3 Fundamental Data Structures

28 October 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2024-10-28 15:42`

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 high-level idea of basic *hashing strategies* and their performance characteristics

Outline

3 Fundamental Data Structures

Clicker Question

What's the running time (on our word-RAM model with word size w) of this Java instruction?

```
Object[] A = new Object[n];
```

- A 1
- B $\Theta(1)$
- C $\Theta(\log n)$
- D $\Theta(w)$
- E $\Theta(n/w)$
- F $\Theta(n)$
- G $\Theta(n \log n)$
- H $\Theta(nw)$
- I $\Theta(n^2)$

→ sli.do/cs566

Clicker Question

What's the running time (on our word-RAM model with word size w) of this Java instruction?

```
Object[] A = new Object[n];
```

initialization w/ null

- A ~~1~~
- B $\cancel{\Theta(1)}$
- C $\cancel{\Theta(\log n)}$
- D $\cancel{\Theta(w)}$
- E $\Theta(n/w)$ ✓
- F $\Theta(n)$ ✓
- G $\cancel{\Theta(n \log n)}$
- H $\cancel{\Theta(nw)}$
- I $\cancel{\Theta(n^2)}$

→ sli.do/cs566

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/Java code)
  - creating arrays of a fixed/known size.
  - creating instances (objects) of a known class.

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/Java code)
  - creating arrays of a fixed/known size.
  - creating instances (objects) of a known class.

no predefined capacity!

Python abstracts this away!
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!

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

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)

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~~ 4)

Abstract Data Types

abstract data type (ADT)

- list of supported operatio
- **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 implemen
- ⇝ **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/cs566

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/cs566

Stacks

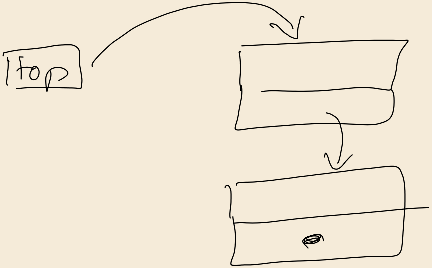
Stack ADT

- `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)




```
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

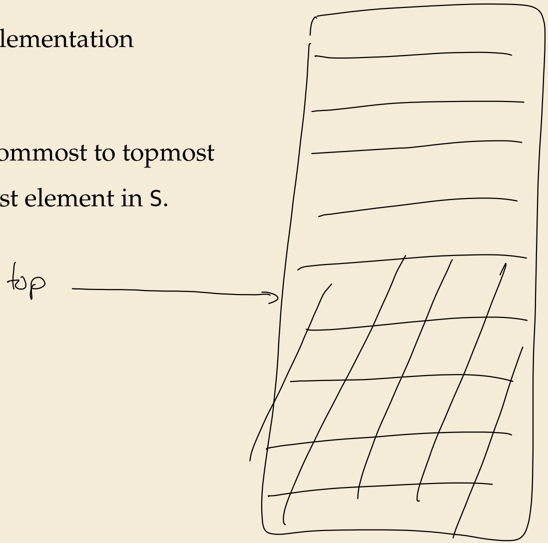
  Can we avoid extra space for pointers?

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.



Array-based implementation for Stack

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

Invariants:

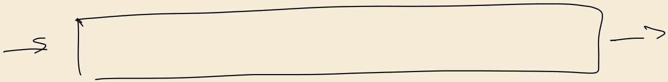
- maintain array S of elements, from bottommost to topmost
- maintain index *top* of position of topmost element in `S`.

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?

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

3.2 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!

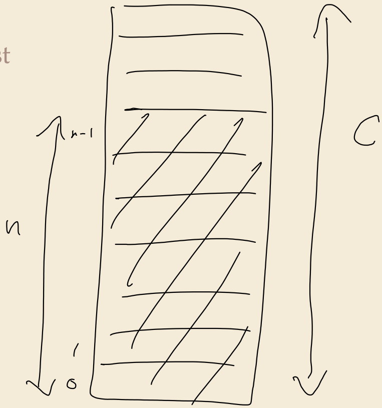
Doubling trick

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 $\frac{1}{4}C \leq n \leq C$

$\rightsquigarrow$ 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 $\frac{1}{4}C \leq n \leq C$

$\rightsquigarrow$ can always push more elements!

for n elements we use $\Theta(n)$ space

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.

$\rightsquigarrow$ ***"Resizing Arrays"***

an implementation technique, not an ADT!

Clicker Question

Which of the following statements about resizable array that currently stores n elements is always 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/cs566

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!

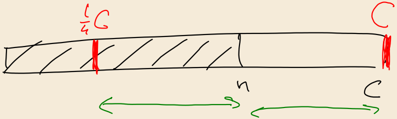
Amortized analysis w/ potential function

c_i = actual cost of operation i

Φ_i = potential after operation i

a_i = amortized cost of operation i

$:= c_i - 4(\Phi_i - \Phi_{i-1})$



Φ_i = distance to dangerous boundary $= \min\{C-n,\ n-\frac{1}{4}C\}$

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 $n \le C \le 4n$

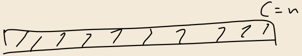
Formally: consider "credits/potential" $\Phi = \min\{n - \frac{1}{4}C, C - n\} \in [0, 0.6n]$

- 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

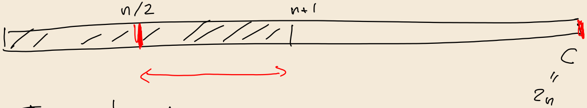
$\rightsquigarrow$ **sequence** of m operations: total actual cost $\le$ total amortized cost $+$ final credits

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

copying push:

before 

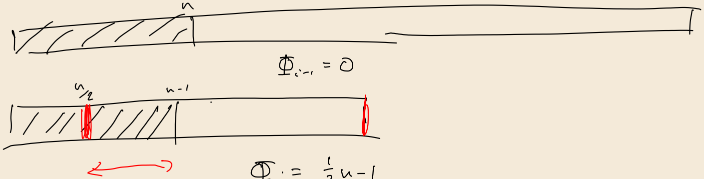
$\Phi_{i-1} = 0$

after 

$$\Phi_i = \frac{1}{2}n + 1$$

$$a_i = 2n+1 - 4\left(\frac{1}{2}n+1\right) = 1-4 = -3 \leq 5$$

copying pop



$\Phi_{i-1} = 0$

$\Phi_i = \frac{1}{2}n - 1$

$$a_i = 2n+1 - 4\left(\frac{1}{2}n-1\right) = 5$$

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

$$= \sum_{i=1}^{m} c_i - 4 \underbrace{\sum_{i=1}^{m} (\Phi_i - \Phi_{i-1})}$$

telescoping sum

$$= \quad '' \quad - 4(\Phi_m - \Phi_0)$$

$$\sum_{i=1}^{4} \Phi_i - \Phi_{i-1}$$

$$= (\Phi_4 - \cancel{\Phi_3})$$
$$\cancel{+ (\Phi_3} - \cancel{\Phi_2})$$
$$\cancel{+ (\Phi_2} - \cancel{\Phi_1})$$
$$\cancel{+ (\Phi_1} - \Phi_0)$$

$$\hookrightarrow \sum_{i=1}^{m} c_i \leq 5m + 4(\Phi_m - \Phi_0)$$

$$\leq 5m + 4 \cdot 0.6n - 0$$

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/cs566

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~~ $\cancel{2n}$~~.~~
- 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 $O(n + m)$. ✓ $\Rightarrow$ "O(1) amortized per operation"
- D ~~Inserting and deleting any element takes~~ $\cancel{O(1)}$ ~~amortized time.~~

→ sli.do/cs566

Deamortized Resizable Arrays

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

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

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** 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

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** 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:

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

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

other time-space trade-offs possible

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.

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!

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, $k = 0, 1, \ldots, \lg n$
    kth superblock consists of $2^{k/2}$ actual blocks of $2^{k/2}$ elements each.
  - $O(\sqrt{n})$ extra space is best possible

$\notin$ exam

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

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/cs566

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*~~*de**creasing*** 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`

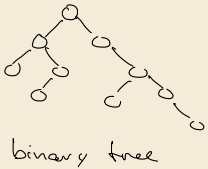
PQ implementations

Elementary implementations

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

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

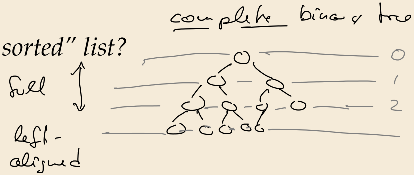
PQ implementations



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?



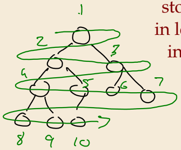
Yes! *Binary heaps.*

Array view

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

$\equiv$

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



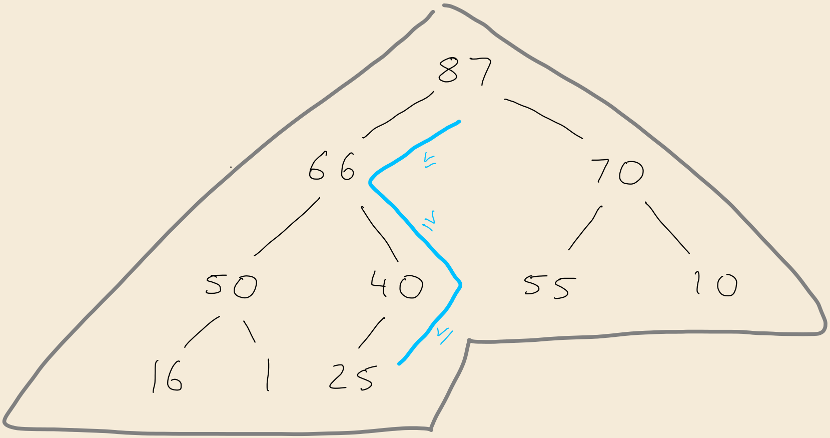
Tree view

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

all but last level full
last level flush left

parent $\geq$ children

Binary heap example



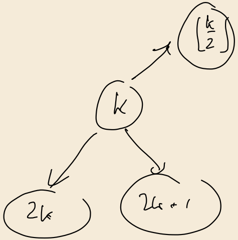
Why heap-shaped trees?

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 ⚠ Recall: nodes at indices $[\mathbf{1}..n]$

  - 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 ⚠ Recall: nodes at indices $[\mathbf{1}..n]$

  - parent at $\lfloor k/2 \rfloor$ (for $k \ge 2$)
  - 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?

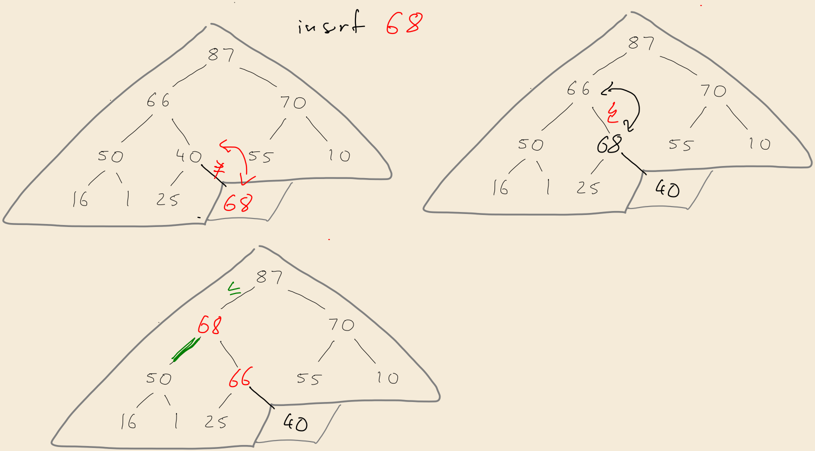
- A ~~A tree in which every node has exactly 2 children.~~
- B ~~A tree where all keys in the left subtree are smaller than the key at the root and all keys in the right subtree are bigger than the key at the root.~~
- C A tree where all keys in the left subtree and right subtree are smaller than the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

→ sli.do/cs566

3.4 Operations on Binary Heaps

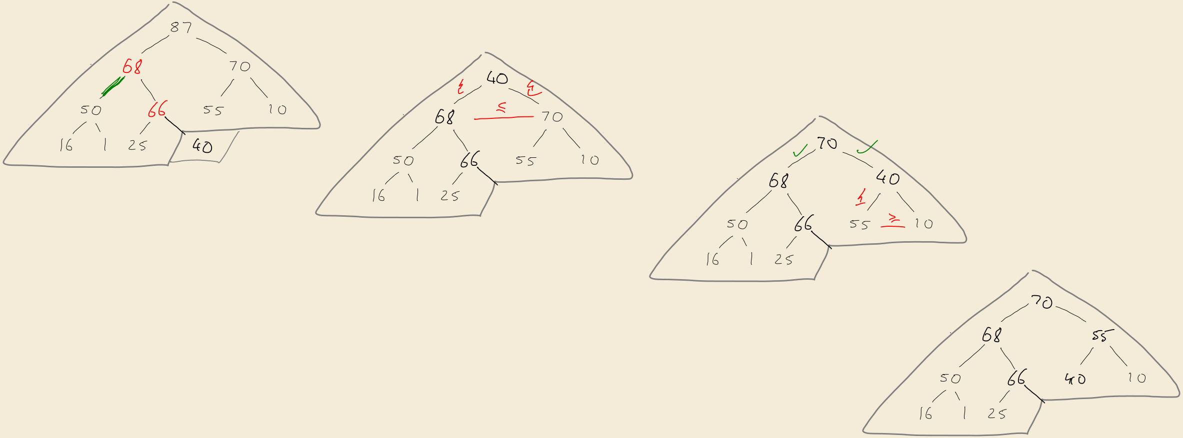
Insert

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



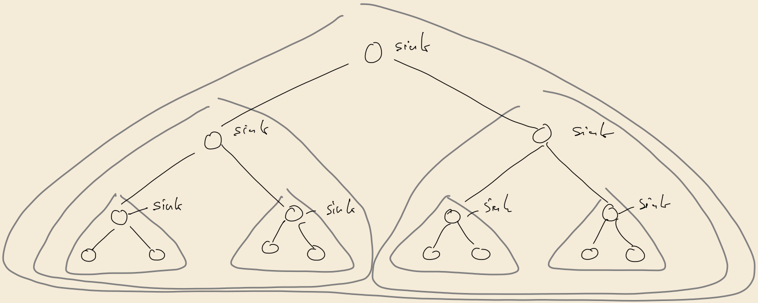
Delete Max

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? $\displaystyle\sum_{\ell=0}^{k} 2^{\ell} = 2^{k+1} - 1$

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

Analysis

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()`	$O(1)$

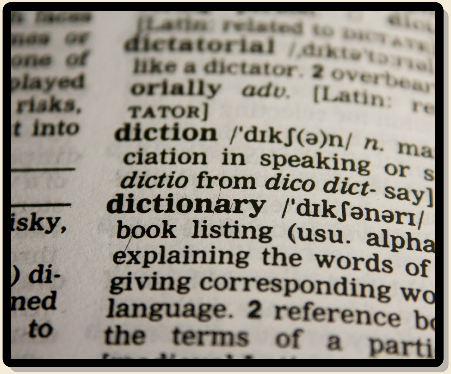
3.5 Symbol Tables

Symbol table ADT

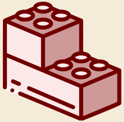
Java: `java.util.Map<K,V>`

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) 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.

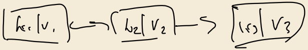
(Every programming library has a symbol table implementation.)

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

Elementary implementations

Unordered (linked) list:

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

Sorted *linked* list: assumes keys are ordered

- $\Theta(n)$ 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

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

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

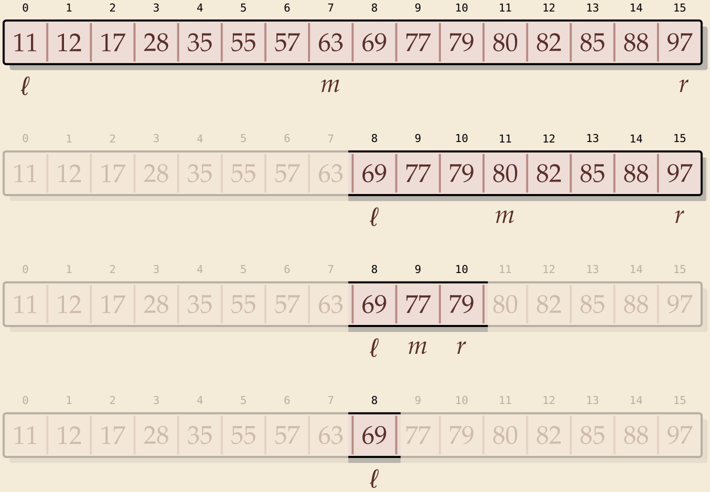
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	12	17	28	35	55	57	63	69	77	79	80	82	85	88	97
								ℓ			m				r

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



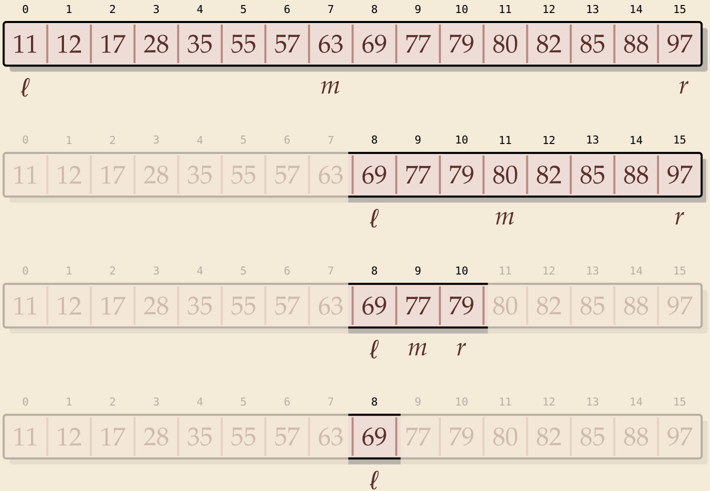
Binary search

unsuccessful search (search miss)

successful search (search hit)
↳ best case 1

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

Example: search for 69. 1



Binary search:

- halve remaining (± 1) list in each step

$\rightsquigarrow$ $\leq \lfloor \underbrace{\lg}_{=\log_2} n \rfloor + 1$ cmps ($<, =, >$) in the worst case

⚠ needs random access!

3.6 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.
- **D** A tree that is stored in the heap-area of the memory.

→ sli.do/cs566

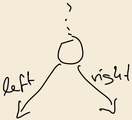
Clicker Question

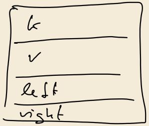
What is a binary search tree (tree in symmetric order)?

- 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/cs566

Binary search trees





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

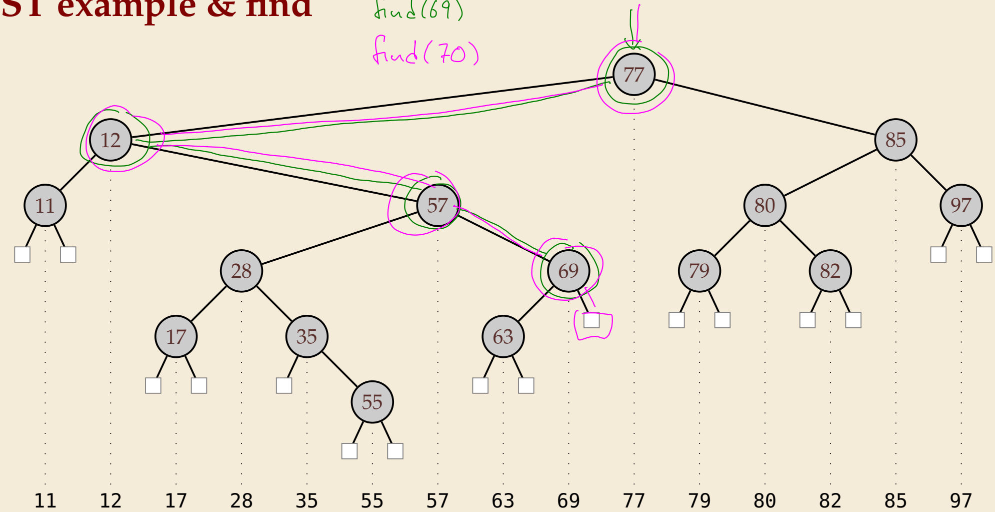
- binary tree
  - Each node has left and right child
  - Either can be empty (`null`)
- Keys satisfy ***search-tree property***

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

BST example & find

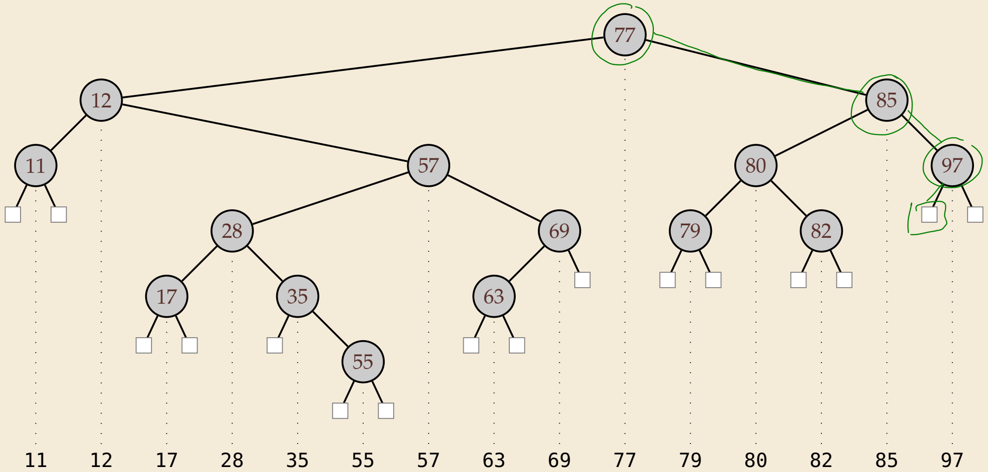
find(69)

find(70)



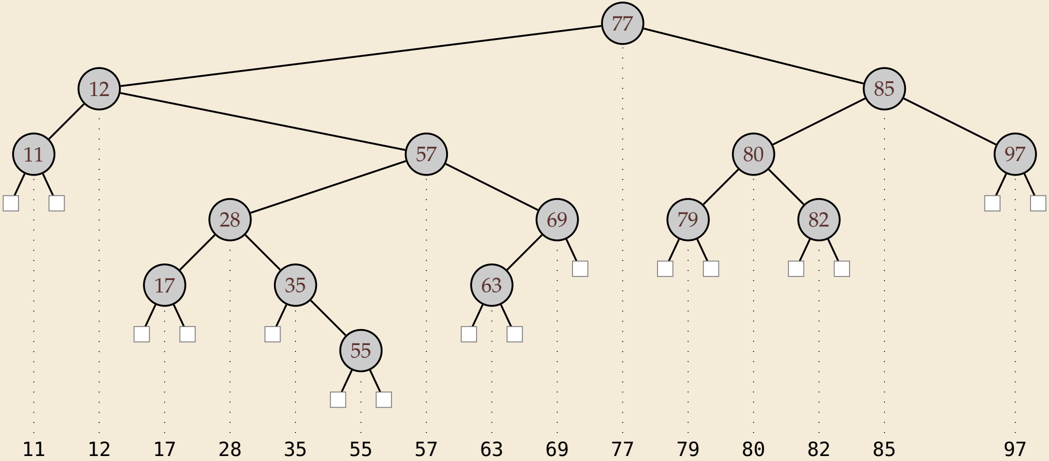
BST insert

Example: Insert 8~~8~~7



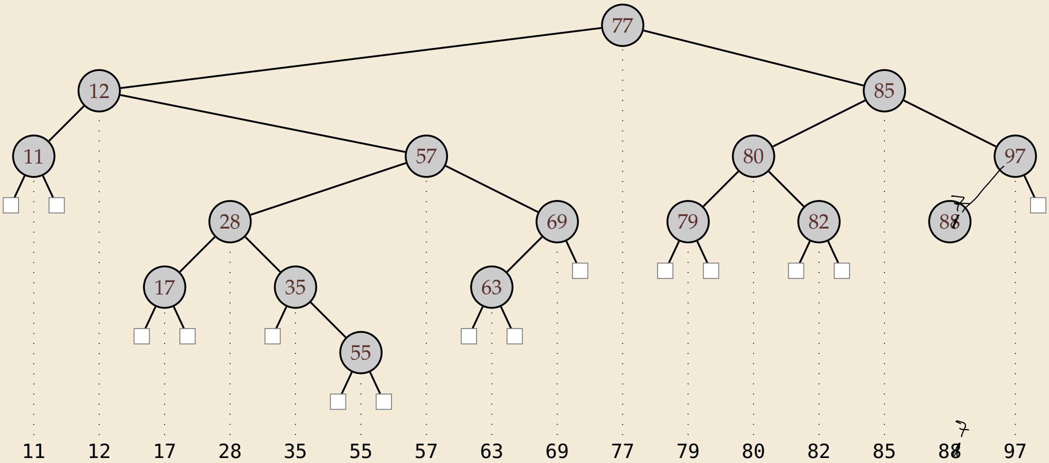
BST insert

Example: Insert 88



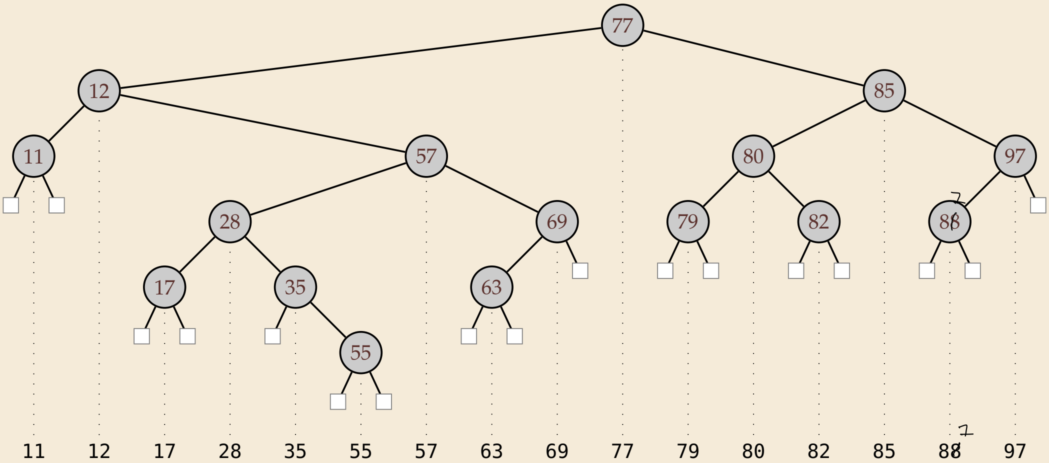
BST insert

Example: Insert 8~~8~~7



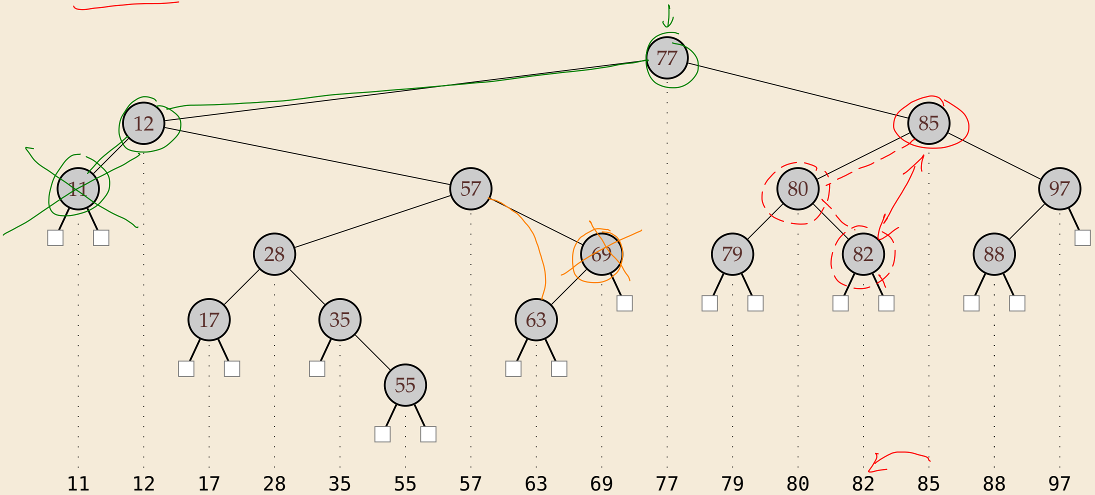
BST insert

Example: Insert 8~~8~~7



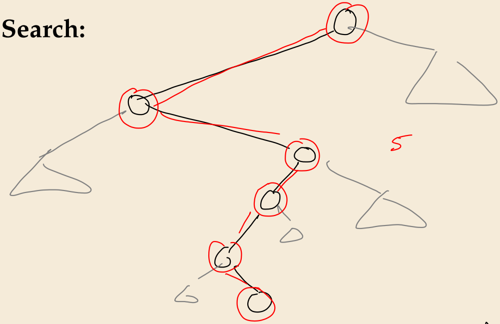
BST delete

- 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:**



$\#\text{cmps} = \text{length of search path} + 1 \le \underbrace{\text{height of BST}}_{h} + 1$

total time $O(h)$

- **Insert:** $O(h)$ unsuccessful search $\rightsquigarrow$ one path + $O(1)$ effort

- **Delete:** $O(h)$ at most one root-to-leaf path

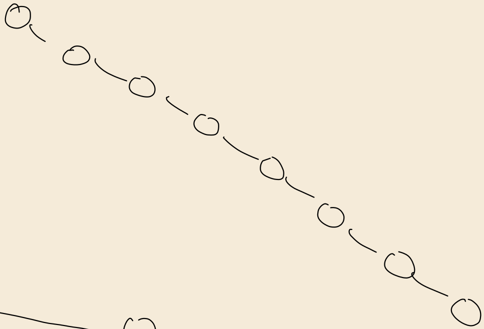
BST summary

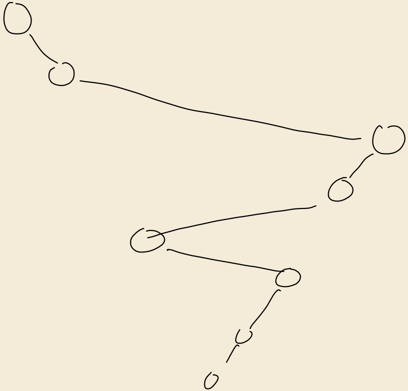
Operation	Running Time
`construct(`$A[1..n]$`)`	$O(nh)$
`put(`k`,`v`)`	$O(h)$
`get(`k`)`	$O(h)$
`delete(`k`)`	$O(h)$
`contains(`k`)`	$O(h)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

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: $\hat{=}$ random permutation model

- Assumption: insertions come in random order
  no deletions

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

even "with high probability": $\forall d\ \exists c : \Pr[h \ge c \lg(n)] \le n^{-d}$ $\Big]$ $\notin$ exam

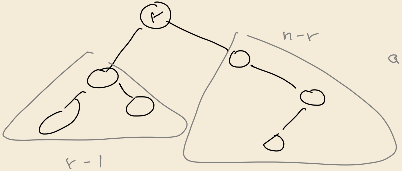
Precise analysis:

cost of finding random key in BST

easier: EXPECTED_π cost of searching ALL keys in BST

$\|$

C_n $\qquad$ $n = \#$keys $\qquad$ random: order of insertions of $[n]$



again built from random permutation

$$C_n = \overset{\text{root}}{n} + \underbrace{\frac{1}{n}\sum_{r=1}^{n}\left[C_{r-1} + C_{n-r}\right]}_{= 2\sum_{i=0}^{n-1} C_i} \qquad (n \geq 1)$$

$$C_0 = 0$$

(1) Full history $\qquad n \cdot C_n = n^2 + 2\sum_{i=0}^{n-1} C_i$

$$n\,C_n - (n-1)\,C_{n-1} = n^2 - (n-1)^2 + 2\,C_{n-1}$$

(2) Telescoping $\qquad n\,C_n = (n+1)\,C_{n-1} + 2n - 1 \quad \Big|\ :(n)(n+1) \qquad (n \geq 1)$

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2n-1}{n(n+1)}$$

$D_n := \dfrac{C_n}{n+1} \qquad D_n = D_{n-1} + \text{— '' —} \qquad (n \geq 1)$

$$= \sum_{k=1}^{n} \frac{2k-1}{k(k+1)} \qquad (\text{recursion} \rightsquigarrow \text{sum})$$

$$\frac{2k-1}{k(k+1)} = \frac{-1}{k} + \frac{3}{k+1}$$

$$D_n = \sum_{k=1}^{n}\left(\frac{3}{k+1} - \frac{1}{k}\right) = 3\sum_{k=2}^{n+1}\frac{1}{k} - \overbrace{\sum_{k=1}^{n}\frac{1}{k}}^{=H_n}$$

$$= \frac{3}{k+1} - 3 + 3H_n - H_n$$

$$= 2H_n - 3 + \frac{3}{k+1}$$

$$C_n = 2(n+1)H_n - 3(n+1) + 3 \underset{\text{TCSCS}}{\sim} 2n\ln n \approx 1.38\, n \lg n$$

$$H_n \sim \ln$$

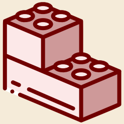
3.7 Ordered Symbol Tables

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

aside

rope: ds for strings

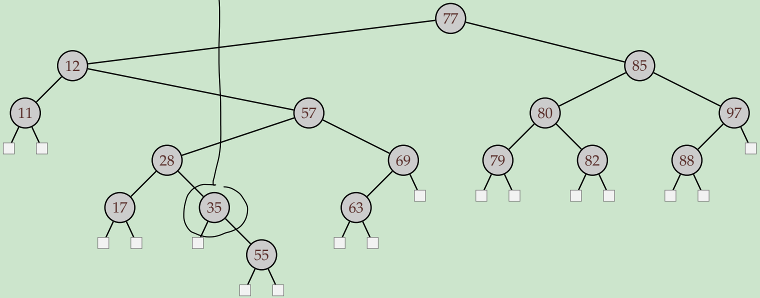


With select, we can simulate access as in a truly dynamic array!.

(Might not need any keys at all then!)

Clicker Question

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

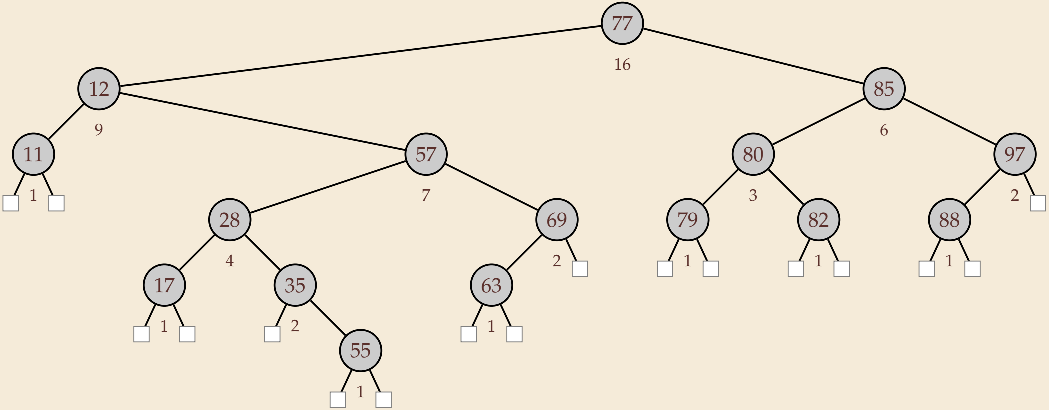


→ *sli.do/cs566*

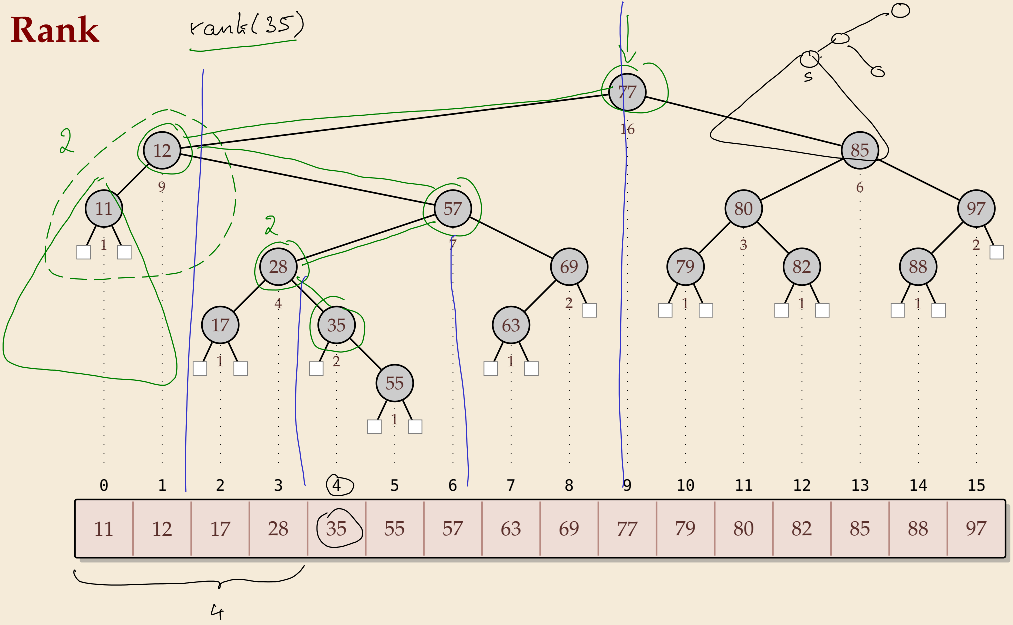
Augmented BSTs

k

s = subtree size



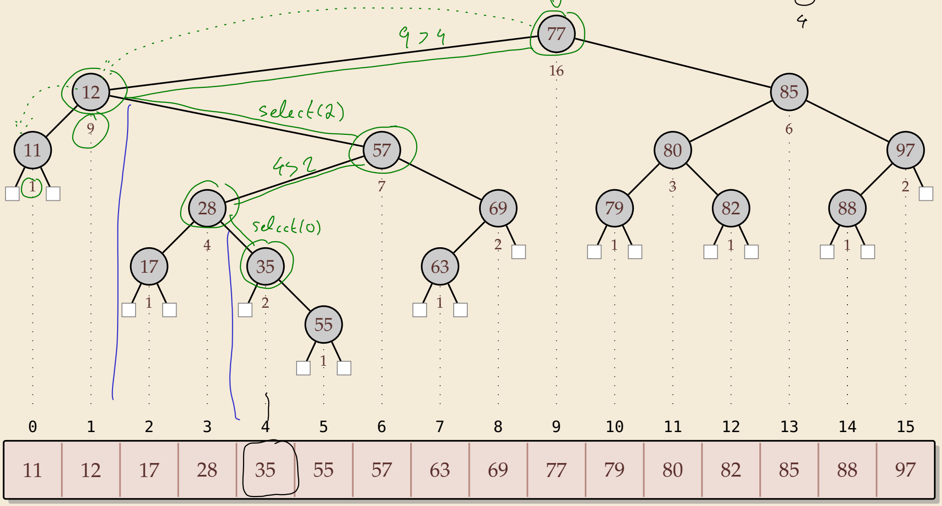
Rank

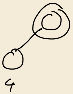


rank(35)
77
16
12
9
85
6
11
1
57
7
80
3
97
2
28
4
69
2
79
1
82
1
88
1
17
1
35
2
63
1
55
1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
11 12 17 28 35 55 57 63 69 77 79 80 82 85 88 97
4

Select

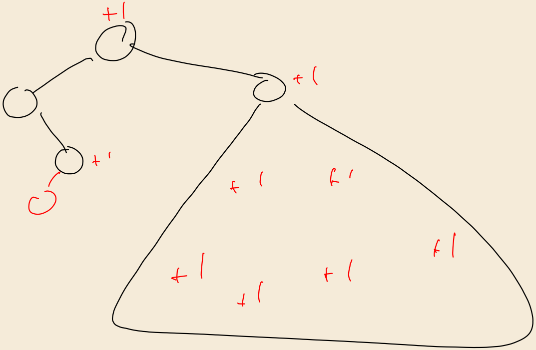
select(4)





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!

$\rightsquigarrow$ Cannot efficiently maintain node ranks.

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

3.8 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/cs566

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* ($\mathrm{BB}[\alpha]$ trees)
  - ...

Other options:

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

- **amortization:** *splay trees*, *scapegoat trees*
  COLA (cache oblivious lookahead array)
- **randomization:** *randomized BSTs*, *treaps*, *skip lists*

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`)`	$\cancel{O(\log n)}$ $O(1)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$\cancel{O(\log n)}$ $O(1)$
`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!

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?

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.

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 $\Omega(\log n)$. □

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 $\Omega(\log n)$. $\square$

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

$\rightsquigarrow$ 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 $h : U \to [0..m)$

  - 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

Symbol Table without Sorting

- key idea in hashing: everything is ultimately an integer, or can be turned into one!

⇝ hash function $h : U \to [0..m)$

  - 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

(can make it so (*"perfect hashing"*), but usually too expensive)

ϟ 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.
  - *Birthday Paradox:* quite likely! Some collision with prob. $\geq \frac{1}{e}$ when $n \geq 2\sqrt{m}$

  ⇝ need to deal with them

Handling Collision

- Two basic strategies to deal with collisions:
  - *Buckets/Chaining*: Allow multiple items at each table location
    each table location points to linked list
  - *Open addressing*: Allow each item to go into multiple locations
    need strategy to define and search these locations
    - linear probing
    - quadratic probing
    - Robin Hood hashing
    - Cuckoo hashing

(for full details of these strategies, see *Algorithms and Data Structures*)

Handling Collision

- Two basic strategies to deal with collisions:
  - *Buckets/Chaining*: Allow multiple items at each table location
    each table location points to linked list
  - *Open addressing*: Allow each item to go into multiple locations
    need strategy to define and search these locations
    - linear probing
    - quadratic probing
    - Robin Hood hashing
    - Cuckoo hashing

  (for full details of these strategies, see *Algorithms and Data Structures*)

- We evaluate strategies by the average cost of `get`, `put`, `delete`
  in terms of n, m, and/or the *load factor* $\alpha = n/m$.

$\rightsquigarrow$ 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* (not here; examples in *Algorithms and Data Structures*)

Comparison of Classic Hashing Schemes

Hash table design	Search hit	Search miss	Insert	Space	good α
Separate Chaining	$\sim \frac{1}{2}\alpha$	$\sim \alpha$	$=$ miss	$n + m$	≈ 2
Linear Probing	$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$	$\sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$	$=$ miss	m	≤ 0.5
Quadratic Probing	$\sim 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{1}{2}\alpha$	$\sim \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$	$=$ miss	m	≤ 0.7
Robin Hood Hashing	$O(1)$	$O(1)$	$=$ miss	m	≤ 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 (without space for optional optimizations)

Comparison of Classic Hashing Schemes

Hash table design	Search hit	Search miss	Insert	Space	good α
Separate Chaining	$\sim \frac{1}{2}\alpha$	$\sim \alpha$	= miss	$n + m$	≈ 2
Linear Probing	$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$	$\sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$	= miss	m	≤ 0.5
Quadratic Probing	$\sim 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{1}{2}\alpha$	$\sim \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$	= miss	m	≤ 0.7
Robin Hood Hashing	$O(1)$	$O(1)$	= miss	m	≤ 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 (without space for optional optimizations)

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