Data Structures: Review

- Abstract Data Type
- List
- Stack
- Queue
- Set
- Dictionary
- Priority Queue : Heaps

Imdadullah Khan

Introduction

- Algorithms work on data
- Data must be represented in a usable way
- Data is stored in data structures
- Choice of data structure significantly affects the efficiency of the algorithm

Abstract Data Types (ADTs)

Just as functions (procedures) extend the notion of operators in a programming language,

ADTs extend the notion of data types in a programming language

- ADTs are data types that have associated
  - set of (valid) values (type of data)
  - set of operations that can be applied on the set of values

For example the **Set** ADT could be defined as

- Sets of Integers
- Union, Intersection, set complement, set differences

Concrete implementation of an ADT is a Data Structure

Set can be implemented as arrays, linked list, bit vectors

Some Useful ADT's

Some of the frequently used ADT's are

- List
- Stack
- Queue
- Set
- Dictionary
- Priority Queue

List

- Sequence of elements of a certain type
- Elements can be linearly ordered
- Notion of position, next/previous, start/end of list
- Associated operations:
  - $\textsc{Insert}(\mathcal{L}, x, p)$
  - $\textsc{Delete}(\mathcal{L}, x)$ or $\textsc{Delete}(\mathcal{L}, p)$
  - $\textsc{Retrieve}(\mathcal{L}, p)$
  - $\textsc{Search}(\mathcal{L}, x)$
  - $\textsc{Next}(\mathcal{L}, p)$
  - $\textsc{First}(\mathcal{L})$
- Implemented using an array/linked list

Stacks

- Last-In First-Out(LIFO) list
- Associated operations:
  - PUSH(x)
  - POP()
  - ISEMPTY()
  - ISFULL()
  - SIZE()
- Implemented using an array/linked list
- Use: In OS to handle recursive and procedural calls, DFS

Queues

- First-In First-Out (FIFO) list
- Associated operations:
  - ENQUEUE(x)
  - DEQUEUE(x)
  - ISEMPTY()
  - ISFULL()
  - SIZE()
- Implemented using an array/linked list
- Use: In processes scheduler, BFS

Dictionary

- The **Set** ADT, in addition to INSERT, SEARCH and DELETE, includes operations such as UNION ($A \cup B$), INTERSECTION ($A \cap B$) and SUBTRACTION ($A \setminus B$)
- The full Set ADT is not generally needed
- E.g. Students' record at RO (Zambeel)
- E.g: How would you store the quiz scores for a set of students?
- We need to maintain a set with insertion, deletion and searching
- The scores of a quiz can be represented using a dictionary with roll numbers as keys and scores as values

  scores $=$
  $\{`16020102' : 17,$
  $`11010051' : 84,$
  $`11050001' : 22,$
  $`12060009' : 92\}$

Dictionary

- A dictionary maintains a <u>set</u> of elements
- Unique elements; elements are known by their "keys"
- Elements could be compound $(key, value)$ pairs
- Associated operations:
    - $\textsc{Insert}(\mathcal{D}, k, v)$
    - $\textsc{Delete}(\mathcal{D}, k)$
    - $\textsc{Search}(\mathcal{D}, k)$
    - $\textsc{isEmpty}(\mathcal{D})$
    - $\textsc{Size}(\mathcal{D})$
- What if an entry (k, v') exists in $\mathcal{D}$ and $\textsc{Insert}(\mathcal{D}, k, v)$ is called?
- Dictionary can be implemented using
    - an array (sorted or unsorted)
    - a linked list (sorted or unsorted)
    - binary search trees (balanced or unbalanced)
    - hash tables

Dictionary Implementations - Array

Unsorted Array:

- SEARCH: Linear search - traverse array sequentially ▷ $O(n)$
- INSERT: Insertion at the end of array (first empty slot) ▷ $O(1)$
- DELETE: Given a position, shift left remaining elements ▷ $O(n)$

Sorted Array:

- SEARCH: Binary search; repeatedly halve search interval ▷ $O(\log n)$
- INSERT: Lookup to find position and shift to make space ▷ $O(n)$
- DELETE: Given a position, shift left remaining elements ▷ $O(n)$

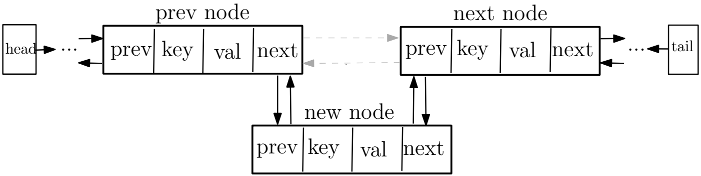
Dictionary Implementations - Linked List

Unsorted Linked List:

- LOOKUP: Linear search ▷ $O(n)$
- INSERT: Insertion at start or end ▷ $O(1)$
- DELETE: Lookup and link previous with next (doubly linked list) ▷ $O(n)$

Sorted Linked List:

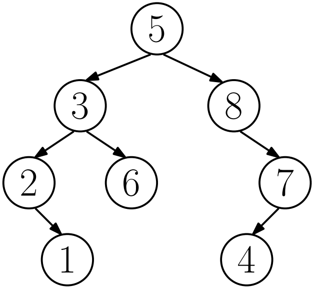
- LOOKUP: Linear search (can't jump to mid directly) ▷ $O(n)$
- INSERT: Lookup to find position and update previous and next ▷ $O(n)$
- DELETE: Lookup and link previous with next ▷ $O(n)$



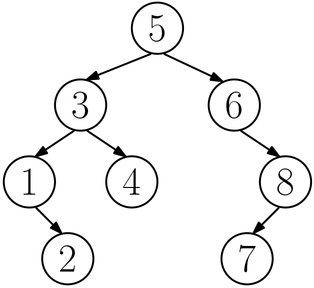
Insertion to a sorted doubly linked list

Dictionary Implementations - Binary Search Tree

- A binary tree has a root node, a left subtree and a right subtree
- Each node contains data, left pointer and right pointer
- Binary Search Tree is a Binary Tree with additional properties:
  - Nodes have keys for comparison
  - Keys in left subtree are smaller than node's key
  - Keys in right subtree are larger than node's key



Binary Tree



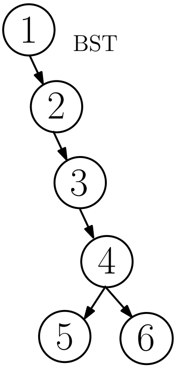
Binary Search Tree

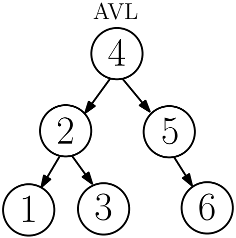
Dictionary Implementations - Binary Search Tree

- For a dictionary, the node data is a (*key*) pair
- SEARCH: Compare with root; recursively lookup in appropriate subtree
- INSERT: Lookup for appropriate leaf position to insert node
- DELETE: - Given key, lookup to find pointer to node. Given pointer to node, remove and recursively link parent with one of the children Read Textbook
- For a BST of height h, all the above operations take $O(h)$ time

Dictionary Implementations - AVL Tree

- An AVL tree is a binary search tree with additional properties:
  - The height of the left and right subtree of a node differ by at most 1
  - The left and right subtrees of a node are AVL trees
- AVL tree is a *balanced* BST; it's height is always $O(log(n))$



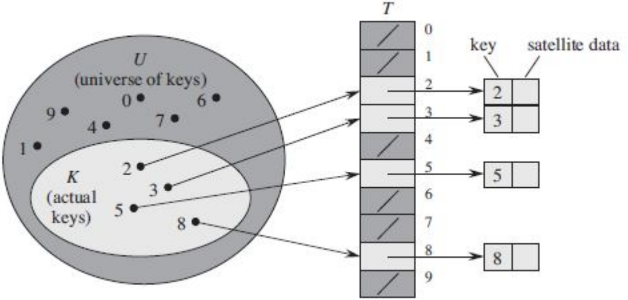


Dictionary Implementations - AVL Tree

- SEARCH, INSERT and DELETE methods are the same as BST but an AVL tree may become unbalanced after INSERT and DELETE
- An unbalanced tree is rebalanced using rotation; an adjustment to the tree, around an item, that maintains the required ordering of items Read Textbook
- All operations of AVL Tree have the same time complexity as BST i.e. $O(\log n)$, as rotation takes only constant time

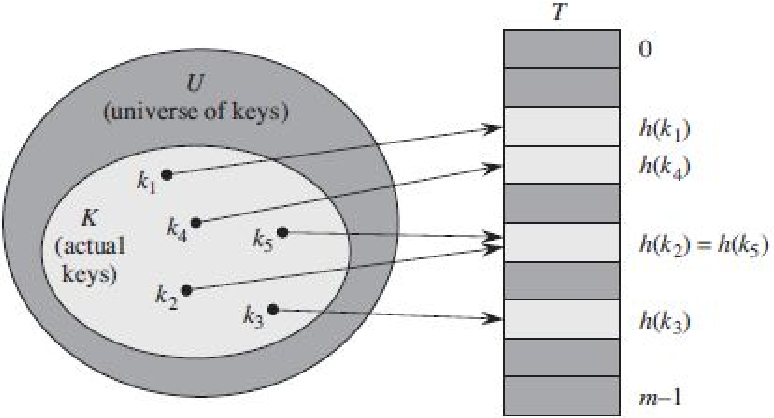
Dictionary Implementations - Hash Tables

- Suppose n data elements are to be stored in a dictionary with keys $k \in U$ where universe set $U = [1 \ldots N]$
- Let $m \in \mathbb{Z}^+$ and $h : U \to [m]$
- Make a array (or table) $T[1, ..., m]$
- SEARCH: **return** $T[h(k)]$ $\triangleright$ $O(1)$
- INSERT: $T[h(k)] \leftarrow 1$ $\triangleright$ $O(1)$
- DELETE: $T[h(k)] \leftarrow 0$ $\triangleright$ $O(1)$



Dictionary Implementations - Hash Tables

- What if $h(k_x) = h(k_y)$? Collision occurs

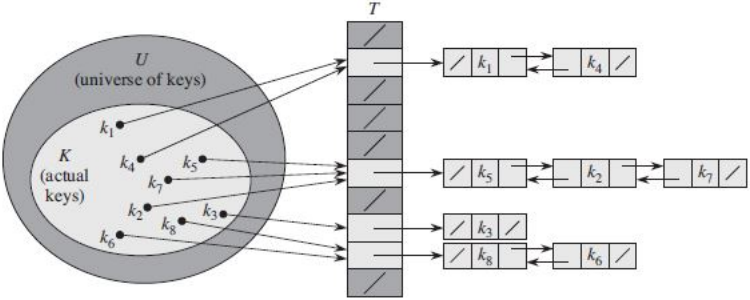


Collision occurs between k_2 and k_5

Dictionary Implementations - Hash Tables

Hashing with Chaining:

- Make $T[1, \dots, m]$ an array of linked lists
- LOOKUP: Lookup in list $T[h(k)]$
- INSERT: Insert in list $T[h(k)]$
- DELETE: Delete from list $T[h(k)]$
- Runtime of all operations: O(length of longest list in $T[k]$)
  ensure not many keys k map to the same index in T under h



Dictionary Implementations - Hash Tables

Uniform hashing:

- each key k has an equal chance of being mapped to the an index in T under h

$$Pr[h(x) = h(y)] = \frac{1}{m}$$

- Expected length of list of each index $= \frac{n}{m}$
- Space and time complexity trade-off controlled by size of table m
- Example: **Linear Congruential Hash Function**
  Select a prime number $p \geq m$
  Choose integers a, b randomly s.t. $a \neq 0$
  Then,

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m)$$

Read Number Theory slides from CS 210 Discrete Mathematics

Dictionary Implementations - Summary

- Runtimes of dictionary operations: different implementations

Operation	Unsor. Array	Sor. Array	Unsor. L-list	Sor. L-list	BST	AVL	Hash Function
Search(D,k)	$O(n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$	$O(1)$
Insert(D,k,v)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(h)$	$O(\log n)$	$O(1)$
Delete(D,k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(h)$	$O(\log n)$	$O(1)$

Priority Queue ADT

- Data elements have an associated priorities (called keys)
- Retrieval is done on the basis of this priority
- Operations:
  - $\mathcal{P} = \textsc{Initialize}()$
    - Initialize a priority queue with n elements with associated priorities
  - $\textsc{Insert}(\mathcal{P}, v, k)$
    - Insert an element v with priority k,
  - $\textsc{ExtractMin}(\mathcal{P})$
    - Returns the element with minimum priority and delete it (also called $\textsc{deleteMin}$)
    - One can analgously define $\textsc{ExtractMax}(\mathcal{P})$
  - $\textsc{DecreaseKey}(\mathcal{P}, v, k')$
    - Change the priority of element v to k'
    - One can analgously define $\textsc{IncreaseKey}(\mathcal{P}, v, k')$

Priority Queue ADT: Applications

Used in many algorithms

- scheduling systems
- shortest process first
- longest request first
- Cache Replacement algorithms (e.g. LRU, LFU)
- Hierarchical (Agglomerative) Clustering
- Dijkstra's and Prim's algorithm

Priority Queue: Implementation

Can be Implemented using

- Arrays (sorted or unsorted)
- Linked List (sorted or unsorted)
- Binary Heaps

Note: The terms Heap and Priority Queue are often used interchangeably as priority queues are mostly implemented using heaps and they support the same operations

Priority Queue: Implementation

- Unsorted Array
  - INITIALIZE: create array with elements in arbitrary order ▷ $O(n)$
  - INSERT: insert at the end of the array ▷ $O(1)$
  - EXTRACTMIN: FINDMIN in array by key, return, delete and shift ▷ $O(n)$
- Sorted Array
  - INITIALIZE: sort array in descending order by key ▷ $O(n \log n)$
  - INSERT: binary search for position and shift elements on right ▷ $O(n)$
  - EXTRACTMIN: remove the last element from array ▷ $O(1)$
- Unsorted Linked-List (doubly linked list)
  - INITIALIZE: create linked list with elements in arbitrary order ▷ $O(n)$
  - INSERT: insert new node at head of linked list ▷ $O(1)$
  - EXTRACTMIN: FINDMIN in array by key, return, delete and shift ▷ $O(n)$
- Sorted Linked-List
  - INITIALIZE: sort linked list in ascending order by key ▷ $O(n^2)$
  - INSERT: lienear search for position and insert new node ▷ $O(n)$
  - EXTRACTMIN: remove the element at head of linked-list ▷ $O(1)$

DECREASEKEY is essentially a search, replace and (if needed) reorder

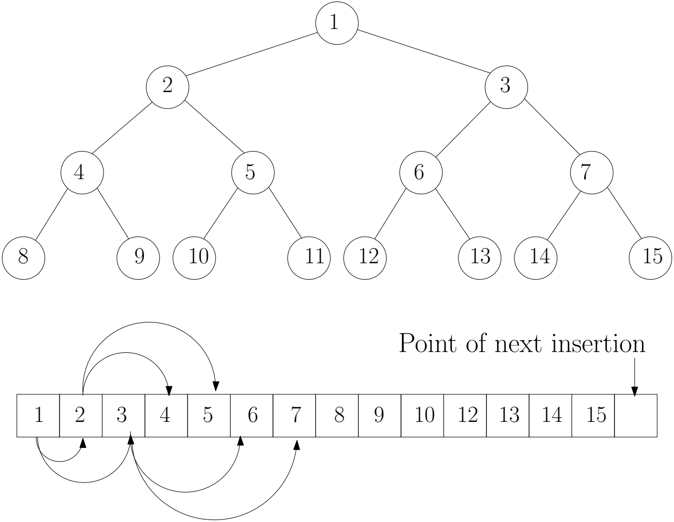
Priority Queue: Heap Implementation

- A binary heap can be used to implement priority queues
- A rooted binary tree that satisfies the *heap property*
- *Min-Heap Property:* If u is parent of v, then $key(u) \leq key(v)$
- *Max-Heap Property:* If u is parent of v, then $key(u) \geq key(v)$
- Associated Operations (min-heap):
  - $\mathcal{H} \leftarrow \textsc{Initialize}()$ $\triangleright\ O(n)$
    builds a heap given a set of data elements with keys.
  - $\textsc{Insert}(H, v, k)$ $\triangleright\ O(\log n)$
    inserts an element v with key value k in H
  - $\textsc{Delete}(H, v)$ $\triangleright\ O(\log n)$
    deletes the element x from H given the pointer to x
  - $\textsc{DecreaseKey}(H, v, k')$ $\triangleright\ O(\log n)$
    decreases the key of element x in H to new value k' given pointer to x
  - $v \leftarrow \textsc{extractMin}(H)$ $\triangleright\ O(\log n)$
    returns the element v with min key value and deletes it from H
- Priority of each element in $\mathcal{P}$ is key of the respective element in $\mathcal{H}$.

Min-Heap

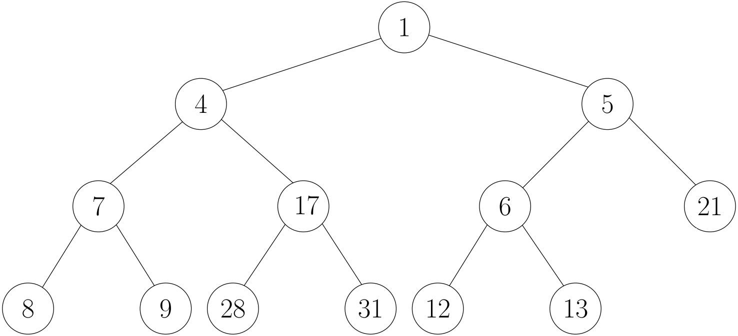
- A min-heap maintains a set of n elements each with a key and satisfies the min-heap property.
- Heap implementation uses a **complete binary tree** (binary heap) $\mathcal{H}$
- Every node has a key smaller than its both children
- Data element with minimum key is at the root
- Binary heap can also be represented as an array $\mathcal{A}$ of n elements
- The minimum value (root) is at A[1]
- The left and right child of a node at $\mathcal{A}[i]$ are at $\mathcal{A}[2i]$ and $\mathcal{A}[2i+1]$, respectively
- The parent of a node at $\mathcal{A}[i]$ is at $\mathcal{A}[\lfloor \frac{i}{2} \rfloor]$
- The sequence of vertices visited by **level order traversal** (BFS) of the binary tree maps to the order of nodes in the array representation

Min-Heap: Binary Tree and Array Representation



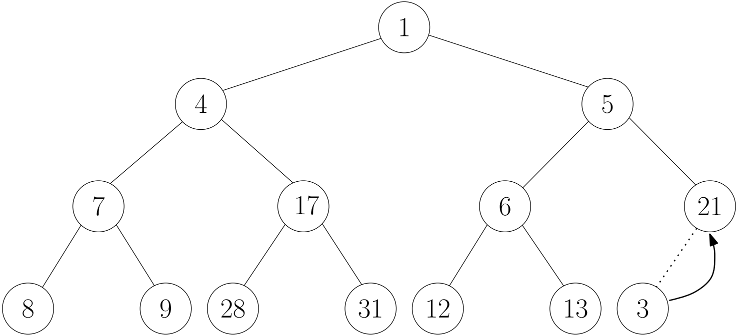
A binary min-heap and its array representation

Min-Heap Operations: Insert



$\text{Insert}(H, v, 3)$

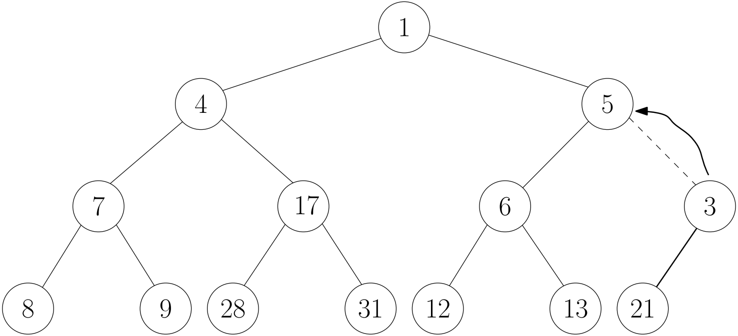
Min-Heap Operations: Insert



INSERT$(H, v, 3)$

Insert v to next available position in H

Min-Heap Operations: Insert

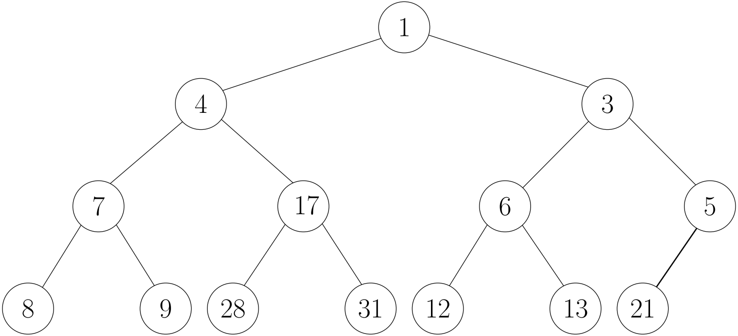


Insert$(H, v, 3)$

Sift-up the added node as needed to restore heap property

Each Sift-up moves the node to the level above in the tree

Min-Heap Operations: Insert

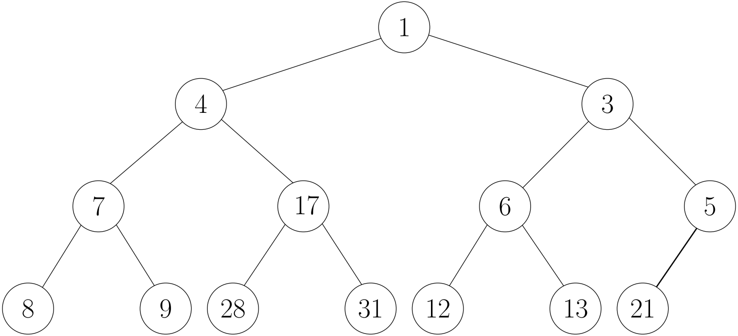


INSERT$(H, v, 3)$
Min-Heap property restored
At max, as many Sift-Up moves can be made as the height of the tree
Recall: Height of a balanced complete binary tree with n nodes is $O(\log n)$
Therefore, sifting up takes $O(\log n)$ time

Pseudocode : Insert, Decrease-Key and Sift-Up

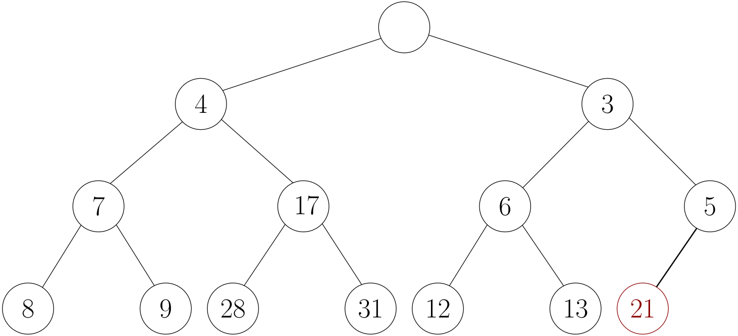
```
function Insert(H, v, k)                                          ▷ O(log n)
    H.Append(v)          ▷ insert v at end, i.e. last available position in H
    key(H[v]) ← k
    SiftUp(H, v)

function SiftUp(H,v)                                              ▷ O(log n)
    p ← getParent(v)
    if key(H[v]) < key(H[p]) then
        swap(H[v], H[p])
        SiftUp(H, p)

function DecreaseKey(H, v, k)                                     ▷ O(log n)
    key(H[v]) ← k
    SiftUp(H, v)
```

Min-Heap Operations: Extract-Min



Extract-Min(H)

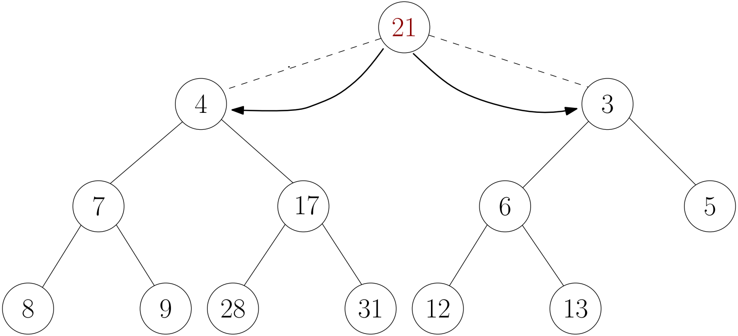
Min-Heap Operations: EXTRACT-MIN



EXTRACT-MIN(H)

Extract the root node to be returned

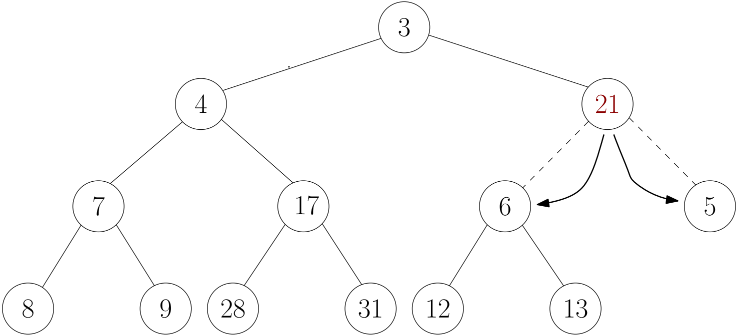
Min-Heap Operations: Extract-Min



Extract-Min(H)

Delete the last filled node at its place and move its element to root

Min-Heap Operations: Extract-Min

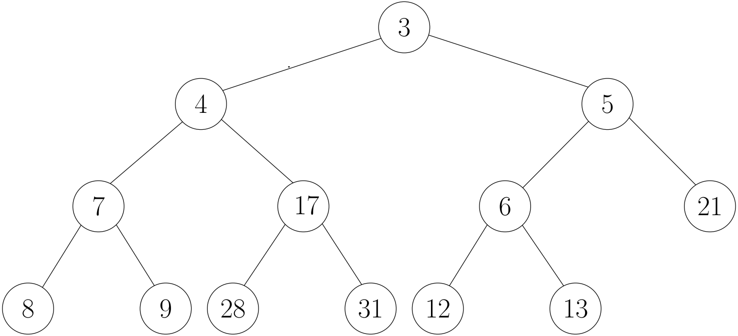


Extract-Min(H)

Sift-down as needed to restore heap property

Each sift-down moves the node to the level below in the tree

Min-Heap Operations: Extract-Min



Extract-Min(H)

Min-heap property restored

At max, as many Sift-Down moves can be made as the height of tree

Recall: Height of a balanced complete binary tree with n nodes is $O(\log n)$

Therefore, sifting down takes $O(\log n)$ time

Pseudocode: Extract-Min and Delete

---

function EXTRACT-MIN(H) $\triangleright$ $O(\log n)$
  return DELETE(H, *root*)

function DELETE(H, *node*) $\triangleright$ $O(\log n)$
  $key \leftarrow H[node]$ $\triangleright$ *node* is pointer to element to be removed
  SWAP($H[node], H[lastFilledPosition]$)
  REMOVE($H[lastFilledPosition]$)
  SIFTDOWN(*node*)
  return *key*

Pseudocode: Sift-Down

```
function SiftDown(H, node)                                  ▷ O(log n)
    l ← leftChild(node)
    r ← rightChild(node)
    if key(H[l]) < key(H[r]) then
        if key(H[node]) > key(H[l]) then
            Swap( H[node], H[l] )
            SiftDown(H, l)
    else
        if key(H[node]) > key(H[r]) then
            Swap( H[node], H[r] )
            SiftDown(H, r)
```