Lecture 6: Data Structures III

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: October 22, 2024

AC: 787201

Announcements

1. Third Quiz, due Friday
   - Similar format to before
   - Covers fundamental data structures (Lectures 4–6)
   - Quiz is **closed resource**
     - No books, notes, internet, etc.
     - Do not discuss until after submission deadline (Friday night, after midnight)
2. Programming Assignment (Draft) Posted Today
   - Due Wednesday, 13 November
3. Attendance Code:

787201

Meeting Goals

- Finish up heaps
  - Give an efficient array-backed PriorityQueue
- Introduce two more ADTs:
  - OrderedSet
  - Map
- Introduce binary search trees
- Discuss balanced binary search trees

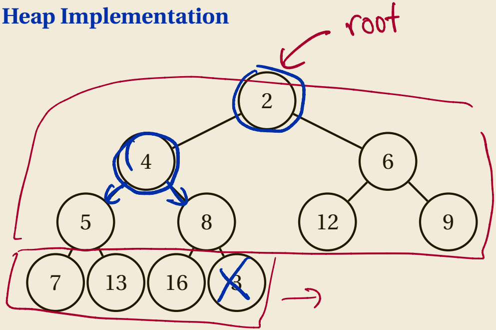
Heaps

Last Time: Priority Queues and Heaps

Priority Queues, Formally

- S is the state of the queue, initially $S = \varnothing$
- $S.\textsc{Insert}(x, p(x)) : S = x_0 x_1 \cdots x_i x_{i+1} \cdots x_{n-1} \mapsto x_0 x_1 \cdots x_i \, x \, x_{i+1} \cdots x_{n-1}$
    - where $p(x_i) \leq p(x) < p(x_{i+1})$
- $S.\textsc{Min}()$: returns x_0 where $S = x_0 x_1 \cdots x_{n-1}$
- $S.\textsc{RemoveMin}() : xS \mapsto S$, returns x

Heap Implementation



- Insert via BubbleUp procedure
- RemoveMin via TrickleDown procedure

$O(\log n)$ steps

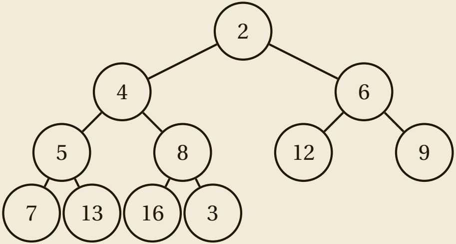
Last Time: Priority Queues and Heaps

Priority Queues, Formally

- S is the state of the queue, initially $S = \varnothing$
- $S.\textsc{Insert}(x, p(x)) : S = x_0 x_1 \cdots x_i x_{i+1} \cdots x_{n-1} \mapsto x_0 x_1 \cdots x_i \, x \, x_{i+1} \cdots x_{n-1}$
  - where $p(x_i) \le p(x) < p(x_{i+1})$
- $S.\textsc{Min}() :$ returns x_0 where $S = x_0 x_1 \cdots x_{n-1}$
- $S.\textsc{RemoveMin}() : xS \mapsto S$, returns x

Node { Value, parent, left child, right child }

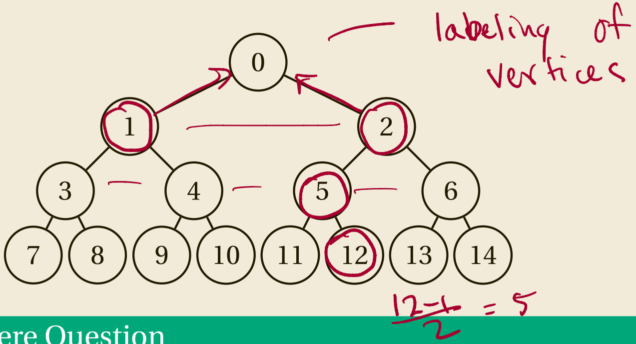
Heap Implementation



- INSERT via BUBBLEUP procedure
- REMOVEMIN via TRICKLEDOWN procedure
- **Issue:** using NODEs incurs overhead
  - locality of reference
  - storing additional references

Question. How can we represent heaps as arrays?

A Clue: Number the Vertices

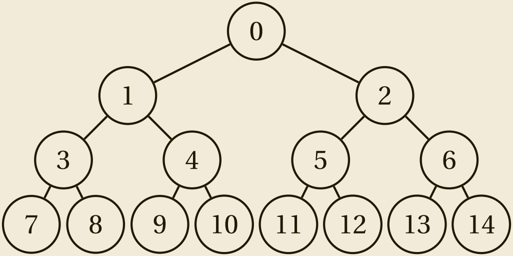


PollEverywhere Question

Suppose a vertex is assigned a label $i > 0$ in this numbering of the vertices. What is the label of i's parent in the labeling?

pollev.com/comp526

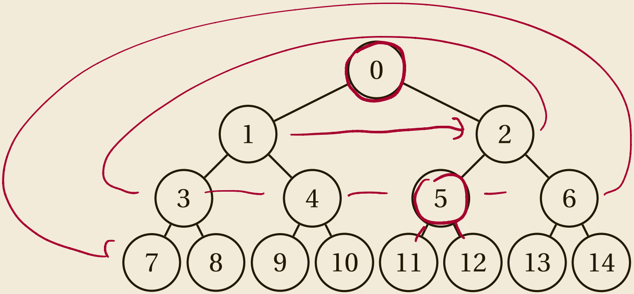
A Clue: Number the Vertices



Relationships:

- If $i > 0$, then i's parent has index $\lfloor (i-1)/2 \rfloor$ ← round down

A Clue: Number the Vertices



Relationships:

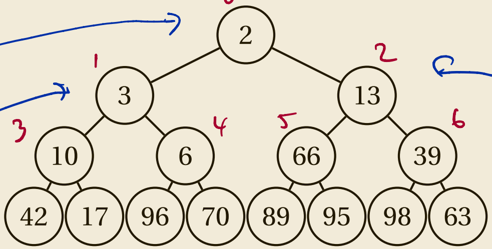
- If $i > 0$, then i's parent has index $\lfloor (i-1)/2 \rfloor$
- i's left child has index $2i+1$
- i's right child has index $2i+2$

Arrays as Heaps

Associate numbering of tree vertices as array indexes!

- If $i > 0$, then i's parent has index $\lfloor (i-1)/2 \rfloor$
- i's left child has index $2i+1$
- i's right child has index $2i+2$

Complete binary tree representation



Array representation

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	13	10	6	66	39	42	17	96	70	89	95	98	63

Example: Array BUBBLEUP

We can apply heap procedures directly to the array without reference to the tree itself!

- If $i > 0$, then i's parent has index $\lfloor (i-1)/2 \rfloor$
- i's left child has index $2i+1$
- i's right child has index $2i+2$

```
procedure INSERT(p)
    v ← new vertex storing p
    u ← first vtx with < 2 children
    add v as u's child
    PARENT(v) ← u
    while value(v) < value(u) and u ≠ ⊥ do
        SWAP(value(v), value(u))
        v ← u
        u ← PARENT(v)
    end while
end procedure
```

first index w/out element (array view)

|| heap size

Example: Array BubbleUp

We can apply heap procedures directly to the array without reference to the tree itself!

- If $i > 0$, then i's parent has index $\lfloor (i-1)/2 \rfloor$
- i's left child has index $2i+1$
- i's right child has index $2i+2$

```
procedure INSERT(p)
    v ← new vertex storing p
    u ← first vtx with < 2 children
    add v as u's child
    PARENT(v) ← u
    while value(v) < value(u) and u ≠ ⊥ do
        SWAP(value(v), value(u))
        v ← u
        u ← PARENT(v)
    end while
end procedure
```

Example. Insert(4)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	13	10	6	66	39	42	17	96	70	89	95	98	4

Array Backed Operations

Using arrays, we can define INSERT and REMOVEMIN much more cleanly!

```
procedure INSERT(p)
    i ← n                          ▷ n is heap size
    a[i] ← p
    n ← n + 1
    j ← ⌊(i − 1)/2⌋                ▷ j is i's parent
    while i > 0 and a[i] < a[j] do
        SWAP(a, i, j)
        i ← j
        j ← ⌊(i − 1)/2⌋
    end while
end procedure
```

```
procedure REMOVEMIN
    m ← a[0]
    a[0] ← a[n − 1]
    n ← n − 1
    i ← 0
    j ← argmin {a[2i + 1], a[2i + 2]}
    while j < n and a[i] > a[j] do
        SWAP(a, i, j)
        i ← j
        j ← argmin {a[2i + 1], a[2i + 2]}
    end while
    return m
end procedure
```

Both of these operations still complete after $O(\log n)$ iterations

- very little overhead, since only array operations are used!

Ordered Sets and Maps

Adding Order to Elements

priorities

Question. What made our operations on heaps efficient?

- **Answer:** Order! We can order/compare priorities.

Two more ADT with **ordered** elements:

Ordered Sets store a collection (set) of *distinct* elements from an ordered universe.

- CONTAINS(x) check if the set contains $x' = x$ and return x'
- ADD(x) add x to the set if x was not present
- REMOVE(x) remove x if x was present

Adding Order to Elements

Question. What made our operations on heaps efficient?

- **Answer:** Order! We can order/compare priorities.

Two more ADT with **ordered** elements:

Ordered Sets store a collection (set) of *distinct* elements from an ordered universe.

- CONTAINS(x) check if the set contains $x' = x$ and return x'
- ADD(x) add x to the set if x was not present
- REMOVE(x) remove x if x was present

a[k] ← v

a[k]

Maps[a] store a collection of *values* with associated ordered *keys* with array-like access.

- PUT(k, v) set the value associated with key k to v
- GET(k) return the value associated with key k
- REMOVE(k) remove the pair associated with k
- CONTAINS(k) check if the map contains a value associated with k

[a] Aka: associative arrays, dictionaries (Python `dict`), symbol table

Ordered Sets vs Maps

Ordered Sets

- CONTAINS(x) check if the set contains $x' = x$ and return x'
- ADD(x) add x to the set if x was not present
- REMOVE(x) remove x if x was present

Maps

- PUT(k, v) set the value associated with key k to v
- GET(k) return the value associated with key k
- REMOVE(k) remove the pair associated with k
- CONTAINS(k) check if the map contains a value associated with k

PollEverywhere Question

If we are given an ORDEREDSET implementation, how could we use it to implement a MAP?

pollev.com/comp526

Ordered Sets vs Maps

Ordered Sets

- CONTAINS(x) check if the set contains $x' = x$ and return x'
- ADD(x) add x to the set if x was not present
- REMOVE(x) remove x if x was present

Maps

- PUT(k, v) set the value associated with key k to v
- GET(k) return the value associated with key k
- REMOVE(k) remove the pair associated with k
- CONTAINS(k) check if the map contains a value associated with k

Maps via Ordered Sets

- Create an ordered set that stores pairs (k, v) (tuple)
- Compare $(k, v) \leq (k', v') \iff k \leq k'$
- CONTAINS, REMOVE are same
- To PUT(k, v), use REMOVE($(k, \cdot)$) then ADD((k, v))
- To GET(k), use $(k, v) \leftarrow$ CONTAINS($(k, \cdot)$) and return v

Ordered Sets via Arrays

ORDEREDSETS can be implemented by arrays:

- Maintain a sorted array $a = [x_0, x_1, \ldots, x_n]$ with each $x_i \leq x_{i+1}$.
- ADD(x) and REMOVE(x) implemented in $\Theta(n)$ worst case time
  - To ADD find index i such that $x_i \leq x < x_{i+1}$
  - Shift elements x_j with $j \geq i+1$ to next index
    - This uses $\Theta(n)$ time
  - Set $a[i+1] \leftarrow x$

Example. How to ADD(42)?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	10	28	31	34	39	51	63	70	74	82	87	91	95	

42 goes here

$\Omega(n)$

Ordered Sets via Arrays

OrderedSets can be implemented by arrays:

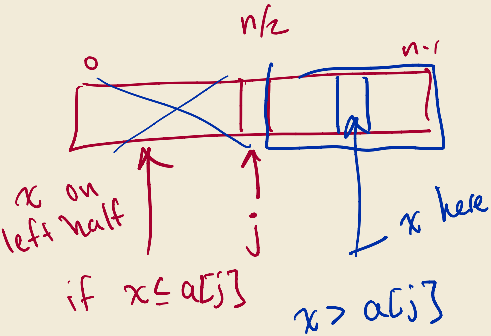
- Maintain a sorted array $a = [x_0, x_1, \ldots, x_n]$ with each $x_i \leq x_{i+1}$.
- Add(x) and Remove(x) implemented in $\Theta(n)$ worst case time
  - To Add find index i such that $x_i \leq x < x_{i+1}$
  - Shift elements x_j with $j \geq i+1$ to next index
    - This uses $\Theta(n)$ time
  - Set $a[i+1] \leftarrow x$

Question. How can we implement Contains(x) more quickly?

Efficient Search

Idea. Binary Search:

- Start at the *middle index j*
  - $x \leq a[j] \implies$ index of x must be $i \leq j$
  - otherwise $i > j$
- Apply procedure to remaining interval with half excluded
  - compare x to midpoint of remaining interval
  - eliminate half of the interval
- Repeat



Efficient Search

Idea. Binary Search:

- Start at the *middle index* j
  - $x \leq a[j] \implies$ index of x must be $i \leq j$
  - otherwise $i > j$
- Apply procedure to remaining interval with half excluded
  - compare x to midpoint of remaining interval
  - eliminate half of the interval
- Repeat

```
procedure BinarySearch(x)
    i ← 0, k ← n − 1
    j ← ⌊(i + k)/2⌋
    while i < j do
        if x ≤ a[j] then
            k ← j
        else
            i ← j
        end if
    end while
    return i
end procedure
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	10	28	31	34	39	42	51	63	70	74	82	87	91	95

Efficiency of Binary Search

PollEverywhere

What is the (worst case) running time of BinarySearch on an array of length n?

pollev.com/comp526

```
procedure BinarySearch(x)
    i ← 0, k ← n − 1
    j ← ⌊(i + k)/2⌋
    while i < j do
        if x ≤ a[j] then
            k ← j
        else
            i ← j
        end if
    end while
    return i
end procedure
```

Efficiency of Binary Search

Proposition

The worst-case running time of BinarySearch is $\Theta(\log n)$.

```
procedure BinarySearch(x)
    i ← 0, k ← n − 1
    j ← ⌊(i + k)/2⌋
    while i < j do
        if x ≤ a[j] then
            k ← j
        else
            i ← j
        end if
    end while
    return i
end procedure
```

Efficiency of Binary Search

size of active interval

Proposition

The worst-case running time of BINARYSEARCH is $\Theta(\log n)$.

Proof.

- Consider the value of $k - i$.
- After ℓ iterations of the loop, have $k - i \leq \frac{n}{2^\ell}$ (induction)
- Termination when $k - i \leq 1$
- $\ell = \lceil \log n \rceil + 1 \implies \frac{n}{2^\ell} \leq 1$

□

log n rounded up

process terminates.

```
procedure BinarySearch(x)
    i ← 0, k ← n − 1
    j ← ⌊(i + k)/2⌋
    while i < j do
        if x ≤ a[j] then
            k ← j
        else
            i ← j
        end if
    end while
    return i
end procedure
```

Making All Operations Efficient?

A Nagging Question

For OrderedSets, we can perform all operations in $o(n)$ time?

- Array implementation only gives Contains in $O(\log n)$ time
- Other operations are $\Theta(n)$
- This seems harder than efficient PriorityQueue as elements can be added *and* removed from anywhere in the data structure

Making All Operations Efficient?

A Nagging Question

For OrderedSets, we can perform all operations in $o(n)$ time?

- Array implementation only gives Contains in $O(\log n)$ time
- Other operations are $\Theta(n)$
- This seems harder than efficient PriorityQueue as elements can be added *and* removed from anywhere in the data structure

Up next: A solution in two parts

1. Binary Search Trees
2. Balancing Binary Trees

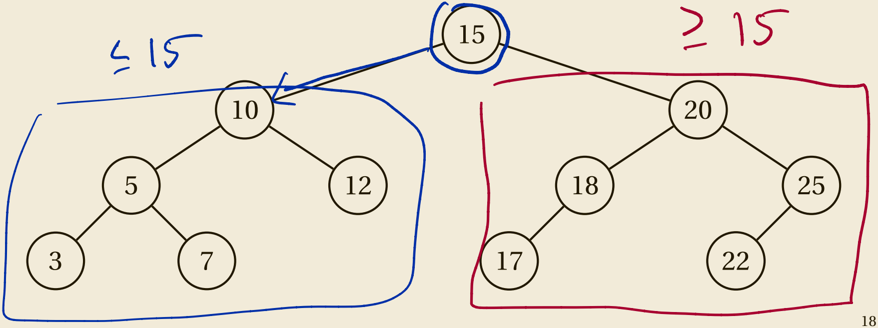
Binary Search Trees

Binary Search Tree Definition

Definition

Suppose T is a binary tree and every vertex v in T has an associated value. We say T is a **binary search tree** (**BST**) if for every vertex (value) v:

1. every *left descendant* u satisfies $u \leq v$,
2. every *right descendant* w satisfies $w \geq v$.

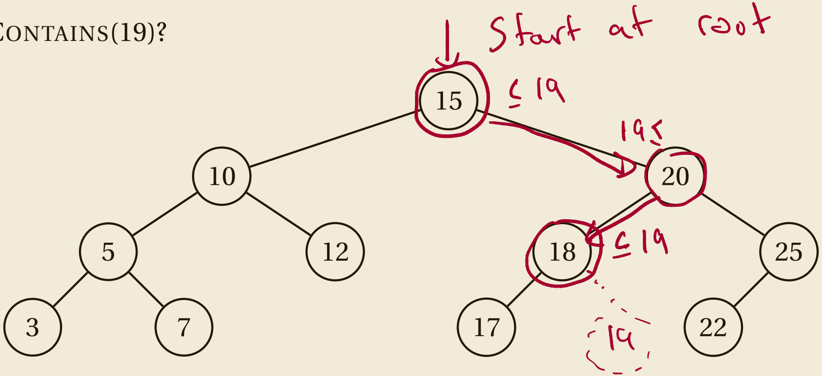


BST Search

Question

Given a BST T, how can we search for a value x in T?

CONTAINS(19)?



- 19 must be to right of 18
- 18 doesn't have right child

⇒ 19 not in BST

BST Search

Question

Given a BST T, how can we search for a value x in T?

$\perp$ "perp" to indicate non-existent node

procedure CONTAINS(x)
  v = tree root
  while $v \neq x$ and $v \neq \perp$ **do**
    if $x < v$ **then**
      $v \leftarrow$ LEFTCHILD(v) go left
    else
      $v \leftarrow$ RIGHTCHILD(v) go right
    end if
  end while
  return v
end procedure

BST Search

Question

Given a BST T, how can we search for a value x in T?

procedure CONTAINS(x)
  $v =$ tree root
  while $v \neq x$ and $v \neq \perp$ **do**
    if $x < v$ **then**
      $v \leftarrow$ LEFTCHILD(v)
    else
      $v \leftarrow$ RIGHTCHILD(v)
    end if
  end while
  return v
end procedure

PollEverywhere

What is the (worst case) running time of CONTAINS on a tree with n vertices?

pollev.com/comp526

BST CONTAINS Efficiency

Observation

The (worst-case) running time of CONTAINS on T is $\Theta(h)$ where h is the **height** of T

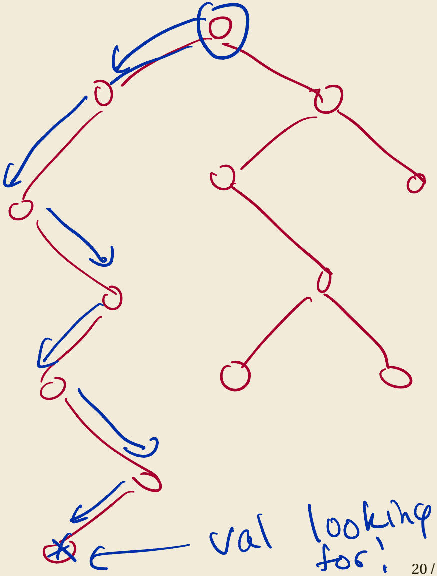
- h is the length of the longest path from root to any leaf in T

The height of T can be:

- As small as $\log n$
- As large as $n - 1$

The Moral

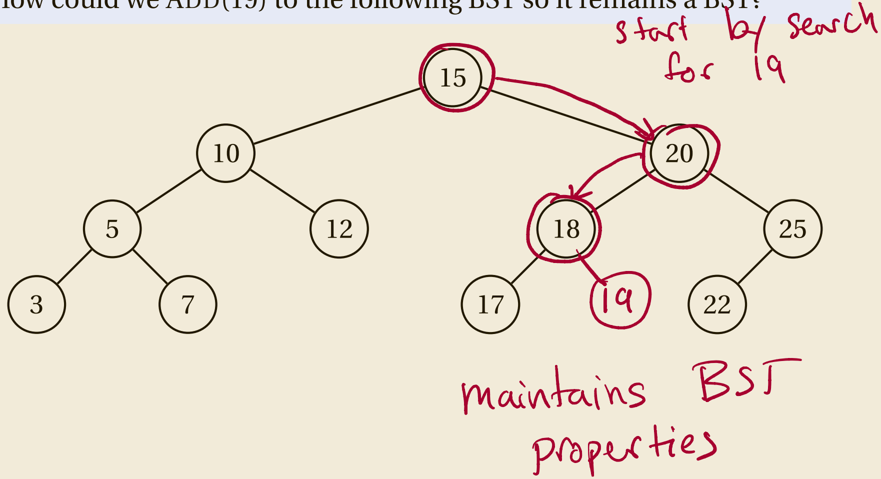
The efficiency of CONTAINS depends on the structure of T.



BST Add

Question

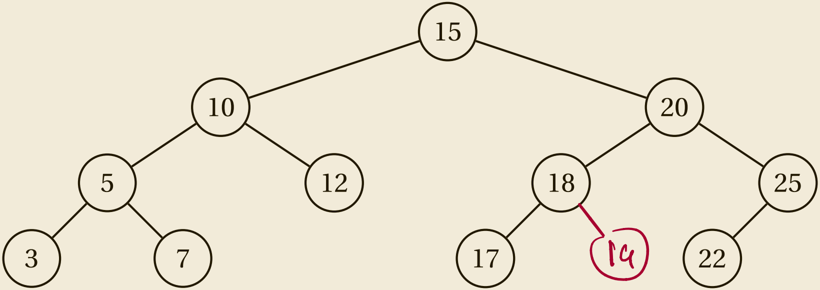
How could we ADD(19) to the following BST so it remains a BST?



BST Add

Question

How could we ADD(19) to the following BST so it remains a BST?



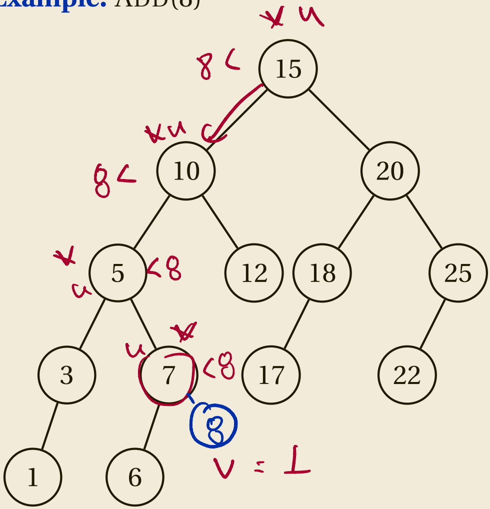
Observation. To ADD(x), we should add a new vertex wherever the CONTAINS(x) execution fails to find x.

Adding in Pseudocode

```
procedure ADD(x)
    v, u ← root
    while v ≠ ⊥ do
        if x = v then
            return
        else if x < v then
            u ← v
            v ← LEFTCHILD(v)
        else
            u ← v
            v ← RIGHTCHILD(v)
        end if
    end while
    if x < u then
        set x as u's left child
    else
        set x as u's right child
    end if
end procedure
```

missing vertex

Example. ADD(8)



Adding in Pseudocode

```
procedure ADD(x)
    v, u ← root
    while v ≠ ⊥ do
        if x = v then
            return
        else if x < v then
            u ← v
            v ← LEFTCHILD(v)
        else
            u ← v
            v ← RIGHTCHILD(v)
        end if
    end while
    if x < v then
        set x as v's left child
    else
        set x as v's right child
    end if
end procedure
```

PollEverywhere Question

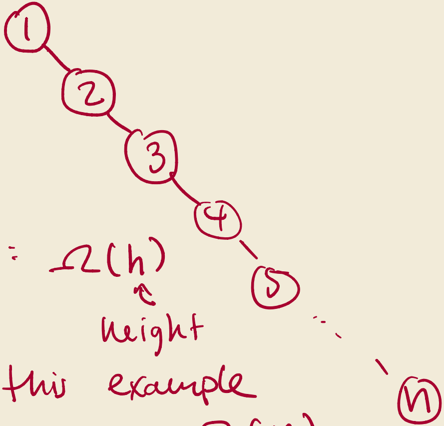
Describe a sequence of ADD(x) operations starting from an empty BST such that every operation takes $\Omega(n)$ time.

pollev.com/comp526

Adding in Pseudocode

```
procedure ADD(x)
    v, u ← root
    while v ≠ ⊥ do
        if x = v then
            return
        else if x < v then
            u ← v
            v ← LEFTCHILD(v)
        else
            u ← v
            v ← RIGHTCHILD(v)
        end if
    end while
    if x < v then
        set x as v's left child
    else
        set x as v's right child
    end if
end procedure
```

A Bad Sequence: 1 2 3 ⋯ n



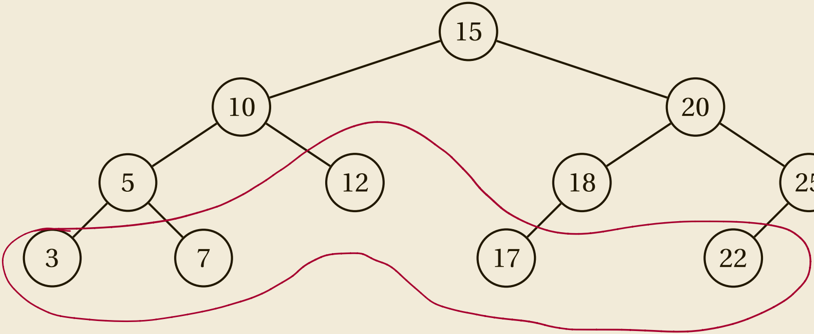
ops: $\Omega(h)$ ← height

For this example $h = n-1 \Rightarrow \Omega(n)$ ops.

BST Remove

Question

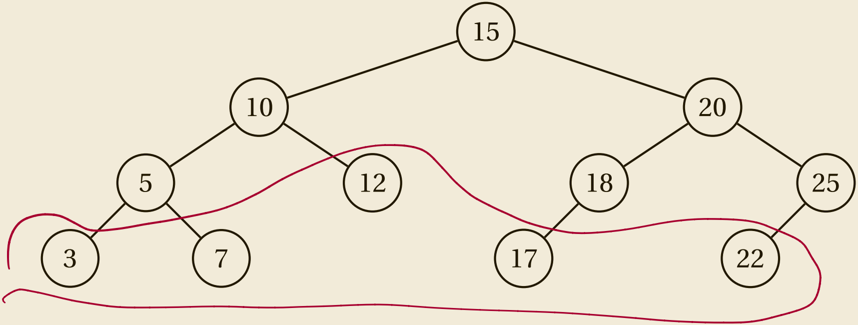
How could we remove an element from a BST?



BST Remove

Question

How could we remove an element from a BST?

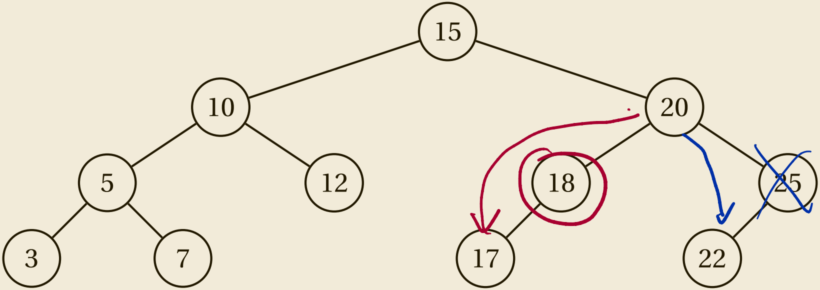


Case 1: A leaf. Just remove it!

BST Remove

Question

How could we remove an element from a BST?

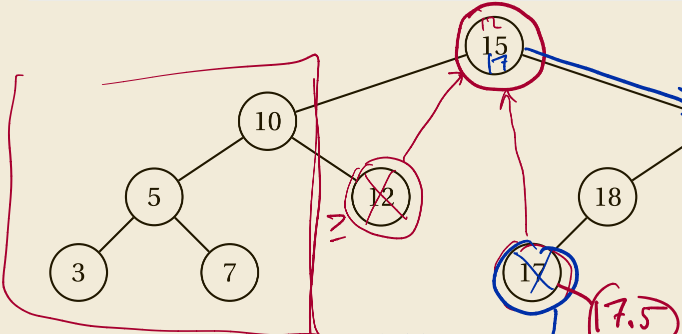


Case 2: A vertex v with single child. Splice! Set v's child to be its parent's child.

BST Remove

Question

How could we remove an element from a BST?



Case 3: A vertex *v* with two children.

1. Find *next smallest* value *w*.
2. Copy *w*'s value to *v*.
3. Remove *w*

must have 0 or 1 children, so remove accordingly

So Far…

…we've implemented

- CONTAINS(x)
- ADD(x)
- REMOVE(x)

for ORDEREDSETS.

But we haven't improved *efficiency*

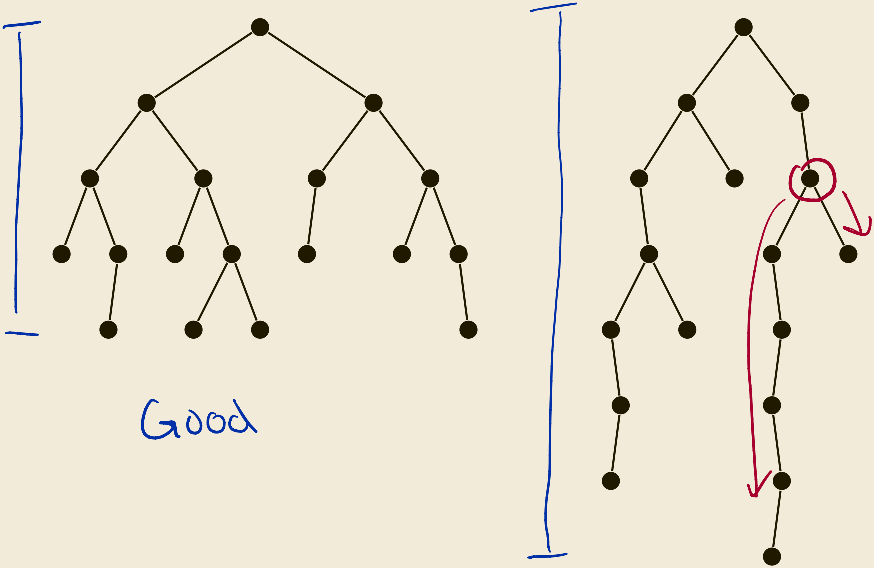
- All of these operations can cost as much as $\Theta(n)$
  - efficiency depends on previous operations performed!

Idea. We can *restructure* BSTs.

- Goal: ensure that the BST has small **height**.
- After each update, check and update tree structure.
  - maintain BST property
  - updates performed efficiently

Balanced Binary Trees

Distinguishing the Good from the Bad



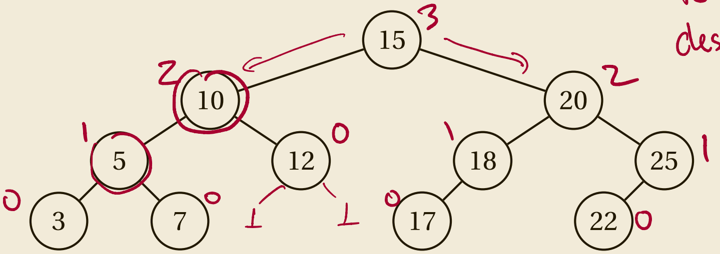
Height Balanced Trees

Definition (Left and Right Height)

Let v be a vertex in a tree. We define:

- $h(\perp) = -1$
- $h(v) = 1 + \max(h(\text{LEFTCHILD}(v)), h(\text{RIGHTCHILD}(v)))$
- $h_\ell(v) = h(\text{LEFTCHILD}(v))$
- $h_r(v) = h(\text{RIGHTCHILD}(v))$

= dist to farthest descendent leaf



Height Balanced Trees

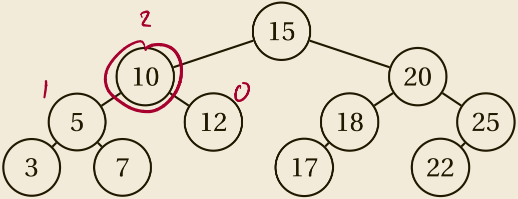
Definition (Left and Right Height)

Let v be a vertex in a tree. We define:

- $h(\perp) = -1$
- $h(v) = 1 + \max(h(\textsc{LeftChild}(v)), h(\textsc{RightChild}(v)))$
- $h_\ell(v) = h(\textsc{LeftChild}(v))$
- $h_r(v) = h(\textsc{RightChild}(v))$

Def. (Height Balanced)

We call a tree **height balanced** if for every vertex v, $|h_\ell(v) - h_r(v)| \le 1$.



Properties of Height Balanced Trees

Proposition

Suppose T is a height balanced tree of height h. Then T has $n \geq 2^{h/2}$ vertices.

Properties of Height Balanced Trees

Proposition

Suppose T is a height balanced tree of height h. Then T has $n \geq 2^{h/2}$ vertices.

Proof.

Let $M(h)$ denote the minimum size of a height balanced tree of height h.

- Observe that $M(0) = 1$, $M(1) = 2$.
- In general $M(h) \geq 1 + M(h-1) + M(h-2)$
  - one subtree of the root is a height balanced tree of height $h-1$
  - other subtree is height balanced with height at least $h-2$
- So $M(h) \geq 2M(h-2)$
- Inductive argument $\implies M(h) \geq 2^{h/2}$.

□

Properties of Height Balanced Trees

Proposition

Suppose T is a height balanced tree of height h. Then T has $n \geq 2^{h/2}$ vertices.

Consequences.

If T is a height balanced tree with n vertices, then its height h satisfies $h \leq 2 \log n$

$\implies$ CONTAINS(x) takes time $O(\log n)$

$\implies$ ADD(x) takes time $O(\log n)$

$\implies$ REMOVE(x) takes time $O(\log n)$

Maintaining Height Balance

Our Strategy. Maintain a BST that is height balanced **for any sequence of operations performed**.

- No one is *forcing* us to keep the tree structure determined by our ADD/REMOVE operations
  - there are many valid BSTs that store the same collection of elements!

Maintaining Height Balance

Our Strategy. Maintain a BST that is height balanced **for any sequence of operations performed**.

- No one is *forcing* us to keep the tree structure determined by our ADD/REMOVE operations
  - there are many valid BSTs that store the same collection of elements!
- Starting from a balanced tree, ADD(x) may introduce imbalance.
- If imbalance is introduced try to fix it:
  - find closest unbalanced vertex to x and correct its balance
  - look for other imbalance and correct it

Maintaining Height Balance

Our Strategy. Maintain a BST that is height balanced **for any sequence of operations performed**.

- No one is *forcing* us to keep the tree structure determined by our ADD/REMOVE operations
  - there are many valid BSTs that store the same collection of elements!
- Starting from a balanced tree, ADD(x) may introduce imbalance.
- If imbalance is introduced try to fix it:
  - find closest unbalanced vertex to x and correct its balance
  - look for other imbalance and correct it

For next time. Think about how you could implement this strategy.

- *Where* could imbalance occur? And how much?
- What *local* operations can fix the imbalance?
- What is the worst-case running time of restoring balance?

Next Time: Sorting

- Finishing Balanced BSTs
- The Sorting Task
- Efficient Sorting by Divide and Conquer

Scratch Notes