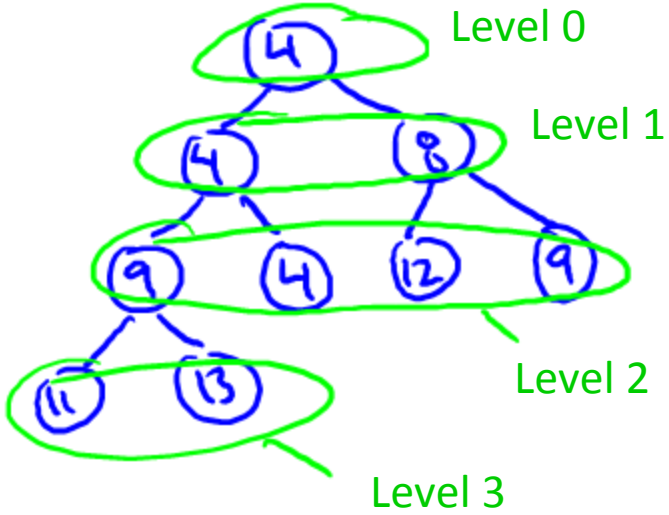
A heap



Consequence : object at root must have minimum key value

Array Implementation





Note : parent(i) = i/2 if i even
= [i/2] if i odd

i.e., round down

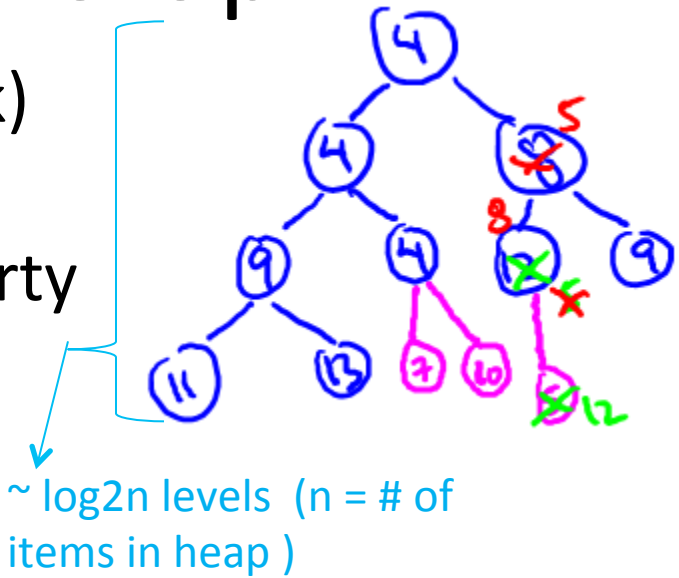
and children of i are 2i, 2i+1

Insert and Bubble-Up

Implementation of Insert (given key k)

Step 1: stick k at end of last level.

Step 2 : Bubble-Up k until heap property is restored (i.e., key of k's parent Is <= k)



Check : 1.) bubbling up process must stop, with heap property restored

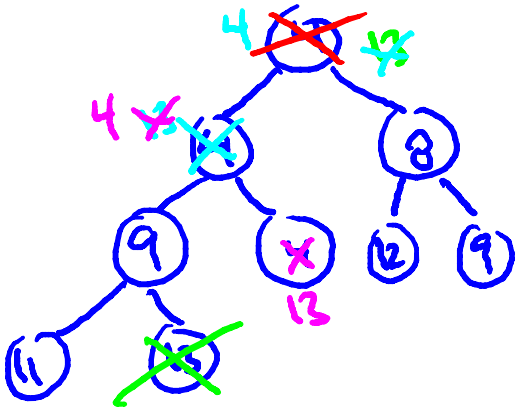
2.) runtime = O(log(n))

Extract-Min and Bubble-Down

Implementation of Extract-Min

1. Delete root
2. Move last leaf to be new root.
3. Iteratively Bubble-Down until heap property has been restored

   [always swap with smaller child!]



Check : 1.) only Bubble-Down once per level, halt with a heap

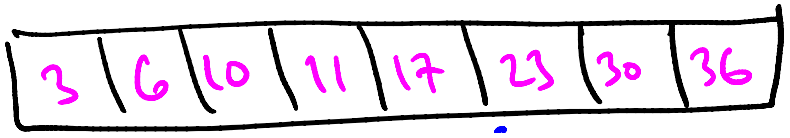
2.) run time = O(log(n))

Design and Analysis of Algorithms I

Data Structures

Balanced Search Trees: Supported Operations

Sorted Arrays: Supported Operations



BUT WHAT ABOUT
INSERTIONS + DELETIONS ?
(would take θ(n) time)

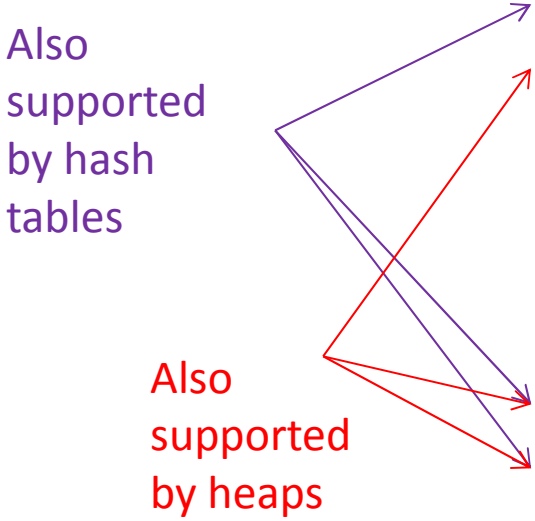
OPERATIONS	RUNNING TIME
SEARCH	θ(log(n))
SELECT (given order statistic I)	O(1)
MIN/MAX	O(1)
PRED/SUCC (given pointer to a key)	O(1)
RANK (i.e., # of keys less than or equal to a given value)	O(log(n))
OUTPUT IN SORTED ORDER	O(n)

Tim Roughgarden

Balanced Search Trees: Supported Operations

Raison d'etre : like sorted array + fast (logarithmic) inserts + deletes !

OPERATIONS	RUNNING TIME
SEARCH	θ(log(n))
SELECT	O(log(n))
MIN/MAX	O(log(n))
PRED/SUCC	O(log(n))
RANK	O(log(n))
OUTPUT IN SORTED ORDER	O(n)
INSERT	O(log(n))
DELETE	O(log(n))



Tim Roughgarden

Design and Analysis
of Algorithms I

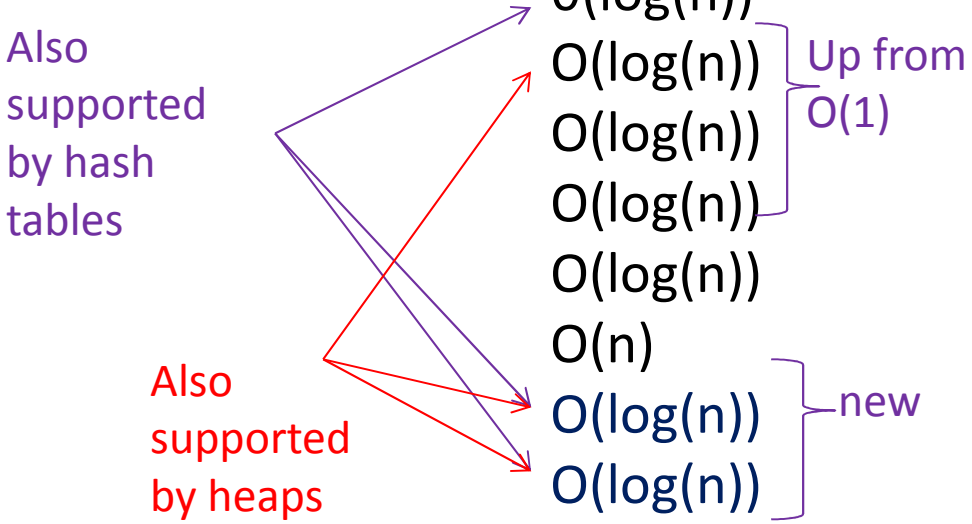
Data Structures

Binary Search Tree Basics

Balanced Search Trees: Supported Operations

Raison d'etre : like sorted array + fast (logarithmic) inserts + deletes !

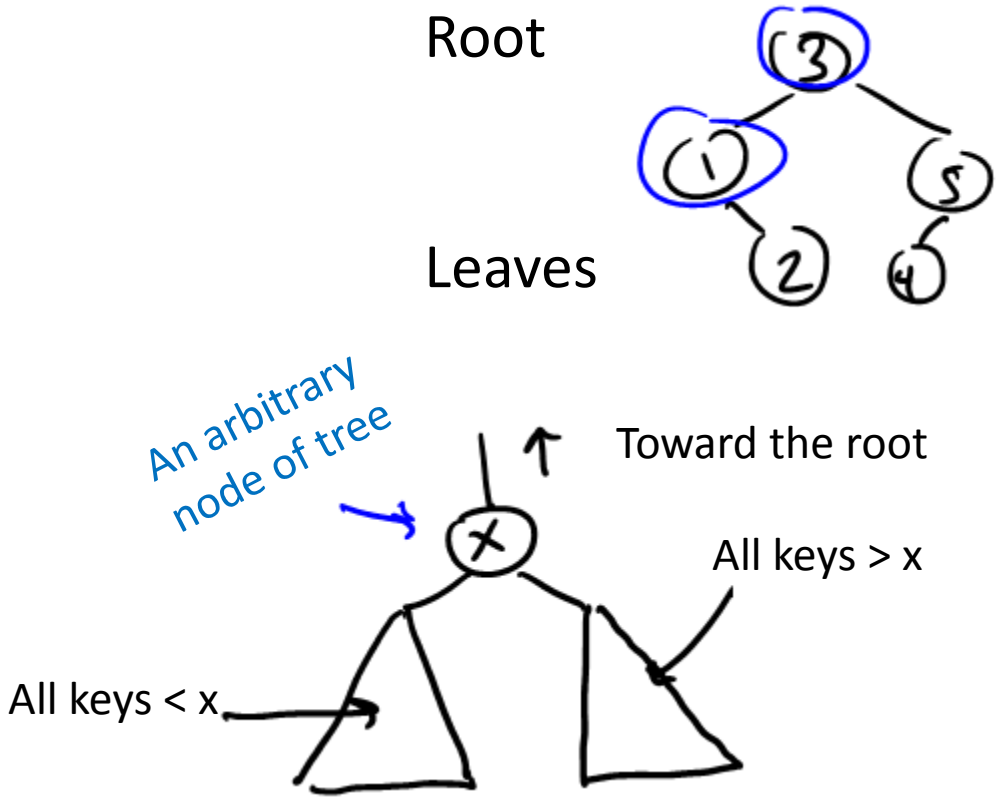
OPERATIONS	RUNNING TIME
SEARCH	θ(log(n))
SELECT	O(log(n))
MIN/MAX	O(log(n))
PRED/SUCC	O(log(n))
RANK	O(log(n))
OUTPUT IN SORTED ORDER	O(n)
INSERT	O(log(n))
DELETE	O(log(n))



Binary Search Tree Structure

-- exactly one node per key
-- most basic version :
each node has
-- left child pointer
-- right child pointer
-- parent pointer

SEARCH TREE PROPERTY :
(should hold at every node of the search tree)



Tim Roughgarden

The Height of a BST

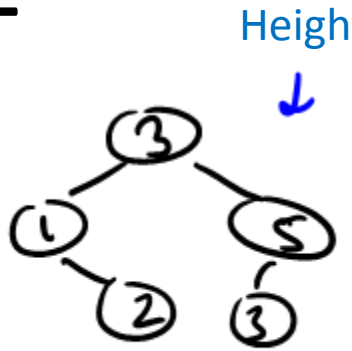
Note : many possible trees for a set of keys.

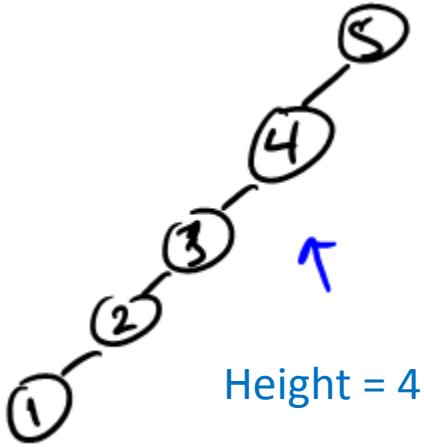
(aka depth) longest root-leaf path

Note : height could be anywhere from $\sim \log_2 n$ to $\sim n$

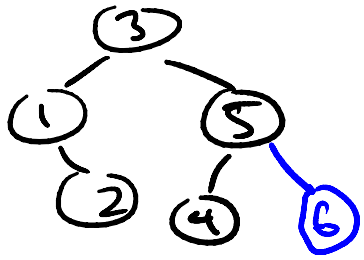
Worst case, a chain

Best case, perfectly balanced





Searching and Inserting



To Search for key k in tree T

-- start at the root

-- traverse left / right child pointers as needed

If k < key at current node

If k > key at current node

-- return node with key k or NULL, as appropriate

To Insert a new key k into a tree T

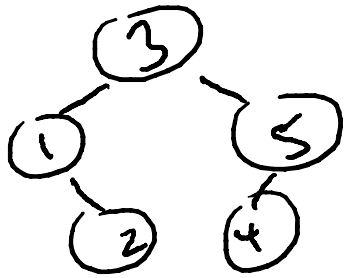
Exercise : preserves search tree property!

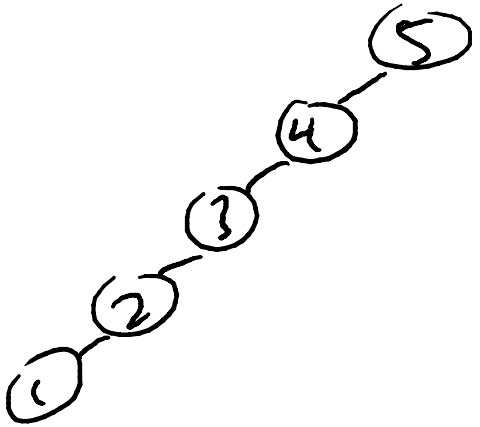
-- search for k (unsuccessfully)

-- rewire final NULL ptr to point to new node with key k

Tim Roughgarden

The worst-case running time of Search (or Insert) operation in a binary search tree containing n keys is…?



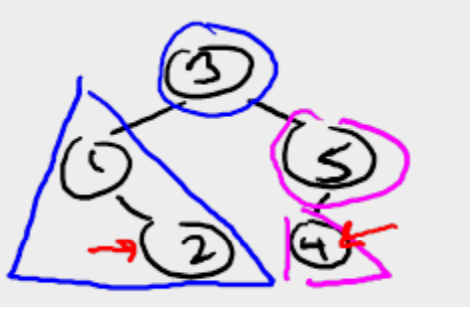


- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(height)$
- $\theta(n)$

Min, Max, Pred, And Succ

To compute the minimum (maximum) key of a tree

- Start at root
- Follow left child pointers (right ptrs, for maximum) untill you cant anymore (return last key found)



To compute the predecessor of key k

- Easy case : If k's left subtree nonempty, return max key in left subtree
- Otherwise : follow parent pointers until you get to a key less than k.

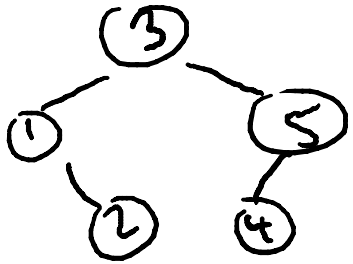
Happens first time you "turn left"

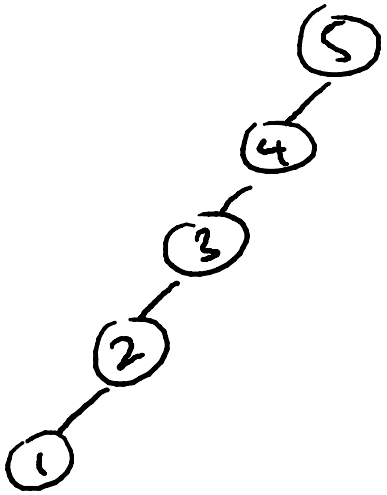
Exercise : prove this works

Tim Roughgarden

The worst-case running time of the Max operation in a binary search tree containing n keys is…?

- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(height)$
- $\theta(n)$





In-Order Traversal

TO PRINT OUT KEYS IN INCREASING ORDER

-Let r = root of search tree, with subtrees TL and TR

- recurse on TL

[by recursion (induction) prints out keys of TL in increasing order]

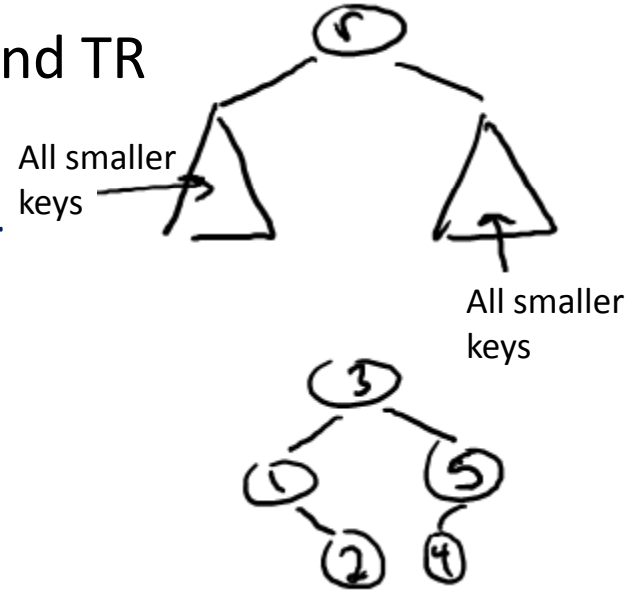
-Print out r's key

-Recurse on TR

[prints out keys of TR in increasing order]

RUNNING TIME

O(1) time, n recursive calls => O(n) total



Deletion

TO DELETE A KEY K FROM A SEARCH TREE

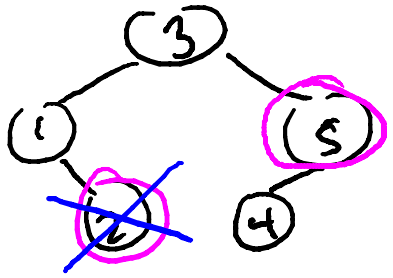
- SEARCH for k

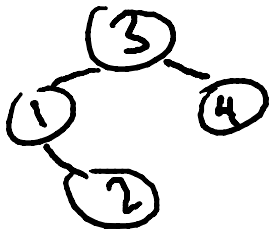
EASY CASE (k's node has no children)

-Just delete k's node from tree, done

MEDIUM CASE (k's node has one child)

(unique child assumes position
previously held by k's node)





Deletion (con'd)

DIFFICULT CASE (k's node has 2 children)

-Compute k's predecessor l
  [i.e., traverse k's (non-NULL) left child ptr, then
    right child ptrs until no longer possible]
- SWAP k and l !

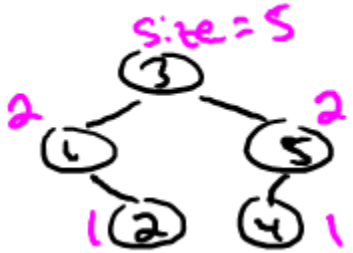
NOTE : in it's new position, k has no right child !
 => easy to delete or splice out k's new node

Exercise : at end, have a valid search tree !

RUNNING TIME : θ(height)

Select and Rank

Idea : store a little bit of extra info at each tree node about the tree itself (i.e., not about the data)



Example Augmentation : size(x) = # of tree nodes in subtree rooted at x.

Note : if x has children y and z,

then size(y) + size(z) + 1

Population in left subtree

Right subtree

x itself

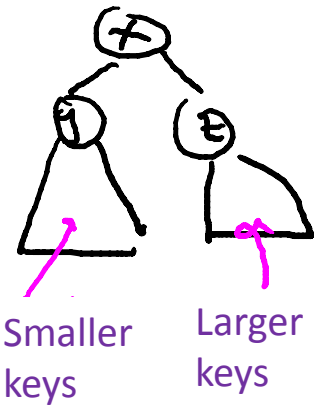
Also : easy to keep sizes up-to-date during an Insertion or Deletion (you check!)

Tim Roughgarden

Select and Rank (con'd)

HOW TO SELECT Ith ORDER STATISTIC FROM AUGMENTED SEARCH TREE (with subtree sizes)

- start at root x, with children y and z
- let a = size(y) [a = 0 if x has no left child]
- if a = i-1, return x's key
- if a >= I, recursively compute ith order statistic of search tree rooted at y
- if a < i-1 recursively compute (i-a-1)th order statistic of search tree rooted at z



RUNNING TIME = θ(height).

[EXERCISE : how to implement RANK ?

Design and Analysis
of Algorithms I

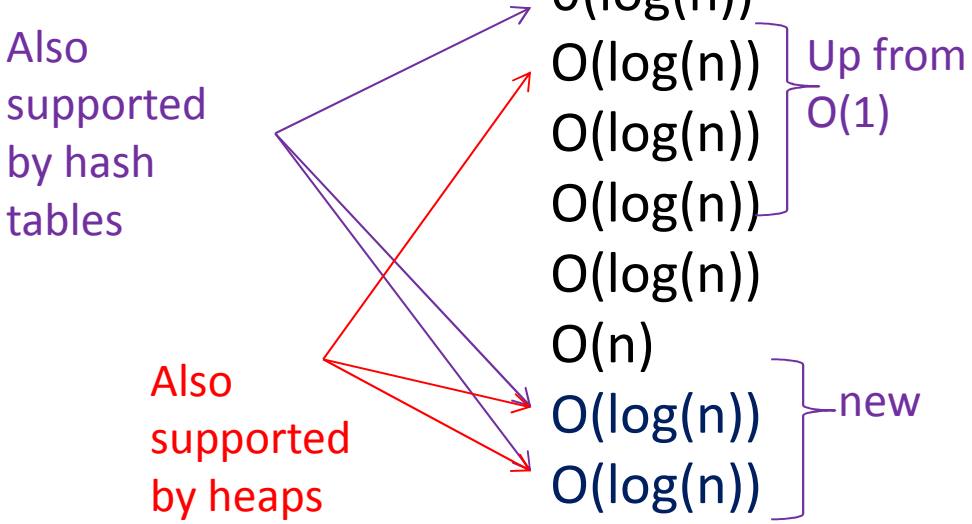
Data Structures

Binary Search Tree Basics

Balanced Search Trees: Supported Operations

Raison d'etre : like sorted array + fast (logarithmic) inserts + deletes !

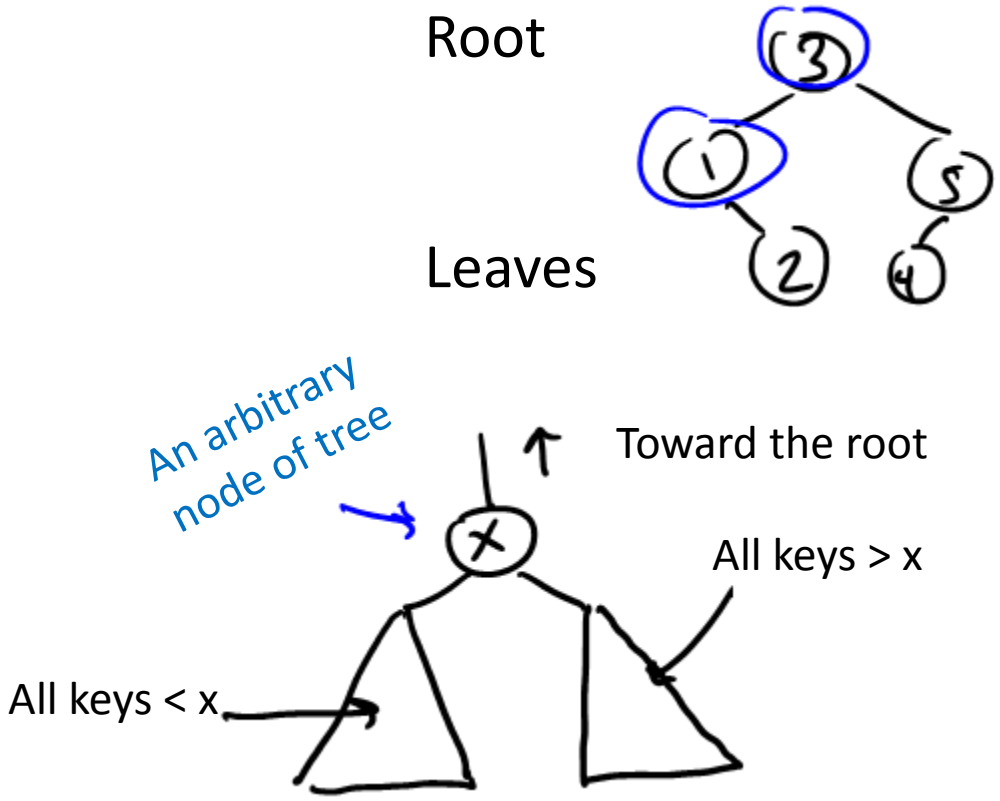
OPERATIONS	RUNNING TIME
SEARCH	θ(log(n))
SELECT	O(log(n))
MIN/MAX	O(log(n))
PRED/SUCC	O(log(n))
RANK	O(log(n))
OUTPUT IN SORTED ORDER	O(n)
INSERT	O(log(n))
DELETE	O(log(n))



Binary Search Tree Structure

-- exactly one node per key
-- most basic version :
each node has
-- left child pointer
-- right child pointer
-- parent pointer

SEARCH TREE PROPERTY :
(should hold at every node of the search tree)



Tim Roughgarden

The Height of a BST

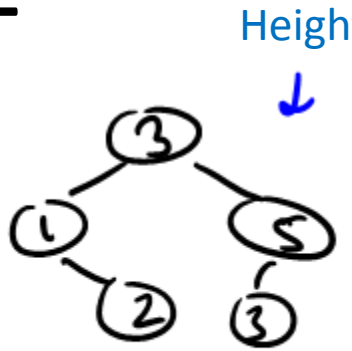
Note : many possible trees for a set of keys.

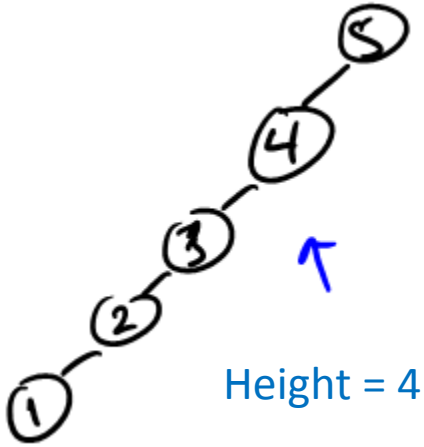
(aka depth) longest root-leaf path

Note : height could be anywhere from $\sim \log_2 n$ to $\sim n$

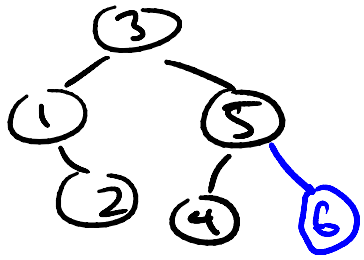
Worst case, a chain

Best case, perfectly balanced





Searching and Inserting



To Search for key k in tree T

-- start at the root

-- traverse left / right child pointers as needed

If k < key at current node

If k > key at current node

-- return node with key k or NULL, as appropriate

To Insert a new key k into a tree T

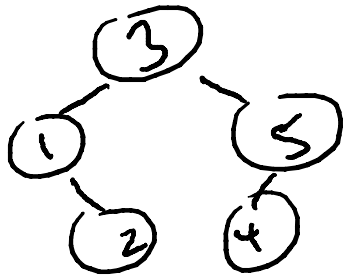
Exercise : preserves search tree property!

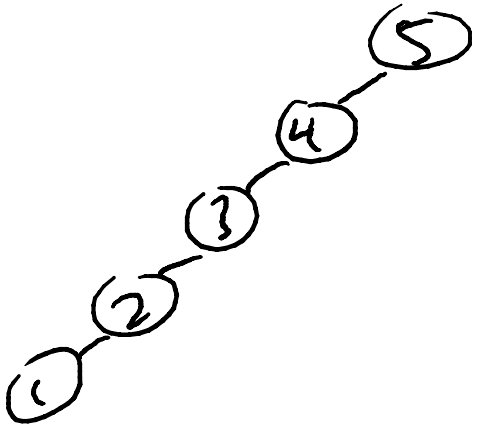
-- search for k (unsuccessfully)

-- rewire final NULL ptr to point to new node with key k

Tim Roughgarden

The worst-case running time of Search (or Insert) operation in a binary search tree containing n keys is…?



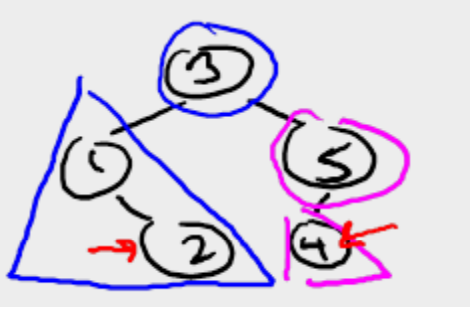


- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(height)$
- $\theta(n)$

Min, Max, Pred, And Succ

To compute the minimum (maximum) key of a tree

- Start at root
- Follow left child pointers (right ptrs, for maximum) untill you cant anymore (return last key found)



To compute the predecessor of key k

- Easy case : If k's left subtree nonempty, return max key in left subtree
- Otherwise : follow parent pointers until you get to a key less than k.

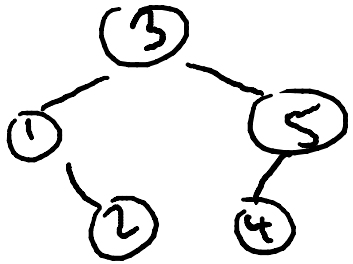
Happens first time you "turn left"

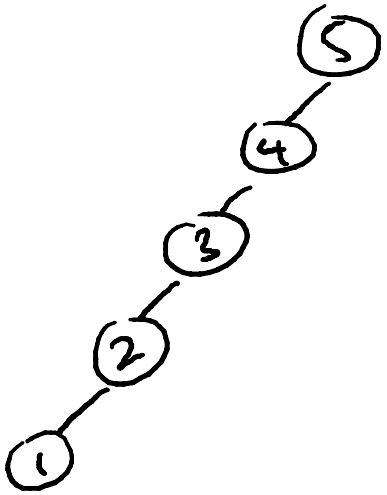
Exercise : prove this works

Tim Roughgarden

The worst-case running time of the Max operation in a binary search tree containing n keys is…?

- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(height)$
- $\theta(n)$





In-Order Traversal

TO PRINT OUT KEYS IN INCREASING ORDER

-Let r = root of search tree, with subtrees TL and TR

- recurse on TL

[by recursion (induction) prints out keys of TL in increasing order]

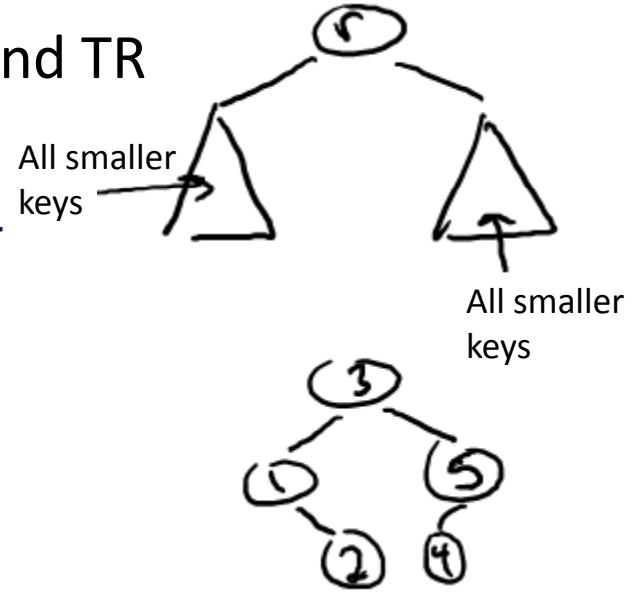
-Print out r's key

-Recurse on TR

[prints out keys of TR in increasing order]

RUNNING TIME

O(1) time, n recursive calls => O(n) total



Deletion

TO DELETE A KEY K FROM A SEARCH TREE

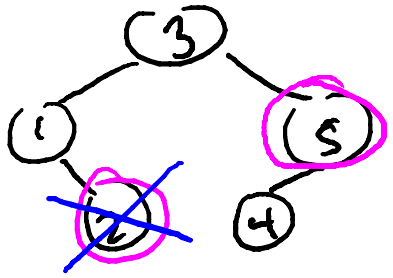
- SEARCH for k

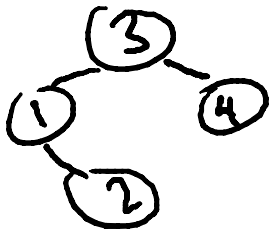
EASY CASE (k's node has no children)

-Just delete k's node from tree, done

MEDIUM CASE (k's node has one child)

(unique child assumes position
previously held by k's node)





Deletion (con'd)

DIFFICULT CASE (k's node has 2 children)

-Compute k's predecessor l
  [i.e., traverse k's (non-NULL) left child ptr, then
    right child ptrs until no longer possible]
- SWAP k and l !

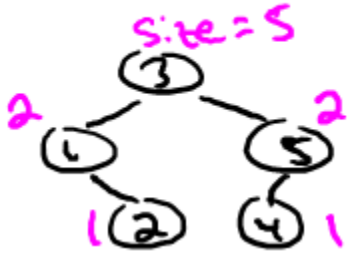
NOTE : in it's new position, k has no right child !
 => easy to delete or splice out k's new node

Exercise : at end, have a valid search tree !

RUNNING TIME : θ(height)

Select and Rank

Idea : store a little bit of extra info at each tree node about the tree itself (i.e., not about the data)



Example Augmentation : size(x) = # of tree nodes in subtree rooted at x.

Note : if x has children y and z,

then size(y) + size(z) + 1

Population in left subtree

Right subtree

x itself

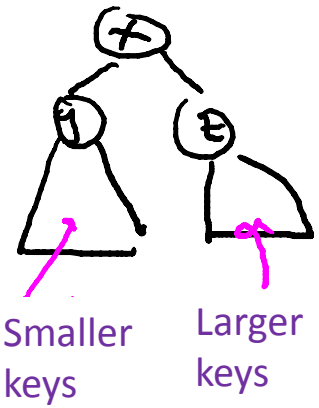
Also : easy to keep sizes up-to-date during an Insertion or Deletion (you check!)

Tim Roughgarden

Select and Rank (con'd)

HOW TO SELECT Ith ORDER STATISTIC FROM AUGMENTED SEARCH TREE (with subtree sizes)

- start at root x, with children y and z
- let a = size(y) [a = 0 if x has no left child]
- if a = i-1, return x's key
- if a >= I, recursively compute ith order statistic of search tree rooted at y
- if a < i-1 recursively compute (i-a-1)th order statistic of search tree rooted at z



RUNNING TIME = θ(height).

[EXERCISE : how to implement RANK ?

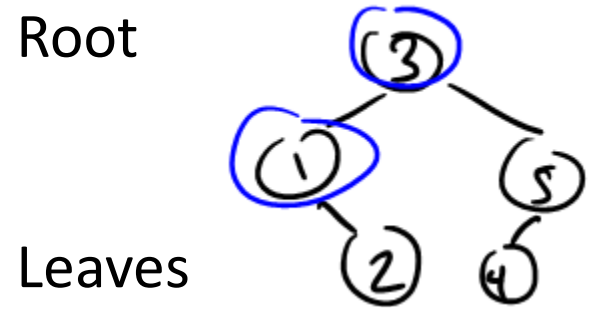
Data Structures

Red-Black Trees

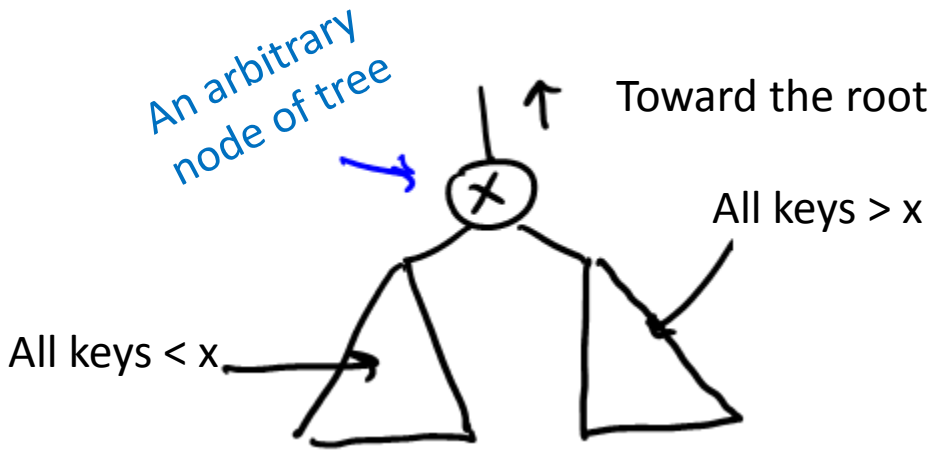
Design and Analysis
of Algorithms I

Binary Search Tree Structure

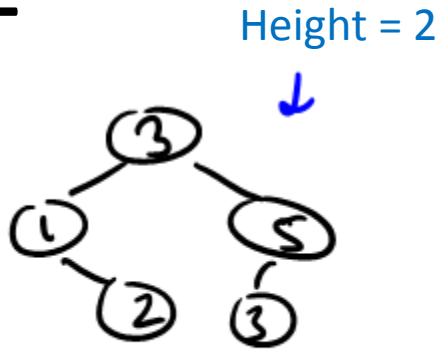
-- exactly one node per key
-- most basic version :
each node has
-- left child pointer
-- right child pointer
-- parent pointer



SEARCH TREE PROPERTY :
(should hold at every node of the search tree)



The Height of a BST



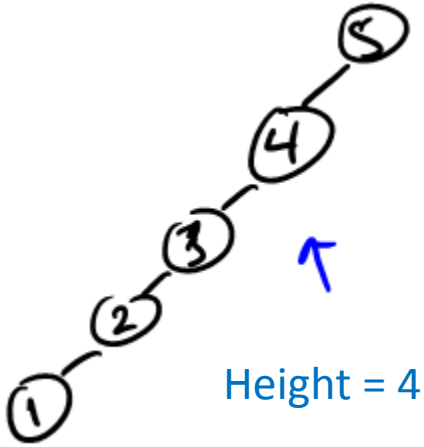
Note : many possible trees for a set of keys.

(aka depth) longest root-leaf path

Note : height could be anywhere from $\sim \log_2 n$ to $\sim n$

Worst case, a chain

Best case, perfectly balanced



Balanced Search Trees

Idea : ensure that height is always O(log(n)) [best possible]
$\Rightarrow$Search / Insert / Delete / Min / Max / Pred / Succ will then run in O(log(n)) time [n = # of keys in tree]

Example : red-black trees [Bayes '72, Guibas-Sedgewick '78]

[see also AUL trees, splay trees, B trees]

Red-Black Invariants

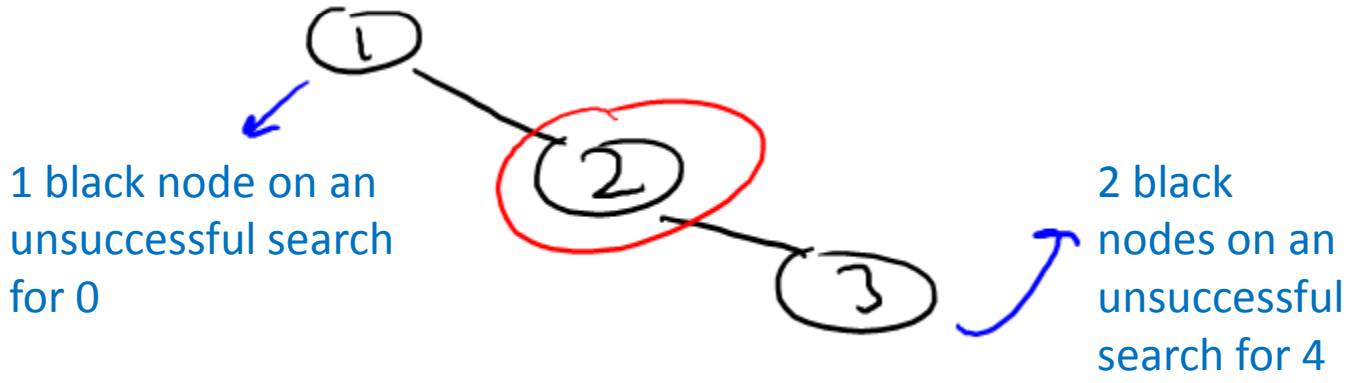
1. Each node red or black
2. Root is black
3. No 2 reds in a row
   [red node => only black children]
4. Every root-NULL path has same number of black nodes

Like in an
unsuccessful search

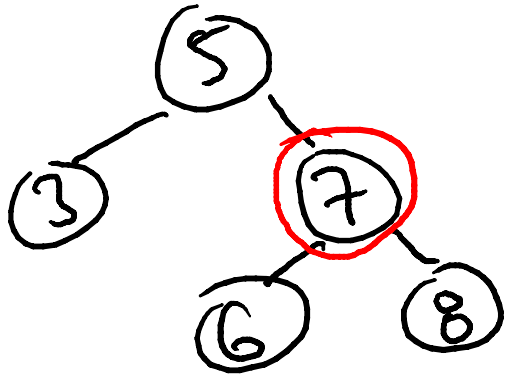
Example #1

Claim : a chain of length 3 cannot be a red-black tree

Proof



Example #2

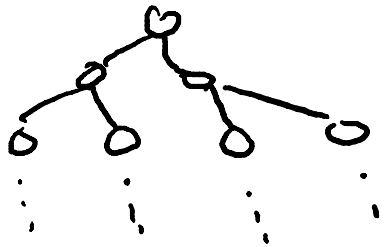


Height Guarantee

Claim : every red-black tree with n nodes has height $\leq 2\log_2(n+1)$

Proof : Observation : if every root-NUL path has >= k nodes, then tree includes (at the top) a perfectly balanced search tree of depth k-1.

=> Size n of the tree must Be at least $2^k - 1$



[k = 3]

Height Guarantee (con'd)

Story so far : size $n \geq 2^k - 1$, where k = minimum # of nodes on root – NULL path

=> $k \leq log_2(n+1)$

Thus : in a red-black tree with n nodes, there is a root-NULL path with at most $\log_2(n+1)$ black nodes.

By 4th Invariant : every root-NULL path has $\leq \log_2(n+1)$ black nodes

By 3rd Invariant : every root-NULL path has $\leq 2\log_2(n+1)$ total nodes.

Q.E.D.

Which of the search tree operations have to be re-implemented so that the Red-Black invariants are maintained?

- Search
- Delete
- Insert and Delete
- None of the above

Data Structures

Rotations

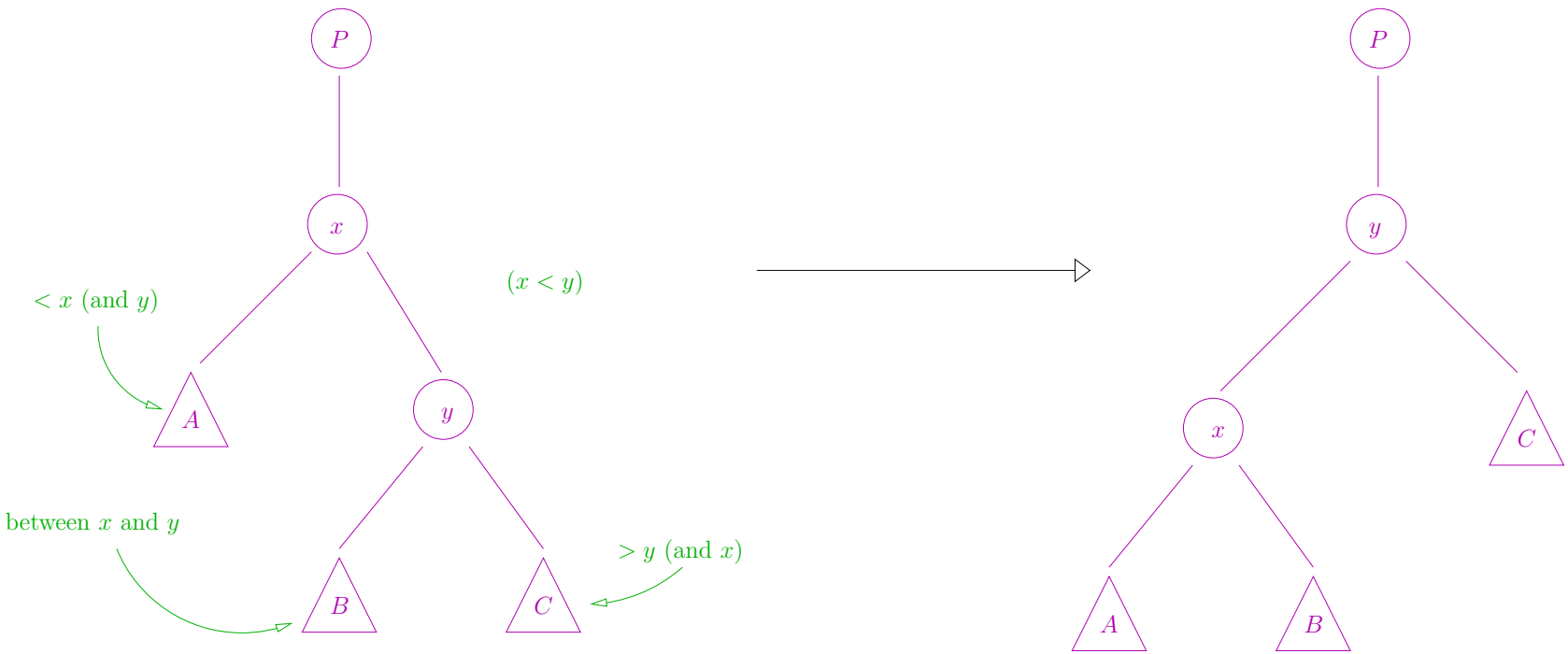
Design and Analysis
of Algorithms I

Left Rotations

Key primitive: *Rotations*. (Common to all balanced search tree implementations: red-black, AVL, B+, etc.)

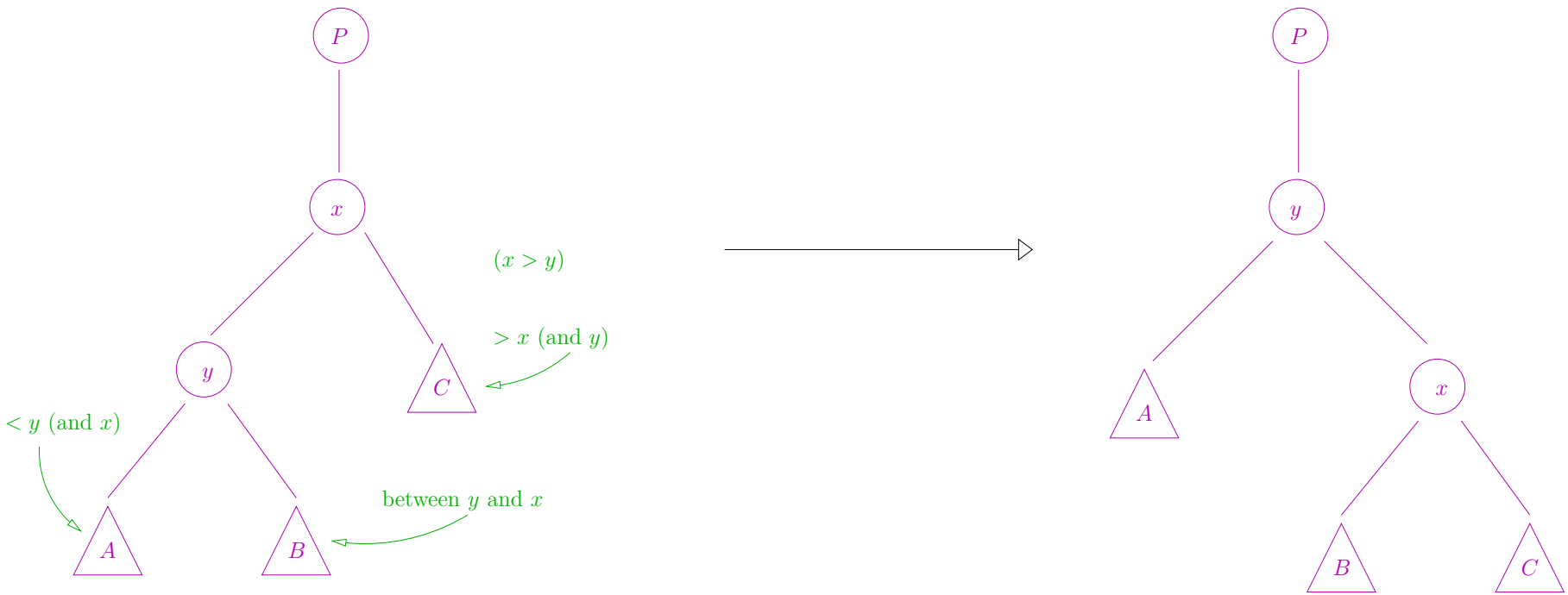
Idea: Locally rebalance subtrees at a node in $O(1)$ time.

Left rotation: (of a parent x and right child y)



Right Rotations

Right rotation:



Nice properties: Search tree property maintain, can implement in $O(1)$ time.

Data Structures

Insertion In A Red-Black Tree

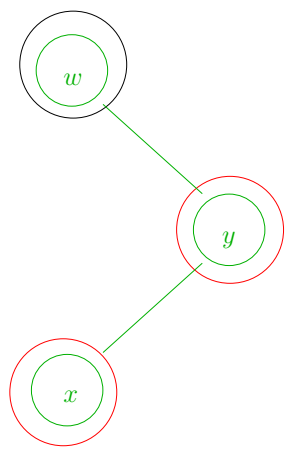
Design and Analysis of Algorithms I

High-Level Plan

Idea for Insert/Delete: Proceed as in a normal binary search tree, then recolor and/or perform rotations until invariants are restored.

Insert(x):

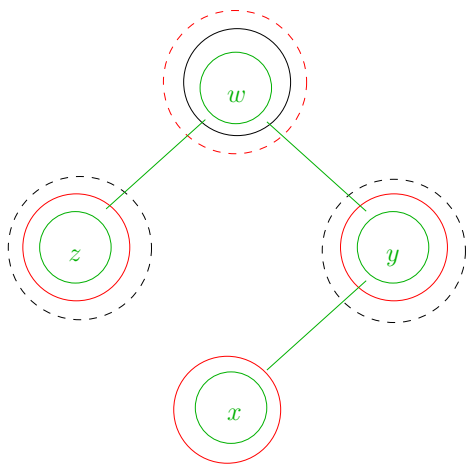
1. Insert x as usual (makes x a leaf).
2. Try coloring x red.
3. If x's parent y is black, done.
4. Else y is red $\Rightarrow$ y has a black parent w.



Insertion

Case 1

Case 1: The other child z of x's grandparent w is also red.
$\Rightarrow$ Recolor y, z black and w red. [key point: does not break invariant (4)]
$\Rightarrow$ Either restores invariant (3) or propagates the double red upward.
$\Rightarrow$ Can only happen $O(\log n)$ times. [If you reach the root, recolor it black $\Rightarrow$ Preserves invariant (4)].



Case 2

Case 2: Let x, y be the current double-red, x the deeper node. Let $w = x$'s grandparent. Suppose w's other child is NULL or is a black node z.

Exercise/case analysis (details omitted): Can eliminate double-red [$\Rightarrow$ All invariants satisfied] in $O(1)$ time via 2-3 rotations+ recolorings.