2 Fundamental Data Structures

6 October 2023

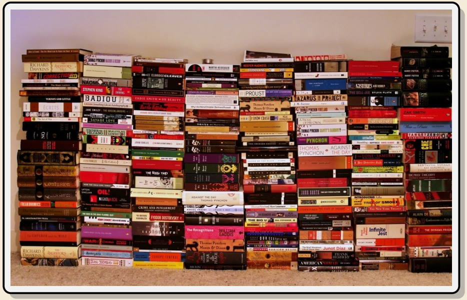
Sebastian Wild

COMP526 (Fall 2023)
University of Liverpool
version `2023-10-04 22:02`

Learning Outcomes

1. Understand and demonstrate the difference between *abstract data type (ADT)* and its *implementation*
2. Be able to define the ADTs *stack*, *queue*, *priority queue* and *dictionary / symbol table*
3. Understand *array*-based implementations of stack and queue
4. Understand *linked lists* and the corresponding implementations of stack and queue
5. Know *binary heaps* and their performance characteristics
6. Understand *binary search trees* and their performance characteristics

Unit 2: *Fundamental Data Structures*



Outline

2 Fundamental Data Structures

Recap: The Random Access Machine

- Data structures make heavy use of pointers and dynamically allocated memory.

- Recall: Our RAM model supports
  - basic pseudocode (≈ simple Python code)
  - creating arrays of a fixed/known size.
  - creating instances (objects) of a known class.

[1, 2, 3]

Python abstracts this away!

no predefined capacity!

There are *no arrays* in Python, only its built-in *lists*.

But: Python *implementations create* lists based on fixed-size arrays (stay tuned!)

Python ≠ RAM: Not every built-in Python instruction runs in $O(1)$ time!

2.1 Stacks & Queues

Abstract Data Types

abstract data type (ADT)

- list of supported operations
- **what** should happen
- **not:** how to do it
- **not:** how to store data

VS.

abstract base classes

≈ Java `interface`, Python `ABCs` (with comments)

data structures

- specify exactly **how** data is represented
- **algorithms** for operations
- has concrete costs (space and running time)

≈ Java/Python `class` (non abstract)

Abstract Data Types

abstract data type (ADT)

- list of supported operations
- **what** should happen
- **not:** how to do it
- **not:** how to store data

abstract base classes

≈ Java `interface`, Python `ABCs` (with comments)

VS.

data structures

- specify exactly **how** data is represented
- **algorithms** for operations
- has concrete costs (space and running time)

≈ Java/Python `class` (non abstract)

Why separate?

- Can swap out implementations ⇝ "drop-in replacements"

⇝ **reusable code**!

- (Often) better abstractions
- Prove generic lower bounds (⇝ Unit 3)

Abstract Data Types

abstract data type (ADT)

- list of supported operatic
- **what** should happen
- **not:** how to do it
- **not:** how to store data

abst

≈ Java `interface`, Python A
(with comments)

Why separate?

- Can swap out implement
- ⇝ **reusable code!**
- (Often) better abstractions
- Prove generic lower bounds (⇝ Unit 3)

Clicker Question

Which of the following are examples of abstract data types?

- **A** ADT
- **B** Stack
- **C** Deque
- **D** Linked list
- **E** binary search tree
- **F** Queue
- **G** resizable array
- **H** heap
- **I** priority queue
- **J** dictionary/symbol table
- **K** hash table

→ `sli.do/comp526`

Clicker Question

Which of the following are examples of abstract data types?

- **A** ~~ADT~~
- **B** Stack ✓
- **C** Deque ✓
- **D** ~~Linked list~~
- **E** ~~binary search tree~~
- **F** Queue ✓
- **G** ~~resizable array~~
- **H** ~~heap~~
- **I** priority queue ✓
- **J** dictionary/symbol table ✓
- **K** ~~hash table~~

→ sli.do/comp526

Stacks

Stack ADT

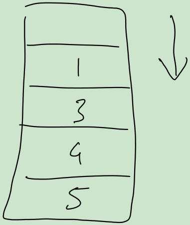
- `top()`
  Return the topmost item on the stack
  Does not modify the stack.
- `push(`x`)`
  Add x onto the top of the stack.
- `pop()`
  Remove the topmost item from the stack
  (and return it).
- `isEmpty()`
  Returns `true` iff stack is empty.
- `create()`
  Create and return an new empty stack.

Clicker Question

Suppose a stack initially contains the numbers 1, 2, 3, 4, 5 with 1 at the top.
What is the content of the stack after the following operations:

```
pop(); pop(); push(1);
```

- **A** 1,2,3,1
- **B** 3,4,5,1
- **C** 1,3,4,5
- **D** empty
- **E** 1,2,3,4,5



→ sli.do/comp526

Clicker Question

Suppose a stack initially contains the numbers $1, 2, 3, 4, 5$ with 1 at the top.
What is the content of the stack after the following operations:

```
pop(); pop(); push(1);
```

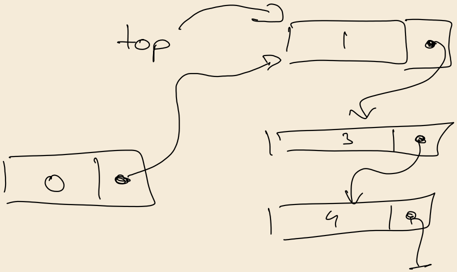
- **A** ~~1,2,3,1~~
- **B** ~~3,4,5,1~~
- **C** 1,3,4,5 ✓
- **D** ~~empty~~
- **E** ~~1,2,3,4,5~~

→ sli.do/comp526

Linked-list implementation for Stack

Invariants:

- maintain pointer *top* to topmost element
- each element points to the element below it (or `null` if bottommost)




```
class Node
    value
    next

class Stack
    top := null
    procedure top()
        return top.value
    procedure push(x)
        top := new Node(x, top)
    procedure pop()
        t := top()
        top := top.next
        return t
```

Linked-list implementation for Stack – Discussion

Linked stacks:

- 👍 require $\Theta(n)$ space when n elements on stack
- 👍 All operations take $O(1)$ time

- 👎 require $\Theta(n)$ space when n elements on stack

  Can we avoid extra space for pointers?

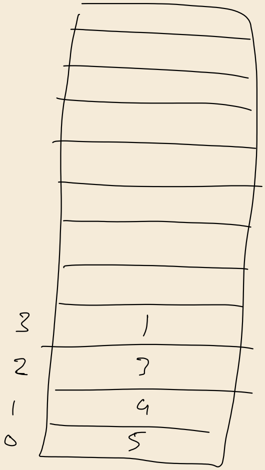
Array-based implementation for Stack

If we want no pointers $\rightsquigarrow$ array-based implementation

Invariants:

- maintain array S of elements, from bottommost to topmost
- maintain index *top* of position of topmost element in S.

top : 3



Array-based implementation for Stack

If we want no pointers ⇝ array-based implementation

Invariants:

- maintain array S of elements, from bottommost to topmost
- maintain index *top* of position of topmost element in `S`.

What to do if stack is full upon `push`?

Array stacks:

- require *fixed capacity* C (decided at creation time)!
- require $\Theta(C)$ space for a capacity of C elements
- all operations take $O(1)$ time

Queues

Operations:

- `enqueue(`x`)`
  Add x at the end of the queue.
- `dequeue()`
  Remove item at the front of the queue and return it.

Implementations similar to stacks.

Bags

What do Stack and Queue have in common?

Bags

What do Stack and Queue have in common?

They are special cases of a ***Bag***!

Operations:

- `insert(x)`
  Add x to the items in the bag.
- `delAny()`
  Remove any one item from the bag and return it.
  (Not specified which; any choice is fine.)
- roughly similar to Java's `java.util.Collection`
  Python's `collections.abc.Collection`

Sometimes it is useful to state that order is irrelevant $\rightsquigarrow$ Bag
Implementation of Bag usually just a Stack or a Queue

2.2 Resizable Arrays

Digression – Arrays as ADT

Arrays can also be seen as an ADT!

Array operations:

- `create(`n`)`   *Java*: `A = new int[`n`];`   *Python:* `A = [0] * `n
  Create a new array with n cells, with positions $0, 1, \ldots, n-1$;
  we write $A[0..n) = A[0..n-1]$

- `get(`i`)`   *Java/Python*: `A[`i`]`
  Return the content of cell i

- `set(`i`,`x`)`   *Java/Python*: `A[`i`] = `x`;`
  Set the content of cell i to x.

$\rightsquigarrow$ Arrays have *fixed* size (supplied at creation).   ($\neq$ `lists` in Python)

Digression – Arrays as ADT

Arrays can also be seen as an ADT! . . . but are commonly seen as specific data structure

Array operations:

- `create(`n`)` *Java*: `A = new int[`n`];` *Python:* `A = [0] *` n
  Create a new array with n cells, with positions $0, 1, \ldots, n-1$;
  we write $A[0..n) = A[0..n-1]$

- `get(`i`)` *Java/Python*: `A[`i`]`
  Return the content of cell i

- `set(`i`,`x`)` *Java/Python*: `A[`i`] =` x`;`
  Set the content of cell i to x.

$\rightsquigarrow$ Arrays have *fixed* size (supplied at creation). ($\neq$ `lists` in Python)

Usually directly implemented by compiler + operating system / virtual machine.

Difference to "real" ADTs: ***Implementation usually fixed***
to "a contiguous chunk of memory".

Doubling trick

Can we have unbounded stacks based on arrays? Yes!

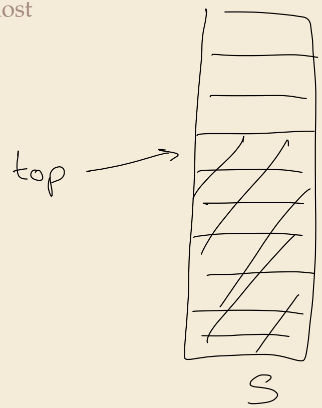
Doubling trick

Can we have unbounded stacks based on arrays? Yes!

Invariants:

- maintain array S of elements, from bottommost to topmost
- maintain index *top* of position of topmost element in S
- maintain capacity $C = S.length$ so that $\frac{1}{4}C \le n \le C$

⇝ can always push more elements!



Doubling trick

Can we have unbounded stacks based on arrays? Yes!

Invariants:

- maintain array S of elements, from bottommost to topmost
- maintain index *top* of position of topmost element in S
- maintain capacity $C = S.length$ so that $\frac{1}{4}C \leq n \leq C$

⇝ can always push more elements!

How to maintain the last invariant?

- before `push`
  If $n = C$, allocate new array of size $2n$, copy all elements.
- after `pop`
  If $n < \frac{1}{4}C$, allocate new array of size $2n$, copy all elements.

⇝ ***"Resizing Arrays"***

an implementation technique, not an ADT!

Clicker Question

Which of the following statements about resizable array that currently stores n elements is correct?

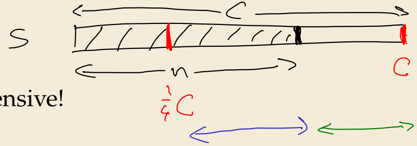
- **A** The elements are stored in an array of size $2n$.
- **B** Adding or deleting an element at the end takes constant time.
- **C** A sequence of m insertions or deletions at the end of the array takes time $O(n + m)$.
- **D** Inserting and deleting any element takes $O(1)$ amortized time.

→ `sli.do/comp526`

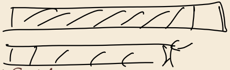
Amortized Analysis

- Any individual operation `push` / `pop` can be expensive!
  $\Theta(n)$ time to copy all elements to new array.
- **But:** An one expensive operation of cost T means $\Omega(T)$ next operations are cheap!

Amortized Analysis



- Any individual operation `push` / `pop` can be expensive!
  $\Theta(n)$ time to copy all elements to new array.
- **But:** An one expensive operation of cost T means $\Omega(T)$ next operations are cheap!



distance to boundary — since $n \le C \le 4n$

Formally: consider "credits/potential" $\Phi = \min\{n - \frac{1}{4}C, C - n\} \in [0, 0.6n]$

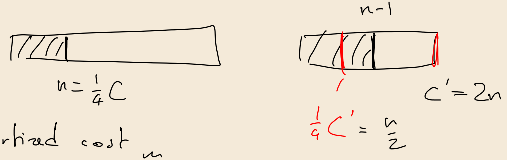
- amortized cost of an operation = actual cost (array accesses) $- 4 \cdot$ change in Φ
  - cheap `push`/`pop`: actual cost 1 array access, consumes ≤ 1 credits $\rightsquigarrow$ amortized cost ≤ 5
  - copying `push`: actual cost $2n + 1$ array accesses, creates $\frac{1}{2}n + 1$ credits $\rightsquigarrow$ amortized cost ≤ 5
  - copying `pop`: actual cost $2n + 1$ array accesses, creates $\frac{1}{2}n - 1$ credits $\rightsquigarrow$ amortized cost 5

$\rightsquigarrow$ **sequence** of m operations: total actual cost $\le$ total amortized cost + final credits

here: $\le \quad 5m \quad + \quad 4 \cdot 0.6n \quad = \Theta(m + n)$

copying push

$$2n+1-4\left(\tfrac{1}{2}n+1\right) = 1-4 = 3 \le 5 \qquad \tfrac{1}{4}C = 2n$$



$$5m \ge \sum_{i=1}^{m} a_i = \sum_{i=1}^{m} \left(c_i - 4(\Phi_i - \Phi_{i-1})\right)$$

(amortized cost $= a_i$)

$$\Rightarrow \sum_{i=1}^{m} c_i \le 5m + 4\Phi_m - 4\Phi_0 \qquad = \sum_{i=1}^{m} c_i - 4\sum_{i=1}^{m} \Phi_i - \Phi_{i-1}$$

$$\le 5m + 2.4n \qquad = \sum_{i=1}^{m} c_i - 4(\Phi_m - \Phi_0)$$

Clicker Question

Which of the following statements about resizable array that currently stores n elements is correct?

A The elements are stored in an array of size $2n$.

B Adding or deleting an element at the end takes constant time.

C A sequence of m insertions or deletions at the end of the array takes time $O(n + m)$.

D Inserting and deleting any element takes $O(1)$ amortized time.

→ sli.do/comp526

Clicker Question

Which of the following statements about resizable array that currently stores n elements is correct?

- **A** ~~The elements are stored in an array of size $2n$.~~
- **B** ~~Adding or deleting an element at the end takes constant time.~~
- **C** A sequence of m insertions or deletions at the end of the array takes time $O(n + m)$. ✓
- **D** ~~Inserting and deleting any element takes $O(1)$ amortized time.~~

→ sli.do/comp526

continue 12:07

2.3 Priority Queues & Binary Heaps

Clicker Question

What is a heap-ordered tree?

- **A** A tree in which every node has exactly 2 children.
- **B** A tree where all keys in the left subtree are smaller than the key at the root and all keys in the right subtree are bigger than the key at the root.
- **C** A tree where all keys in the left subtree and right subtree are smaller than the key at the root.
- **D** An tree that is stored in the heap-area of the memory.

→ *sli.do/comp526*

Priority Queue ADT

Now: elements in the bag have different *priorities*.

(Max-oriented) Priority Queue (MaxPQ):

- `construct(`A`)`
  Construct from from elements in array A.
- `insert(`x`,`p`)`
  Insert item x with priority p into PQ.
- `max()`
  Return item with largest priority. (Does not modify the PQ.)
- `delMax()`
  Remove the item with largest priority and return it.
- `changeKey(`x`,`p'`)`
  Update x's priority to p'.
  Sometimes restricted to *increasing* priority.
- `isEmpty()`

Fundamental building block in many applications.

Priority Queue ADT – min-oriented version

Now: elements in the bag have different *priorities*.

(~~Max~~ Min-oriented) Priority Queue (~~Max~~ MinPQ):

- `construct(`A`)`
  Construct from from elements in array A.
- `insert(`x`,`p`)`
  Insert item x with priority p into PQ.
- ~~`max`~~ `min()`
  Return item with ~~largest~~ smallest priority. (Does not modify the PQ.)
- `del`~~`Max`~~`Min()`
  Remove the item with ~~largest~~ smallest priority and return it.
- `changeKey(`x`,`p'`)`
  Update x's priority to p'.
  Sometimes restricted to ~~*in*~~ *decreasing* priority.
- `isEmpty()`

Fundamental building block in many applications.

Clicker Question

Suppose we start with an empty priority queue and insert the numbers $7, 2, 4, 9, 1$ in that order. What is the result of `delMax()`?

A $-\infty$

B 1

C 2

D 4

E 7

F 9

G not allowed

→ *sli.do/comp526*

Clicker Question

Suppose we start with an empty priority queue and insert the numbers $7, 2, 4, 9, 1$ in that order. What is the result of `delMax()`?

- **A** ~~$-\infty$~~
- **B** ~~1~~
- **C** ~~2~~
- **D** ~~4~~
- **E** ~~7~~
- **F** 9 ✓
- **G** ~~not allowed~~

→ sli.do/comp526

PQ implementations

Elementary implementations

- unordered list $\rightsquigarrow$ $\Theta(1)$ `insert`, but $\Theta(n)$ `delMax`
- sorted list $\rightsquigarrow$ $\Theta(1)$ `delMax`, but $\Theta(n)$ `insert`

PQ implementations

Elementary implementations

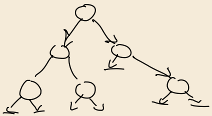
- unordered list $\rightsquigarrow$ $\Theta(1)$ `insert`, but $\Theta(n)$ `delMax`
- sorted list $\rightsquigarrow$ $\Theta(1)$ `delMax`, but $\Theta(n)$ `insert`

Can we get something between these extremes? Like a "slightly sorted" list?

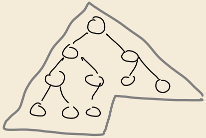
PQ implementations

Elementary implementations

- unordered list ⇝ $\Theta(1)$ `insert`, but $\Theta(n)$ `delMax`
- sorted list ⇝ $\Theta(1)$ `delMax`, but $\Theta(n)$ `insert`



binary tree



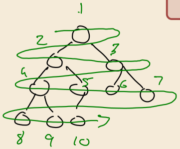
complete bin. tree

Can we get something between these extremes? Like a "slightly sorted" list?

Yes! *Binary heaps.*

Array view

Heap = array A with
$\forall i \in [n] : A[\lfloor i/2 \rfloor] \geq A[i]$



level order

$\equiv$

store nodes
in level order
in $A[1..n]$

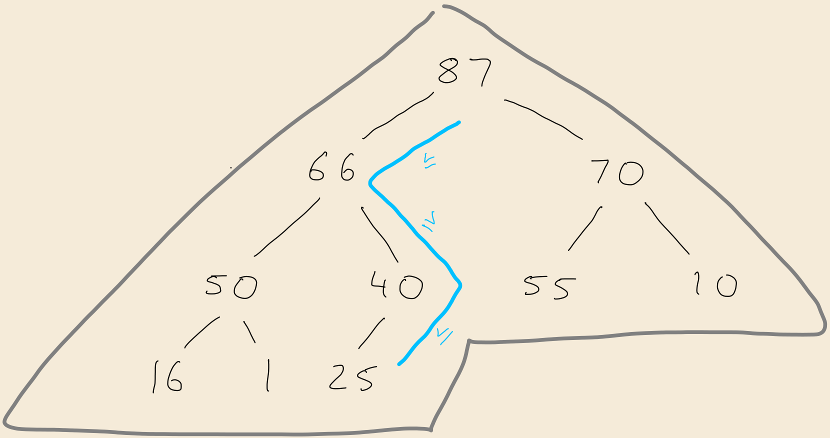
Tree view

Heap = tree that is
(i) a complete binary tree
(ii) heap ordered

all but last level full
last level flush left

parent ≥ children

Binary heap example



Why heap-shaped trees?

Why complete binary tree shape?

- only one possible tree shape $\rightsquigarrow$ keep it simple!
- complete binary trees have minimal height among all binary trees
- simple formulas for moving from a node to parent or children:

  For a node at index k in A

  $k \geq 1$ indices

  - parent at $\lfloor k/2 \rfloor$ (for $k \geq 2$)
  - left child at $2k$
  - right child at $2k + 1$

Why heap-shaped trees?

Why complete binary tree shape?

- only one possible tree shape $\rightsquigarrow$ keep it simple!
- complete binary trees have minimal height among all binary trees
- simple formulas for moving from a node to parent or children:
  For a node at index k in A
  - parent at $\lfloor k/2 \rfloor$ (for $k \geq 2$)
  - left child at $2k$
  - right child at $2k+1$

Why heap ordered?

- Maximum must be at root! $\rightsquigarrow$ `max()` is trivial!
- But: Sorted only along paths of the tree; leaves lots of leeway for fast inserts

how? . . . stay tuned

Clicker Question

What is a heap-ordered tree?

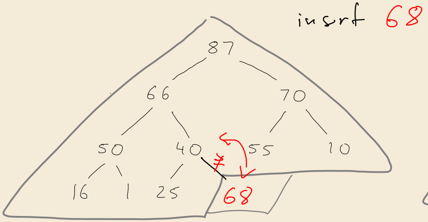
- **A** ~~A tree in which every node has exactly 2 children.~~
- **B** ~~A tree where all keys in the left subtree are smaller than the key at the root and all keys in the right subtree are bigger than the key at the root.~~
- **C** A tree where all keys in the left subtree and right subtree are smaller than the key at the root. ✓
- **D** ~~An tree that is stored in the heap-area of the memory.~~

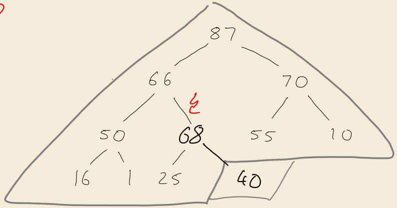
→ sli.do/comp526

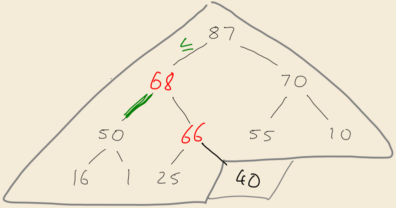
2.4 Operations on Binary Heaps

Insert

1. Add new element at only possible place: bottom-most level, next free spot.
2. Let element *swim* up to repair heap order. — at most height times

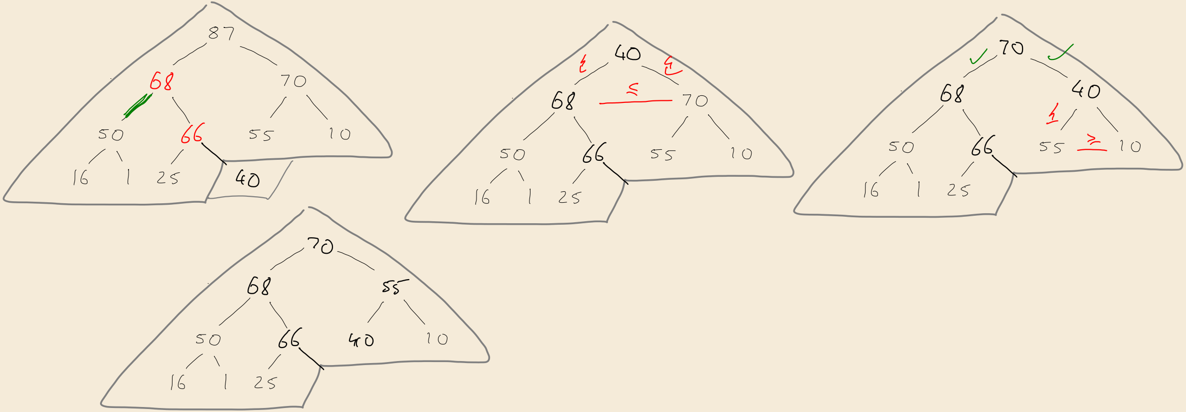






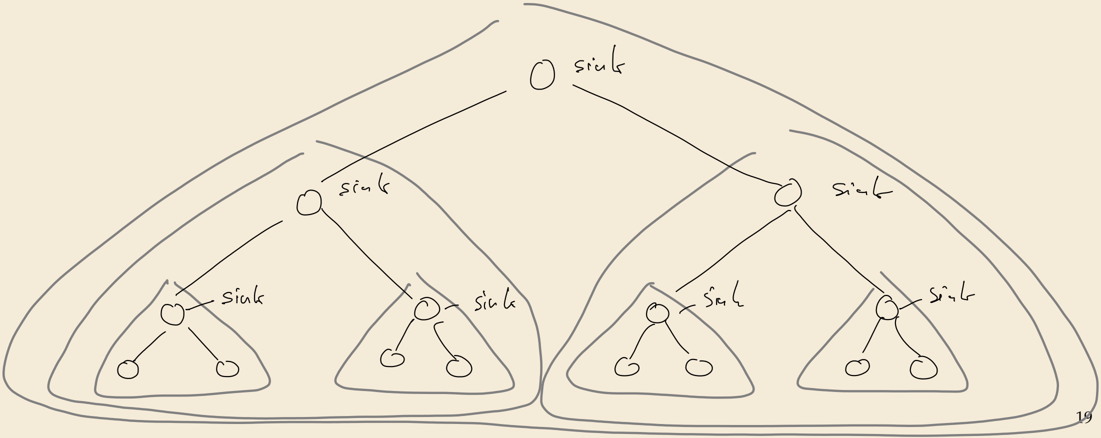
Delete Max

1. Remove max (must be in root).
2. Move last element (bottom-most, rightmost) into root.
3. Let root key *sink* in heap to repair heap order.

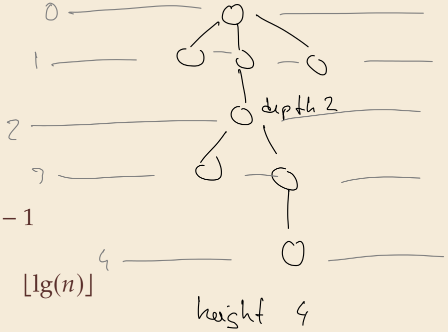


Heap construction

- n times insert $\rightsquigarrow$ $\Theta(n \log n)$
- instead:
  1. Start with singleton heaps (one element)
  2. Repeatedly merge two heaps of height k with new element into heap of height $k + 1$



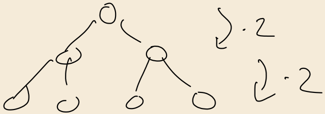
Analysis



Height of binary heaps:

- *height* of a tree: # edges on longest root-to-leaf path
- *depth/level* of a node: # edges from root ⇝ root has depth 0
- How many nodes on first k *full* levels? $\sum_{\ell=0}^{k} 2^{\ell} = 2^{k+1} - 1$

⇝ Height of binary heap: $h \;=\; \min k \text{ s.t. } 2^{k+1} - 1 \geq n \;=\; \lfloor \lg(n) \rfloor$



Analysis

Height of binary heaps:

- *height* of a tree: # edges on longest root-to-leaf path
- *depth/level* of a node: # edges from root $\rightsquigarrow$ root has depth 0
- How many nodes on first k *full* levels? $\displaystyle\sum_{\ell=0}^{k} 2^{\ell} = 2^{k+1} - 1$

$\rightsquigarrow$ Height of binary heap: $h \;=\; \min k \text{ s.t. } 2^{k+1} - 1 \ge n \;=\; \lfloor \lg(n) \rfloor$

Analysis:

- `insert`: new element "swims" up $\rightsquigarrow$ $\le h$ steps (h cmps)
- `delMax`: last element "sinks" down $\rightsquigarrow$ $\le h$ steps ($2h$ cmps)
- `construct` from n elements:

cost $=$ cost of letting *each node* in heap sink!

$$\le 1 \cdot h + 2 \cdot (h-1) + 4 \cdot (h-2) + \cdots + 2^{\ell} \cdot (h - \ell) + \cdots + 2^{h-1} \cdot 1 + 2^{h} \cdot 0$$

$$= \sum_{\ell=0}^{h} 2^{\ell}(h-\ell) \;=\; \sum_{i=0}^{h} \frac{2^h}{2^i} i \;=\; 2^h \sum_{i=0}^{h} \frac{i}{2^i} \;\le\; 2 \cdot 2^h \;\le\; 4n$$

Binary heap summary

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`max()`	$O(1)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

2.5 Symbol Tables

Clicker Question

Have you ever used a physical dictionary (i. e., a printed book with, say, English vocabulary)?

A Yes

B No

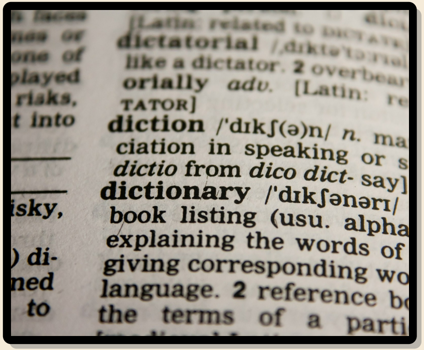
→ sli.do/comp526

Symbol table ADT

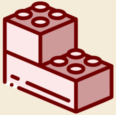
Java: `java.util.Map<K,V>`

Symbol table / Dictionary / Map / Associative array / key-value store:

Python dict `{k:v}`



- put(k,v) *Python* dict: d[k] = v
  Put key-value pair (k, v) into table
- get(k) *Python* dict: d[k]
  Return value associated with key k
- delete(k) *Python* dict: del d[k]
  Remove key k (any associated value) form table
- contains(k) *Python* dict: k in d
  Returns whether the table has a value for key k
- isEmpty(), size()
- create()



Most fundamental building block in computer science.

(Every programming library has a symbol table implementation.)

Symbol tables vs. mathematical functions

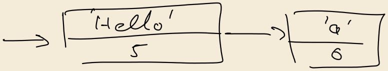
- similar interface
- but: mathematical functions are *static/immutable* (never change their mapping)
  (Different mapping is a *different* function)
- symbol table = *dynamic* mapping
  Function may change over time

Elementary implementations

Unordered (linked) list:

- Fast put
- $\Theta(n)$ time for get
- ⇝ Too slow to be useful

put('Hello', 5), put('a', 0)



Elementary implementations

Unordered (linked) list:

- 👍 Fast `put`
- 👎 $\Theta(n)$ time for `get`

$\rightsquigarrow$ Too slow to be useful

Sorted *linked* list:

- 👎 $\Theta(n)$ time for `put`
- 👎 $\Theta(n)$ time for `get`

$\rightsquigarrow$ Too slow to be useful

$\rightsquigarrow$ *Sorted order does not help us at all?!*

Binary search

*It does help . . . if we have a sorted **array**!*

Example: search for 69

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
ℓ							m								r

Binary search

It does help . . . if we have a sorted ***array***!

$63 < 69$

Example: search for 69

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
ℓ							m								r

$80 > 69$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ			m				r

Binary search

*It does help . . . if we have a sorted **array**!*

Example: search for 69

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
ℓ							m								r

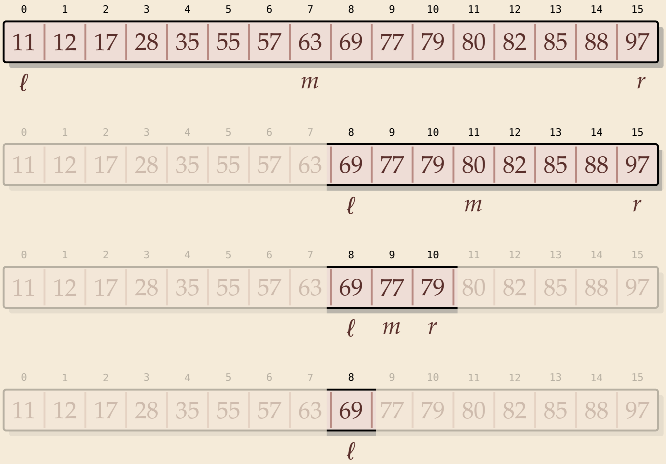
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ			m				r

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ	m	r					

Binary search

*It does help . . . if we have a sorted **array**!*

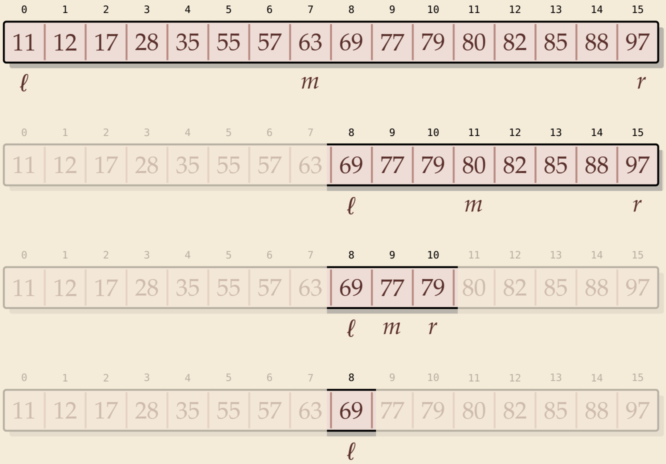
Example: search for 69



Binary search

It does help . . . if we have a sorted ***array***!

Example: search for 69



Binary search:

- halve remaining list in each step (±1)

⇝ $\le \lfloor \lg n \rfloor + 1$ cmps in the worst case

⚠ needs random access!

2.6 Binary Search Trees

Clicker Question

What is a binary search tree (tree in symmetric order)?

- **A** A tree in which every node has exactly 2 children.
- **B** A tree where all keys in the left subtree are smaller than the key at the root and all keys in the right subtree are bigger than the key at the root.
- **C** A tree where all keys in the left subtree and right subtree are bigger than the key at the root. = heap
- **D** A tree that is stored in the heap-area of the memory.

→ sli.do/comp526

Clicker Question

What is a binary search tree (tree in symmetric order)?

A ~~A tree in which every node has exactly 2 children.~~

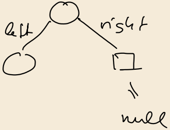
B A tree where all keys in the left subtree are smaller than the key at the root and all keys in the right subtree are bigger than the key at the root. ✓

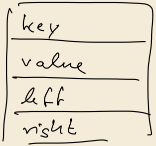
C ~~A tree where all keys in the left subtree and right subtree are bigger than the key at the root.~~

D ~~A tree that is stored in the heap area of the memory.~~

→ sli.do/comp526

Binary search trees



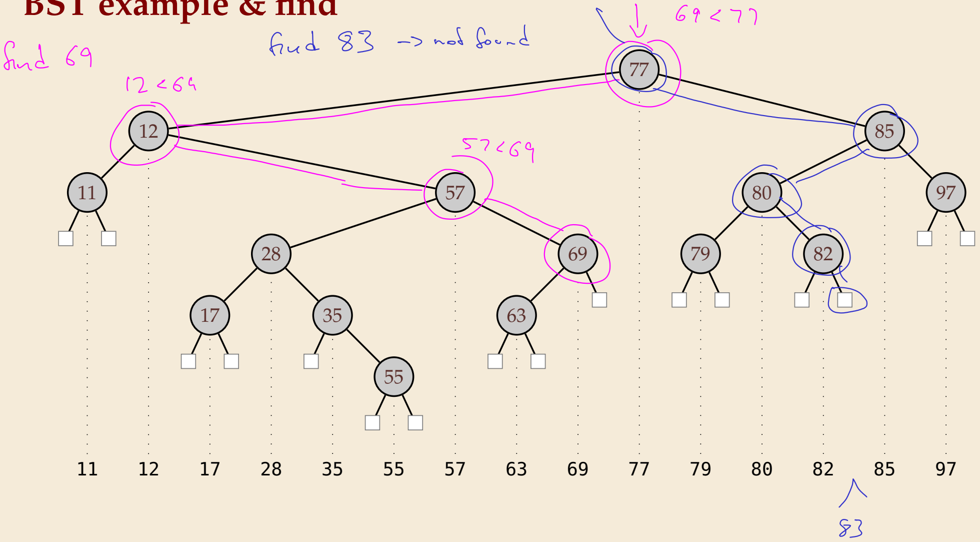


Binary search trees (BSTs) $\approx$ dynamic sorted array

- binary tree
  - Each node has left and right child
  - Either can be empty (`null`)
- Keys satisfy ***search-tree property***

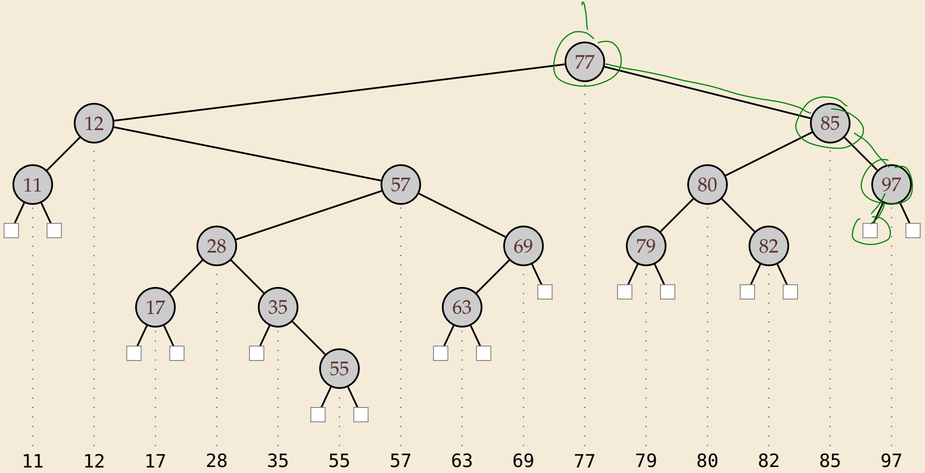
> all keys in left subtree $\le$ root key $\le$ all keys in right subtree

BST example & find



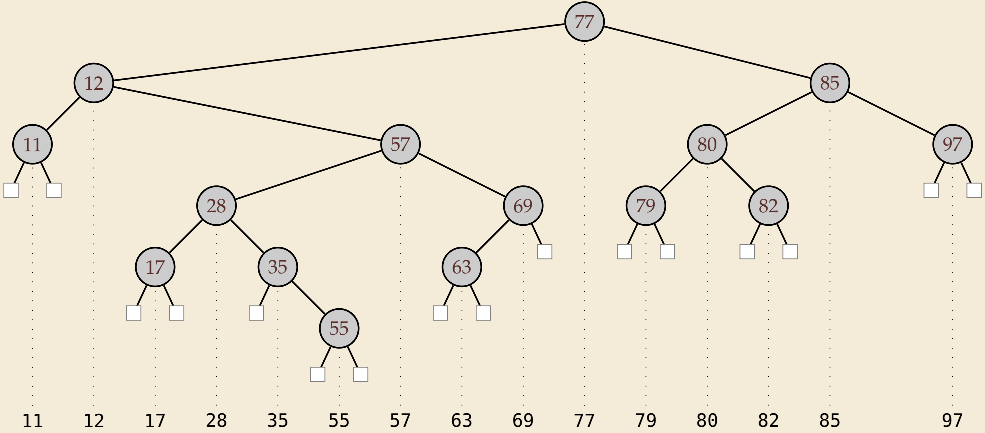
BST insert

Example: Insert 88



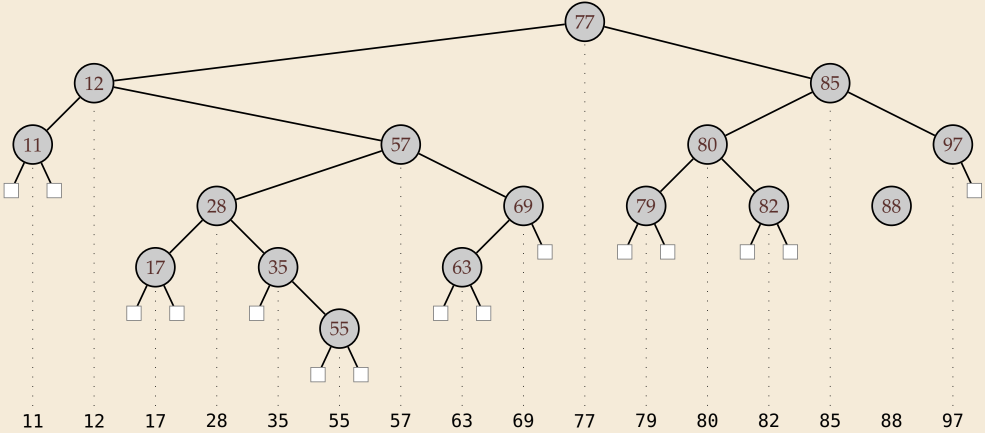
BST insert

Example: Insert 88



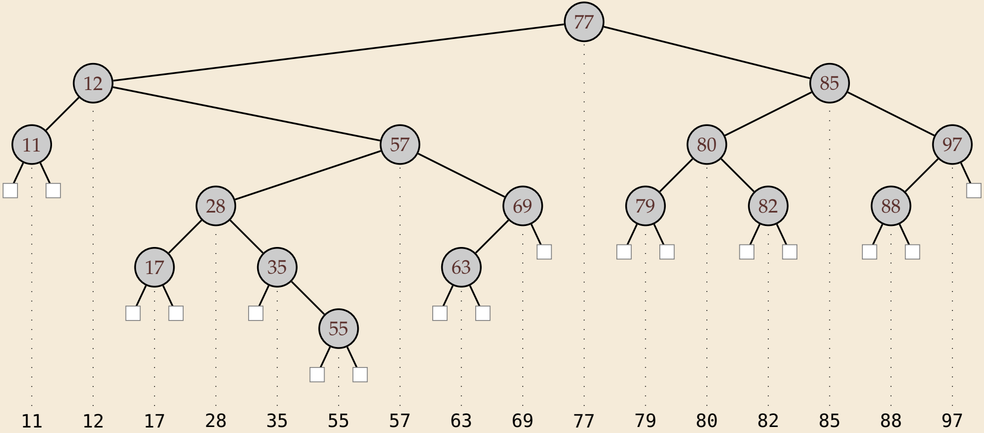
BST insert

Example: Insert 88



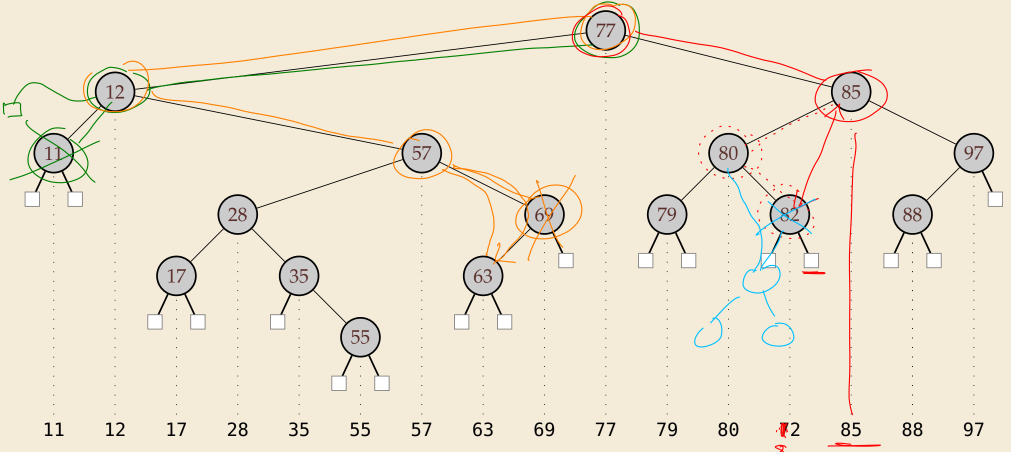
BST insert

Example: Insert 88



BST delete

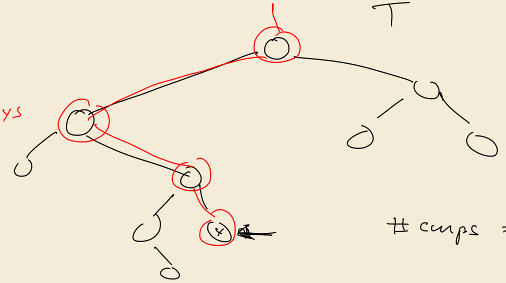
- Easy case: remove leaf, e. g., 11 ⇝ replace by `null`
- Medium case: remove unary, e. g., 69 ⇝ replace by unique child
- Hard case: remove binary, e. g., 85 ⇝ swap with predecessor, recurse



Analysis

- **Search:**

cmps of keys / (<, >, =)



$$\# \text{cmps} = \text{depth}(x) + 1 \leq \underbrace{\text{height}(T)}_{=\, h} + 1 = O(h)$$

- **Insert:**

  ① search $O(h)$

  ② new node + change pointers $O(1)$

  $\Big\}\ O(h)$

- **Delete:**

  ① search

  ② predecessor $\Big\}\ O(h)$

  ③ moving pointers $O(1)$

  $\Big\}\ O(h)$

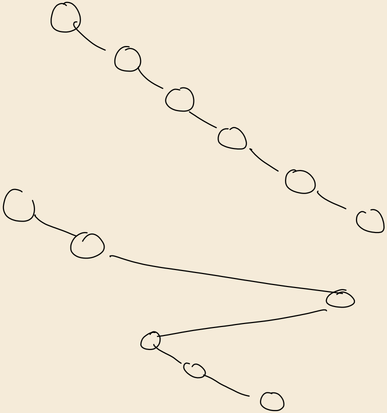
BST summary

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(nh)$
`put(`k`,`v`)`	$O(h)$
`get(`k`)`	$O(h)$
`delete(`k`)`	$O(h)$
`contains(`k`)`	$O(h)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

What is the height of a BST?

Worst Case:

- $h = n - 1 = \Theta(n)$



What is the height of a BST?

Worst Case:

- $h = n - 1 = \Theta(n)$

Average Case:

- Assumption: insertions come in random order
  no deletions

$\rightsquigarrow$ $h = \Theta(\log n)$ in expectation

even "with high probability":
$\forall d\ \exists c : \Pr[h \geq c \lg(n)] \leq n^{-d}$

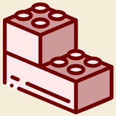
remember: any binary tree on n nodes
must have $\geq \lfloor \log_2(n) \rfloor$ height

2.7 Ordered Symbol Tables

Ordered symbol tables

- `min()`, `max()`
  Return the smallest resp. largest key in the ST
- `floor(`x`)`, $\lfloor x \rfloor = \mathbb{Z}$`.floor(`$x$`)`
  Return largest key k in ST with $k \le x$.
- `ceiling(`x`)`
  Return smallest key k in ST with $k \ge x$.
- `rank(`x`)`
  Return the number of keys k in ST $k < x$.

  select(rank(x)) = x
  x ∈ ST
- `select(`i`)`
  Return the ith smallest key in ST (zero-based, i. e., $i \in [0..n)$)

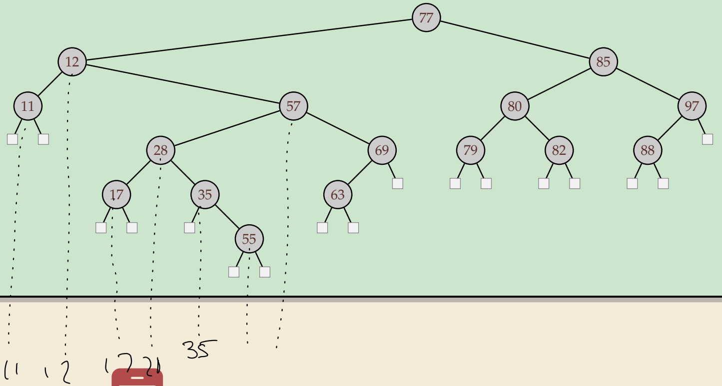


With select, we can simulate access as in a truly dynamic array!.

(Might not need any keys at all then!)

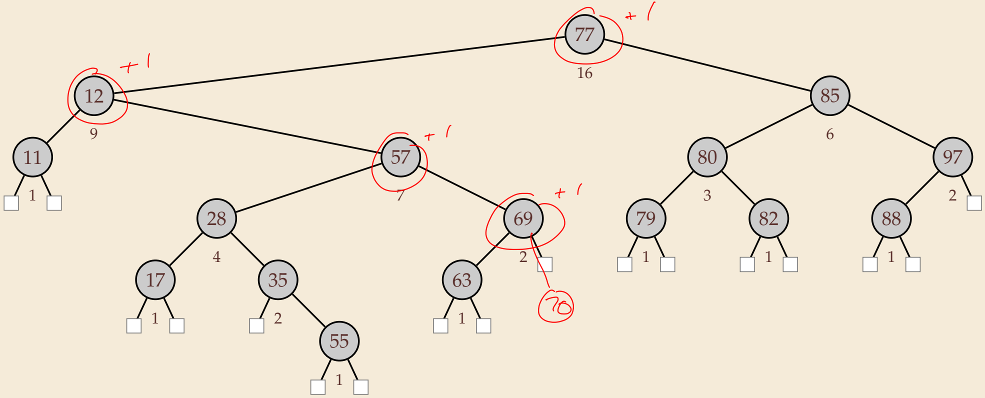
Clicker Question

In the BST below, what would `rank(35)` return?

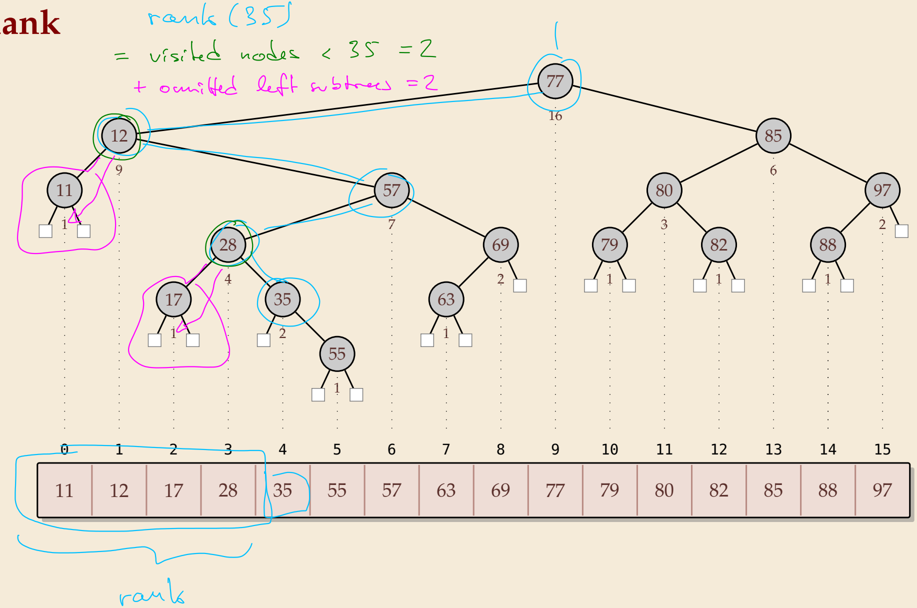


→ `sli.do/comp526`

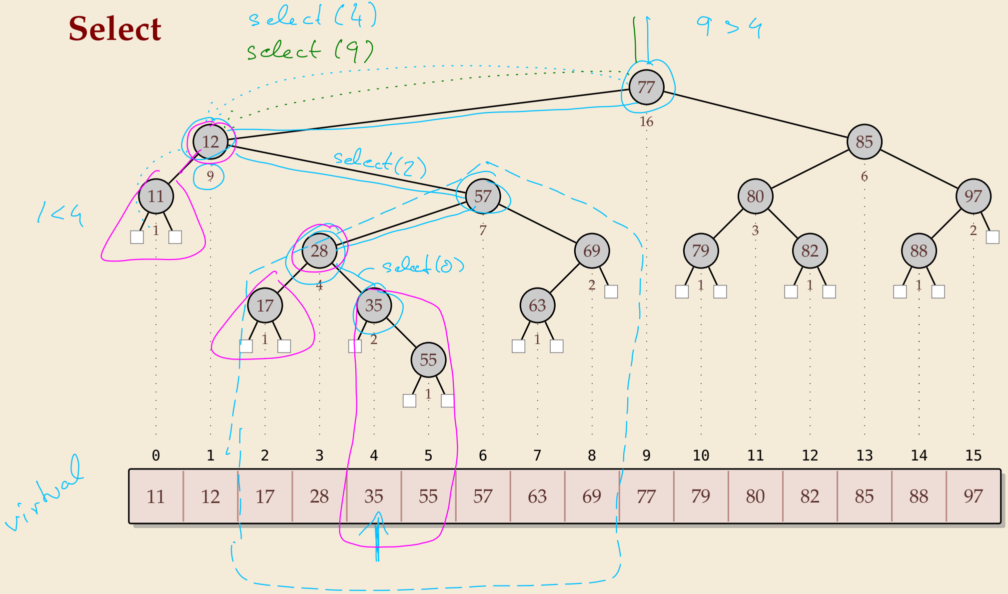
Augmented BSTs



Rank


rank (35)
= visited nodes < 35 = 2
+ omitted left subtrees = 2
77
16
12
9
85
6
11
1
57
7
80
3
97
2
28
4
69
2
79
1
82
1
88
1
17
1
35
2
63
1
55
1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
11 12 17 28 35 55 57 63 69 77 79 80 82 85 88 97
rank

Select



select (4)
select (9)
9 > 4
select(2)
select(8)
1<4
virtual
77
16
12
9
85
6
11
1
57
7
80
3
97
2
28
4
69
2
79
1
82
1
88
1
17
1
35
2
63
1
55
1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
11 12 17 28 35 55 57 63 69 77 79 80 82 85 88 97

Why store subtree sizes?

- Note that in an augmented BST, each node stores the **size of its subtree**.
- . . . why not directly store the **rank**? Would make rank/select much simpler!

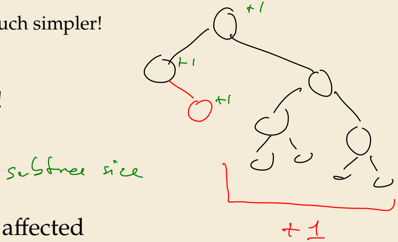
Why store subtree sizes?

- Note that in an augmented BST, each node stores the **size of its subtree**.
- . . . why not directly store the **rank**? Would make rank/select much simpler!

- Problem: Single insertion/deletion can change *all* node ranks!

⇝ Cannot efficiently maintain node ranks.

👍 Subtree sizes only change along search path ⇝ $O(h)$ nodes affected



2.8 Balanced BSTs

Clicker Question

What ways of maintaining a **balanced** binary search tree do you know?
Write “none” if you have not seen balanced BSTs before.

→ *sli.do/comp526*

Balanced BSTs

Balanced binary search trees:

- imposes shape invariant that guarantees $O(\log n)$ height
- adds rules to restore invariant after updates

Balanced BSTs

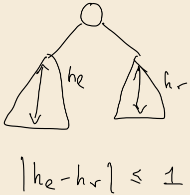
Balanced binary search trees:

- imposes shape invariant that guarantees $O(\log n)$ height
- adds rules to restore invariant after updates
- many examples known
  - *AVL trees* (height-balanced trees)
  - *red-black trees*
  - *weight-balanced trees* ($\mathrm{BB}[\alpha]$ trees)
  - ...

Balanced BSTs

Balanced binary search trees:

- imposes shape invariant that guarantees $O(\log n)$ height
- adds rules to restore invariant after updates
- many examples known
  - *AVL trees* (height-balanced trees)
  - *red-black trees*
  - *weight-balanced trees* ($BB[\alpha]$ trees)
  - ...



Other options:

- **amortization:** *splay trees, scapegoat trees*
- **randomization:** *randomized BSTs, treaps, skip lists*

I'd love to talk more about all of these ...
(Maybe another time)

BSTs vs. Heaps

Balanced binary search tree

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`contains(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()` / `max()`	$O(\log n) \rightsquigarrow O(1)$
`floor(`x`)`	$O(\log n)$
`ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$

Binary heaps

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$O(\log n)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

BSTs vs. Heaps

Balanced binary search tree

Operation	**Running Time**
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`contains(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()` / `max()`	$O(\log n) \rightsquigarrow O(1)$
`floor(`x`)`	$O(\log n)$
`ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$

Binary heaps

Operation	**Running Time**
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$O(\log n)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

- apart from faster `construct`, BSTs always as good as binary heaps

BSTs vs. Heaps

Balanced binary search tree

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`contains(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()` / `max()`	$O(\log n)$ $\rightsquigarrow$ $O(1)$
`floor(`x`)`	$O(\log n)$
`ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$

Binary heaps

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$O(\log n)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

- apart from faster `construct`, BSTs always as good as binary heaps
- MaxPQ abstraction still helpful

BSTs vs. Heaps

Balanced binary search tree

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`contains(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()` / `max()`	$O(\log n)$ $\rightsquigarrow$ $O(1)$
`floor(`x`)`	$O(\log n)$
`ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$

~~**Binary heaps**~~ *Strict Fibonacci heaps*

Operation	Running Time
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	~~$O(\log n)$~~ $O(1)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	~~$O(\log n)$~~ $O(1)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

- apart from faster `construct`, BSTs always as good as binary heaps
- MaxPQ abstraction still helpful
- and faster heaps exist!