

3

Fundamental Data Structures

28 October 2024

Prof. Dr. Sebastian Wild

Learning Outcomes

Unit 3: Fundamental Data Structures

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
7. Know basic *hashing strategies* and their performance characteristics

Outline

3 Fundamental Data Structures

- 3.1 Stacks & Queues
- 3.2 Resizable Arrays
- 3.3 Priority Queues & Binary Heaps
- 3.4 Operations on Binary Heaps
- 3.5 Symbol Tables
- 3.6 Binary Search Trees
- 3.7 Ordered Symbol Tables
- 3.8 Balanced BSTs
- 3.9 Hashing

Recap: The Random Access Machine

- ▶ Data structures make heavy use of pointers and dynamically allocated memory.
- ▶ Recall: Our RAM model supports
 - ▶ basic pseudocode (\approx simple Python code)
 - ▶ creating arrays of a fixed/known size.
 - ▶ creating instances (objects) of a known class.



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 \neq RAM:

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

3.1 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

≈ Java interface, Python ABCs
(with comments)

abstract base classes



VS.

data structures

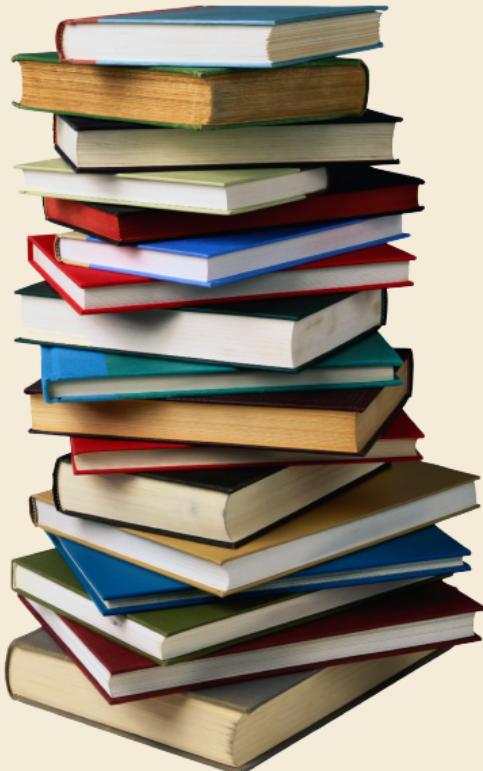
- ▶ specify exactly how data is represented
- ▶ algorithms for operations
- ▶ has concrete costs (space and running time)

≈ 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)

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.

Linked-list implementation for Stack

Invariants:

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

```
1 class Node
2     value
3     next
4
5 class Stack
6     top := null
7     procedure top()
8         return top.value
9     procedure push(x)
10        top := new Node(x, top)
11     procedure pop()
12        t := top()
13        top := top.next
14        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 .



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?

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 ↗ Bag
Implementation of Bag usually just a Stack or a Queue

3.2 Resizable Arrays

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 $0, 1, \dots, n - 1$;
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 .

~~~ 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!

## 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!

# 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!

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

distance to boundary      since  $n \leq C \leq 4n$

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

$$\text{here: } \leq 5m + 4 \cdot 0.6n = \Theta(m + n)$$

# Deamortized Resizable Arrays

What if we need  $O(1)$  worst case time?

- ▶ It's possible to *de-amortize* the resizing arrays solution!
- ▶ maintain 3 arrays:  $S$  (as before) and  $S_2$  and  $S_{1/2}$  of twice and half the size of  $S$
- ▶ write operations go to all 3 arrays
- ▶ upon resize, "shift" arrays up/down  $\rightsquigarrow S_2$  resp.  $S_{1/2}$  become new  $S$ 
  - ▶ allocate new array, but **delay filling it with elements**  $\leftarrow$  general strategy!
  - ▶ every insert or delete copies 2 slots from last resize
- $\rightsquigarrow$  by time for next resize, we have caught up and  $S_2$  resp.  $S_{1/2}$  ready to use

**Analysis:**

- ▶  $O(1)$  worst case time for read/write by index, push, and pop!
- ▶ up to 7 array accesses per operation
- ▶ up to  $7n$  space  $\leftarrow$  other time-space trade-offs possible

assuming memory allocation in  $O(1)$   $\rightsquigarrow$  needs to be uninitialized!

## Rabbit Hole: Can we do this more space-efficiently?

- ▶ It might appear as if every efficient implementation of a stack needs  $\Omega(n)$  extra space on top of space for storing the  $n$  elements in the stack.
- ▶ But this is not true!
- ▶ Can get operations in  $O(1)$  worst-case time with  $O(\sqrt{n})$  extra space at any time (!)
  - ▶ Maintain a collection of small arrays (plus header with pointers to them)
  - ▶ Clever choice of block sizes guarantees  
 $O(\sqrt{n})$  blocks of  $O(\sqrt{n})$  elements throughout and fast calculation of address for an index.  
imaginary “superblocks” of sizes  $2^k$ ,  $k = 0, 1, \dots, \lg n$   
 $k$ th superblock consists of  $2^{k/2}$  actual blocks of  $2^{k/2}$  elements each.
  - ▶  $O(\sqrt{n})$  extra space is best possible

*Resizable Arrays in Optimal Time and Space*

Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro & Robert Sedgewick

WADS 1999

## 3.3 Priority Queues & Binary Heaps

# Priority Queue ADT – min-oriented version

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

(Max-oriented) Priority Queue (MaxPQ):

▶ `construct( $A$ )`

Construct 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.



# 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?

Yes! *Binary heaps*.

### Array view

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

  
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  $\geq$  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  $A$

- ▶ 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

## 3.4 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.

## 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  $\rightsquigarrow$  root has depth 0
- ▶ How many nodes on first  $k$  full levels? 
$$\sum_{\ell=0}^k 2^\ell = 2^{k+1} - 1$$
 $\rightsquigarrow$  Height of binary heap:  $h = \min k \text{ s.t. } 2^{k+1} - 1 \geq n = \lfloor \lg(n) \rfloor$

## Analysis:

- ▶ **insert**: new element “swims” up  $\rightsquigarrow \leq h$  steps ( $h$  cmps)
- ▶ **delMax**: last element “sinks” down  $\rightsquigarrow \leq h$  steps ( $2h$  cmps)
- ▶ **construct** from  $n$  elements:

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

$$\begin{aligned} &\leq 1 \cdot h + 2 \cdot (h-1) + 4 \cdot (h-2) + \cdots + 2^\ell \cdot (h-\ell) + \cdots + 2^{h-1} \cdot 1 + 2^h \cdot 0 \\ &= \sum_{\ell=0}^h 2^\ell (h-\ell) = \sum_{i=0}^h \frac{2^h}{2^i} i = 2^h \sum_{i=0}^h \frac{i}{2^i} \leq 2 \cdot 2^h \leq 4n \end{aligned}$$

## Binary heap summary

| Operation                                    | Running Time |
|----------------------------------------------|--------------|
| <code>construct(<math>A[1..n]</math>)</code> | $O(n)$       |
| <code>max()</code>                           | $O(1)$       |
| <code>insert(<math>x, p</math>)</code>       | $O(\log n)$  |
| <code>delMax()</code>                        | $O(\log n)$  |
| <code>changeKey(<math>x, p'</math>)</code>   | $O(\log n)$  |
| <code>isEmpty()</code>                       | $O(1)$       |
| <code>size()</code>                          | $O(1)$       |

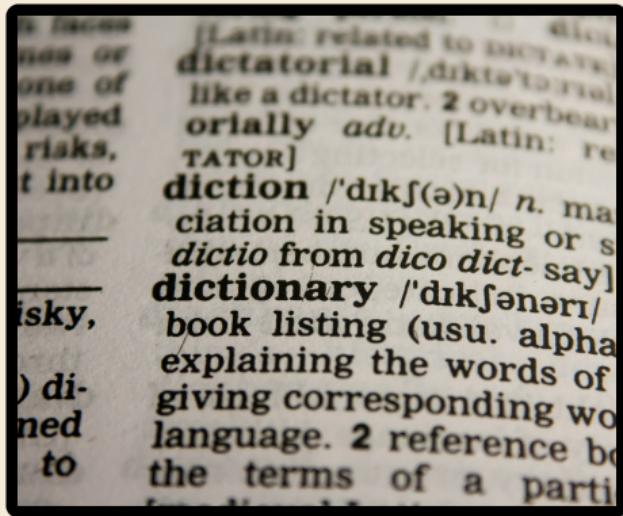
## 3.5 Symbol Tables

# 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) from 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

## Sorted *linked* list:

👎  $\Theta(n)$  time for put

👎  $\Theta(n)$  time for get

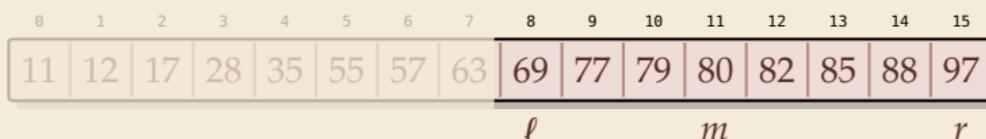
~~ Too slow to be useful

~~ *Sorted order does not help us at all?!*

# Binary search

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

Example: search for 69



Binary search:

- ▶ halve  $\nearrow \pm 1$  remaining list in each step
- $\rightsquigarrow \leq \lfloor \lg n \rfloor + 1$  cmps in the worst case



needs random access!

## 3.6 Binary Search Trees

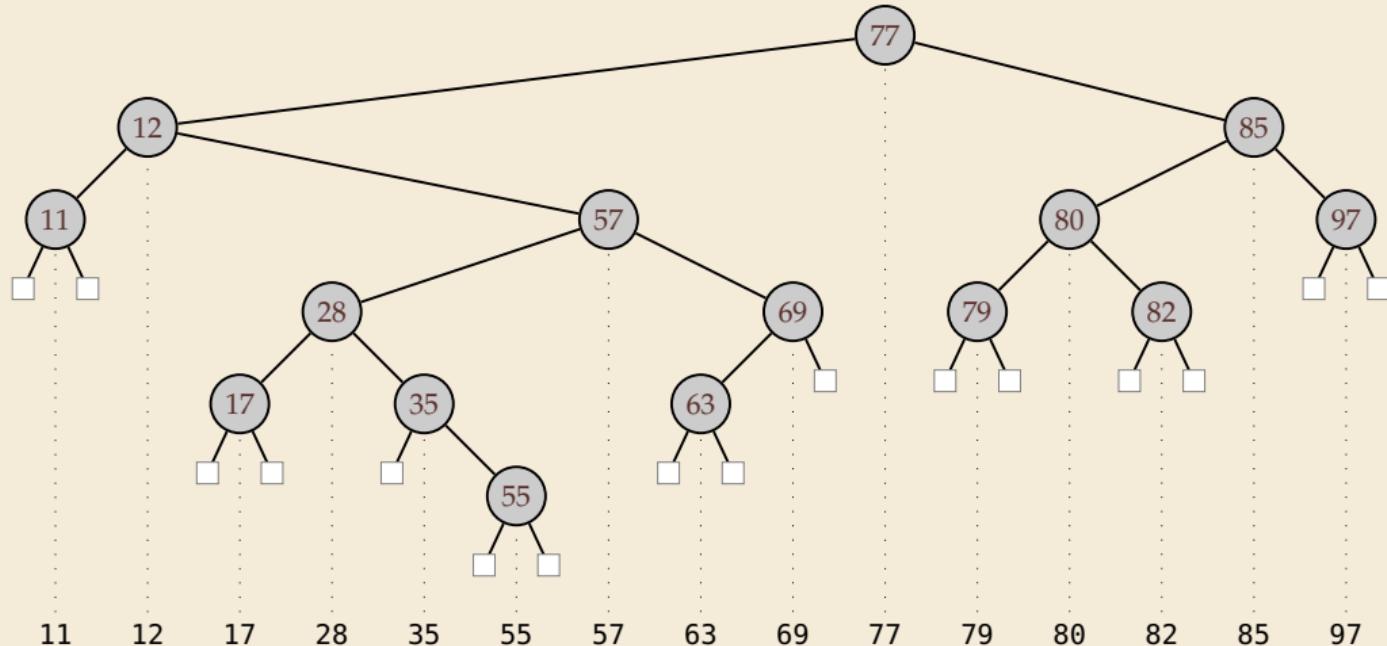
# Binary search trees

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*

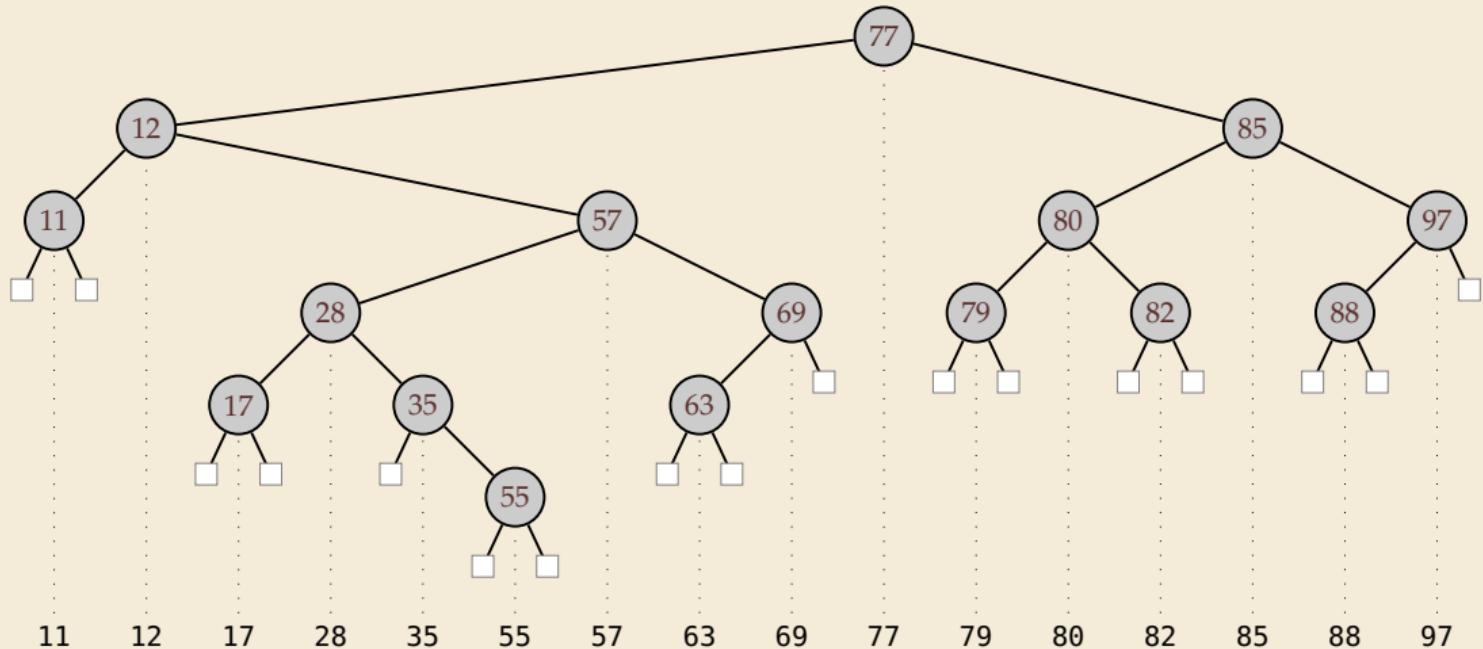
all keys in left subtree  $\leq$  root key  $\leq$  all keys in right subtree

## BST example & find



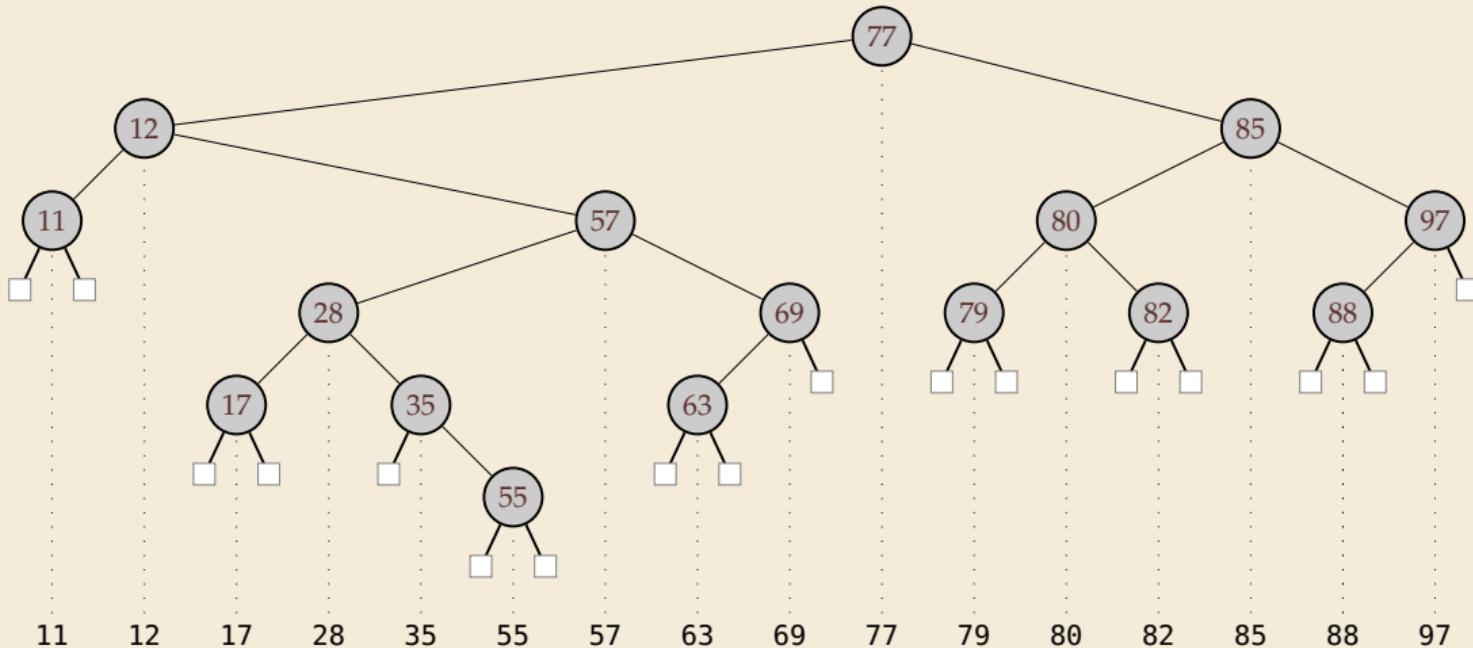
# BST insert

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:

► Insert:

► Delete:

## BST summary

| Operation                                    | Running Time |
|----------------------------------------------|--------------|
| <code>construct(<math>A[1..n]</math>)</code> | $O(nh)$      |
| <code>put(<math>k, v</math>)</code>          | $O(h)$       |
| <code>get(<math>k</math>)</code>             | $O(h)$       |
| <code>delete(<math>k</math>)</code>          | $O(h)$       |
| <code>contains(<math>k</math>)</code>        | $O(h)$       |
| <code>isEmpty()</code>                       | $O(1)$       |
| <code>size()</code>                          | $O(1)$       |

# What is the height of a BST?

Worst Case:

- ▶  $h = n - 1 = \Theta(n)$

Average Case:

- ▶ Assumption: insertions come in random order  
no deletions

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

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

## 3.7 Ordered Symbol Tables

# Ordered symbol tables

- ▶  $\min()$ ,  $\max()$

Return the smallest resp. largest key in the ST

- ▶  $\text{floor}(x)$ ,  $\lfloor x \rfloor = \mathbb{Z}.\text{floor}(x)$

Return largest key  $k$  in ST with  $k \leq x$ .

- ▶  $\text{ceiling}(x)$

Return smallest key  $k$  in ST with  $k \geq x$ .

- ▶  $\text{rank}(x)$

Return the number of keys  $k$  in ST  $k < x$ .

- ▶  $\text{select}(i)$

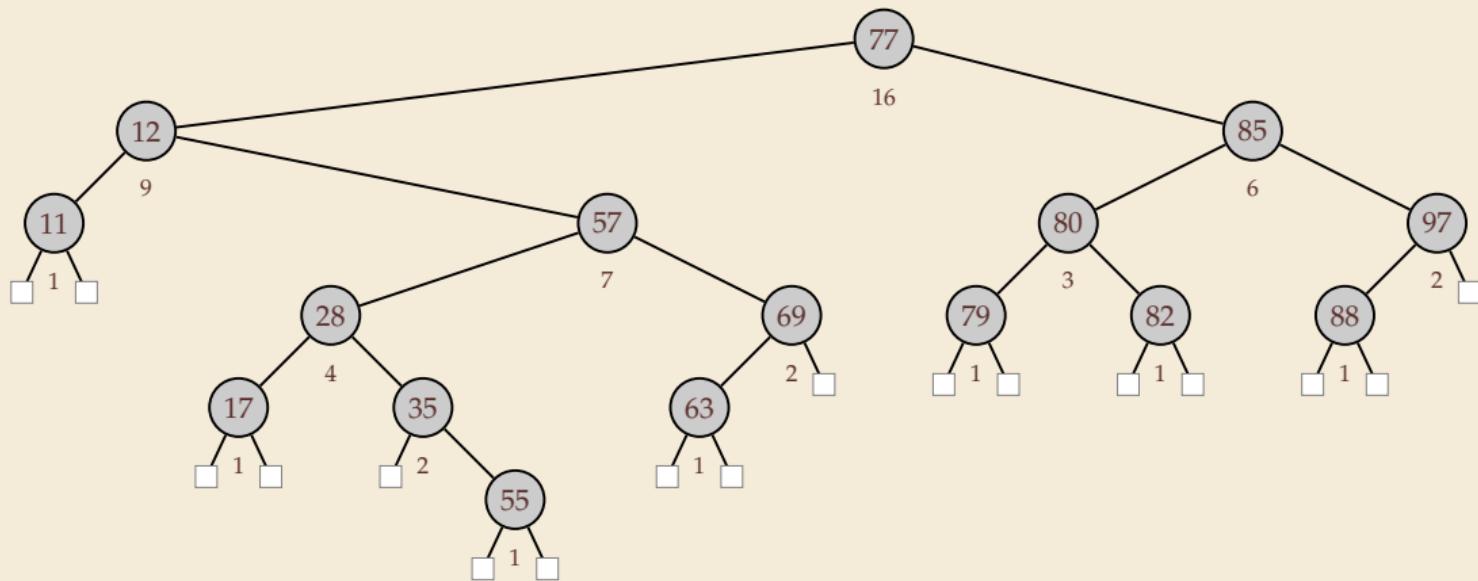
Return the  $i$ th smallest key in ST (zero-based, i. e.,  $i \in [0..n)$ )



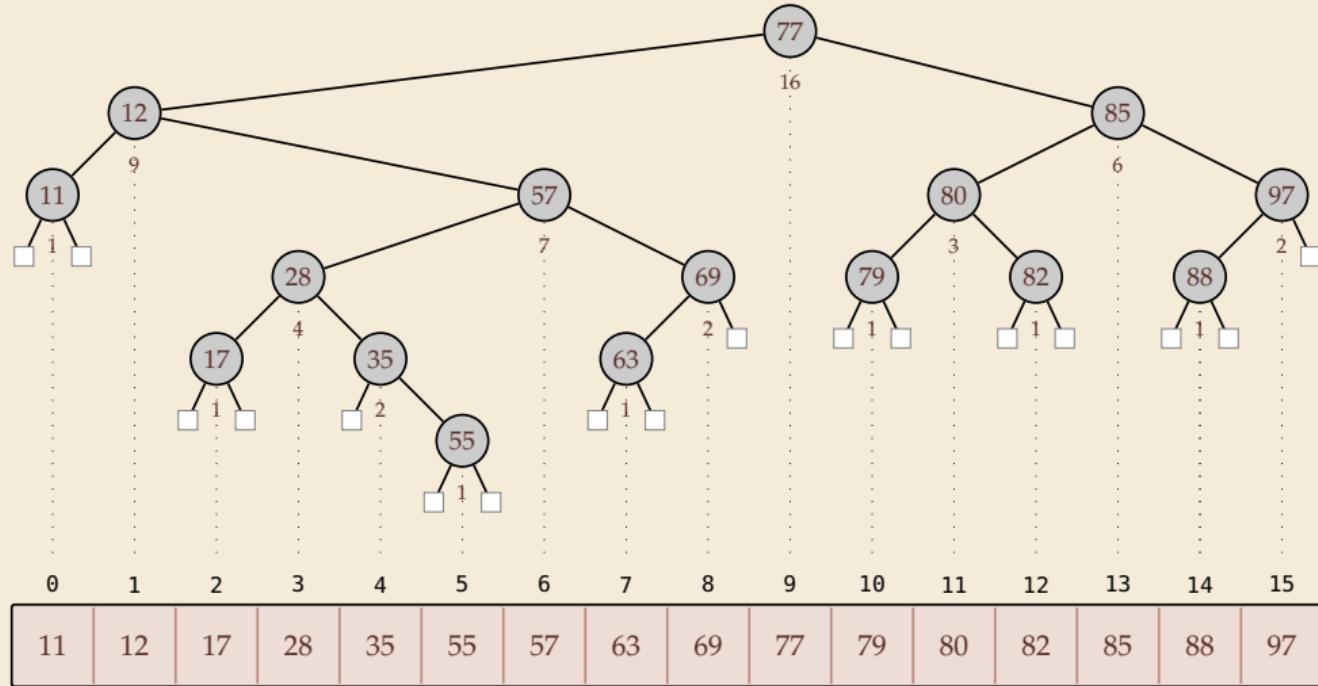
*With select, we can simulate access as in a truly dynamic array!.*

(Might not need any keys at all then!)

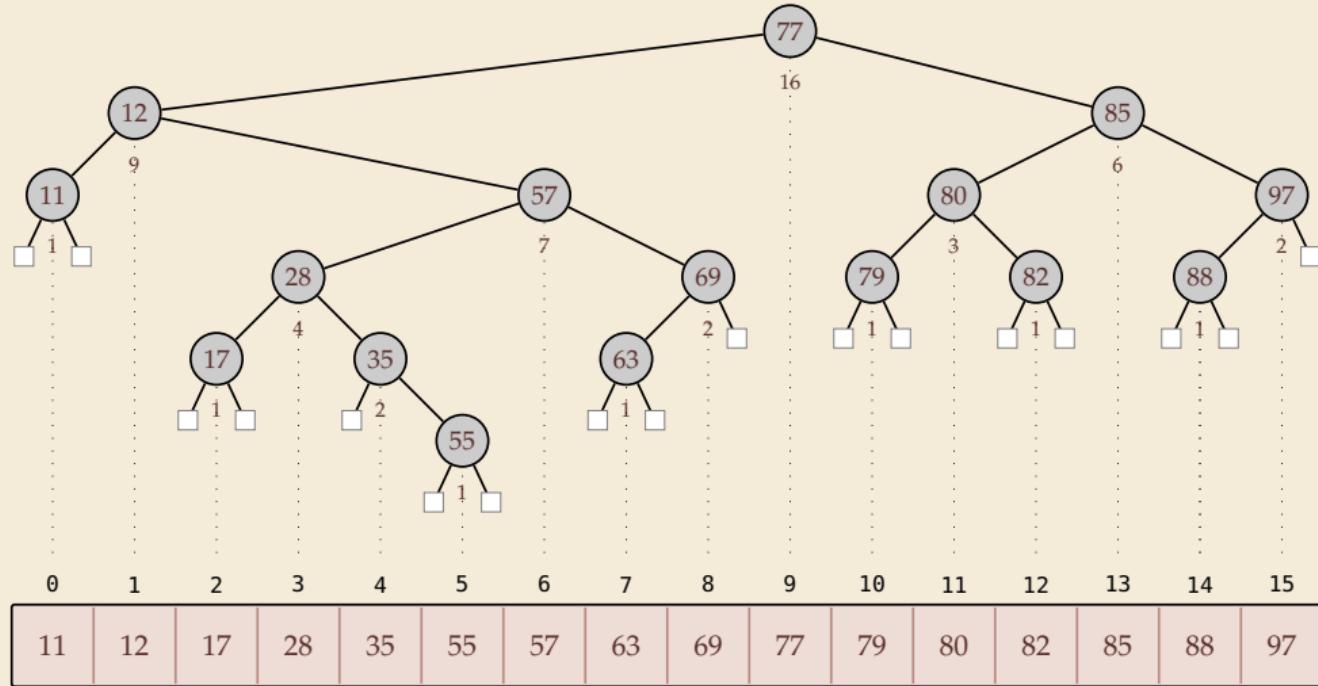
# Augmented BSTs



# Rank



# Select



## 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

## 3.8 Balanced BSTs

# 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* ( $\text{BB}[\alpha]$  trees)
  - ▶ ...

## Other options:

- ▶ **amortization:** *splay trees, scapegoat trees*  
*COLA (cache oblivious lookahead array)*
- ▶ **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                      |
|----------------------------------------------|-----------------------------------|
| <code>construct(<math>A[1..n]</math>)</code> | $O(n \log n)$                     |
| <code>put(<math>k, v</math>)</code>          | $O(\log n)$                       |
| <code>get(<math>k</math>)</code>             | $O(\log n)$                       |
| <code>delete(<math>k</math>)</code>          | $O(\log n)$                       |
| <code>contains(<math>k</math>)</code>        | $O(\log n)$                       |
| <code>isEmpty()</code>                       | $O(1)$                            |
| <code>size()</code>                          | $O(1)$                            |
| <code>min() / max()</code>                   | $O(\log n) \rightsquigarrow O(1)$ |
| <code>floor(<math>x</math>)</code>           | $O(\log n)$                       |
| <code>ceiling(<math>x</math>)</code>         | $O(\log n)$                       |
| <code>rank(<math>x</math>)</code>            | $O(\log n)$                       |
| <code>select(<math>i</math>)</code>          | $O(\log n)$                       |

## ~~Binary heaps~~ Strict Fibonacci heaps

| Operation                                    | Running Time                             |
|----------------------------------------------|------------------------------------------|
| <code>construct(<math>A[1..n]</math>)</code> | $O(n)$                                   |
| <code>insert(<math>x, p</math>)</code>       | <del><math>O(\log n)</math></del> $O(1)$ |
| <code>delMax()</code>                        | $O(\log n)$                              |
| <code>changeKey(<math>x, p'</math>)</code>   | <del><math>O(\log n)</math></del> $O(1)$ |
| <code>max()</code>                           | $O(1)$                                   |
| <code>isEmpty()</code>                       | $O(1)$                                   |
| <code>size()</code>                          | $O(1)$                                   |

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

## 3.9 Hashing

## Lower bound for search

The fastest implementations of the ordered symbol table ADT require  $\Theta(\log n)$  time to search among  $n$  items. Is this the best possible?

**Theorem:** In the comparison model (on the keys),  
 $\Omega(\log n)$  comparisons are required to search a size- $n$  dictionary.

**Proof:** Similar to lower bound for sorting (see Unit 4).

Any algorithm defines a binary decision tree with  
comparisons at the nodes and actions at the leaves.

There are at least  $n + 1$  different actions (return an item, or “not found”).

So there are  $\Omega(n)$  leaves, and therefore the height is  $\Omega(\log n)$ . □

*What if we don't need the ordered symbol table operations?*

~~ Focus on symbol table operations: get, put, contains, delete

# Symbol Table without Sorting

- ▶ key idea in hashing: everything is ultimately an integer, or can be turned into one!
  - ~~ hash function  $h : U \rightarrow [0..m]$ 
    - ▶ maps elements from universe  $U$  to integers
    - ▶  $h(x)$  used as index in a hash table  $T[0..m]$
  - ~~ if  $h$  is quick to compute and all stored elements hash to different indices get, put, contains, delete become simple array operations!
  - ~~ symbol table with  $O(1)$  time per operation
- ⚡ Generally hash function  $h$  is not injective, so many keys can map to the same integer.
- ▶ We get *collisions*: we want to insert  $(k, v)$  into the table, but  $T[h(k)]$  is already occupied.

# Collisions

- ▶ Two basic strategies to deal with collisions:
    - ▶ *Buckets/Chaining*: Allow multiple items at each table location
    - ▶ *Open addressing*: Allow each item to go into multiple locations
  - ▶ We evaluate strategies by the average cost of get, put, delete in terms of  $n$ ,  $m$ , and/or the *load factor*  $\alpha = n/m$ .
  - ~~> Might have to rebuild the whole hash table and change the value of  $m$  when the load factor gets too large or too small.
    - ▶ This is called *rehashing*, and costs  $\Theta(m + n)$ .
    - ▶ alternative: dynamic hashing
- ... For details of these strategies, see *Algorithms and Data Structures*.

# Comparison of Classic Hashing Schemes

| Hash table design       | Search hit                                             | Search miss                                                  | Insert | Space   | good $\alpha$    |
|-------------------------|--------------------------------------------------------|--------------------------------------------------------------|--------|---------|------------------|
| Separate Chaining       | $\sim \frac{1}{2}\alpha$                               | $\sim \alpha$                                                | = miss | $n + m$ | $\approx 2$      |
| Linear Probing          | $\sim \frac{1}{2}(1 + \frac{1}{1-\alpha})$             | $\sim \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$               | = miss | $m$     | $\leq 0.5$       |
| Quadratic Probing       | $\sim 1 + \ln(\frac{1}{1-\alpha}) - \frac{1}{2}\alpha$ | $\sim \frac{1}{1-\alpha} - \alpha + \ln(\frac{1}{1-\alpha})$ | = miss | $m$     | $\leq 0.7$       |
| Robin Hood Hashing      | $O(1)$                                                 | $O(1)$                                                       | = miss | $m$     | $\leq 1 (=any!)$ |
| $d$ -way Cuckoo Hashing | $\leq d$ worst case                                    | $\leq d$ worst case                                          | amort. | $m$     | $< c_d$          |

- ▶ Assumption: uniform hashing (all  $m^n$  hash sequences equally likely)
- ▶ Cost: expected # (equality) comparisons
- ▶ Space usage in words on top of space for items

More improvements possible with word-RAM bitwise tricks  $\rightsquigarrow$  *Advanced Data Structures*

# Hashing vs. Balanced Search Trees

## Advantages of Balanced Search Trees

- ▶  $O(\log n)$  worst-case operation cost
- ▶ Does not require any assumptions, special functions, or known properties of input distribution
- ▶ Predictable (and often smaller) space usage (exactly  $n$  nodes)
- ▶ Never need to rebuild the entire structure
- ▶ supports ordered dictionary operations (rank, select etc.)

## Advantages of Hash Tables

- ▶  $O(1)$  operations (if hashes well-spread and load factor small)
- ▶ We can choose space-time tradeoff via load factor
- ▶ Cuckoo hashing achieves  $O(1)$  worst-case for search & delete