COMP 8547
Advanced Computing Concepts

Dr. Olena Syrotkina

Fall 2022: 02– Linear Data Structures

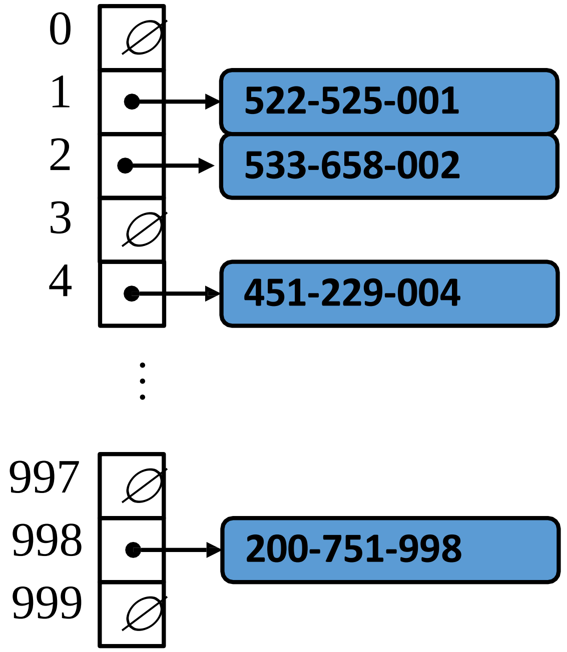
Submission Deadlines

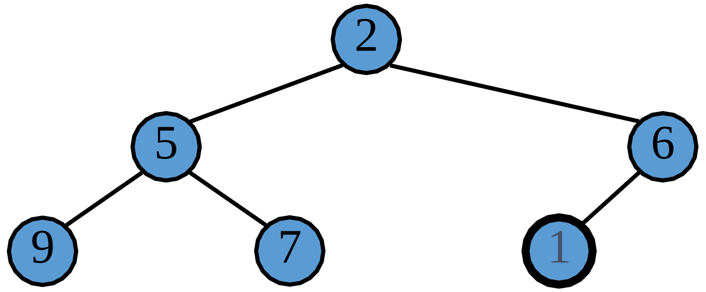
1. Forming a group: **Tonight (Sept 15), 23:59**
2. Choosing your variant for the final project: **Sept 22, 23:59**
3. Assignment 1 (group submission) will be available as soon as you form the Group: **Sept 29, 23:59**

Chapter 2 – Linear Data Structures

Contents

- Abstract Data Types
- Maps
- Hash tables
- Sorted maps
- Priority queues
- Binary heaps
- Heap construction
- Advanced heaps
- Heaps in Java
- Applications
  - Selection problem
  - Counting word frequencies





Abstract Data Types (ADTs)

Def.: An abstract data type (ADT) is a set of objects with a set of operations

- Can be seen as mathematical abstractions

Mathematical abstraction is the process of considering and manipulating operations, rules, methods and concepts deprived from their reference to real world phenomena and circumstances, and also deprived from the content connected to particular applications.

- An ADT definition involves:
  - data, operations, error conditions.
  - Implementations involve algorithms along with their *time* and *space* complexities

Examples of ADTs:

- Lists, stacks, array lists
- Sets, maps, dictionaries, heaps, hash tables
- Graphs

Operations:

- Add, remove, contains, union, find, min, insert, delete, etc.

Map ADT

Main features

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Assume for now that keys are unique
- Applications:
  - address book
  - student-record database
  - Word counts
- Implementation:
  - Sorted maps:
    - Array list (binary search), skip list, binary search trees
  - Unsorted maps:
    - Hash table

Main methods

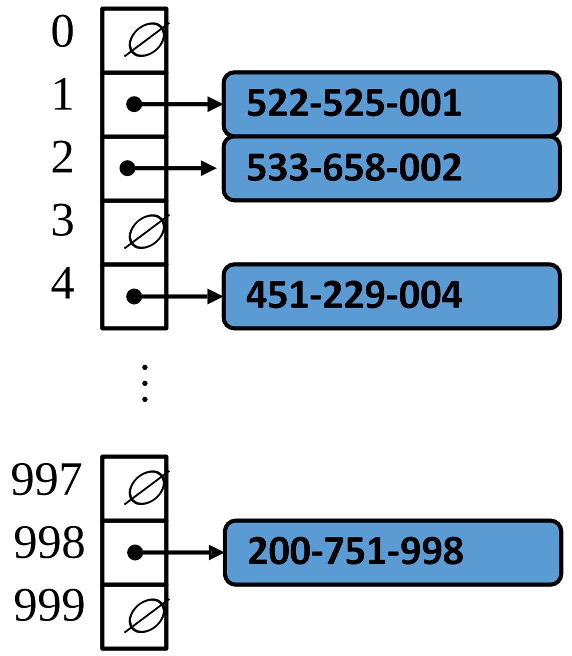
- size(), isEmpty()
- get(k): if the map M has an entry with key k, return its associated values; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- keys(): return an iterator of the keys in M
- values(): return an iterator of the values in M

Hash tables

- A hash table for a given key type consists of:
  - Hash function h
  - Bucket Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index i = h(k)
- A hash table is a way of storing a map in an unsorted list
- A hash function h maps keys of a given type to integers in a fixed interval [0, N - 1]
- Example:

  h(x) = x mod N

  is a hash function for integer keys
- The integer h(x) is called the hash value of key x



A hash table that stores SIN on an array of size N = 1,000

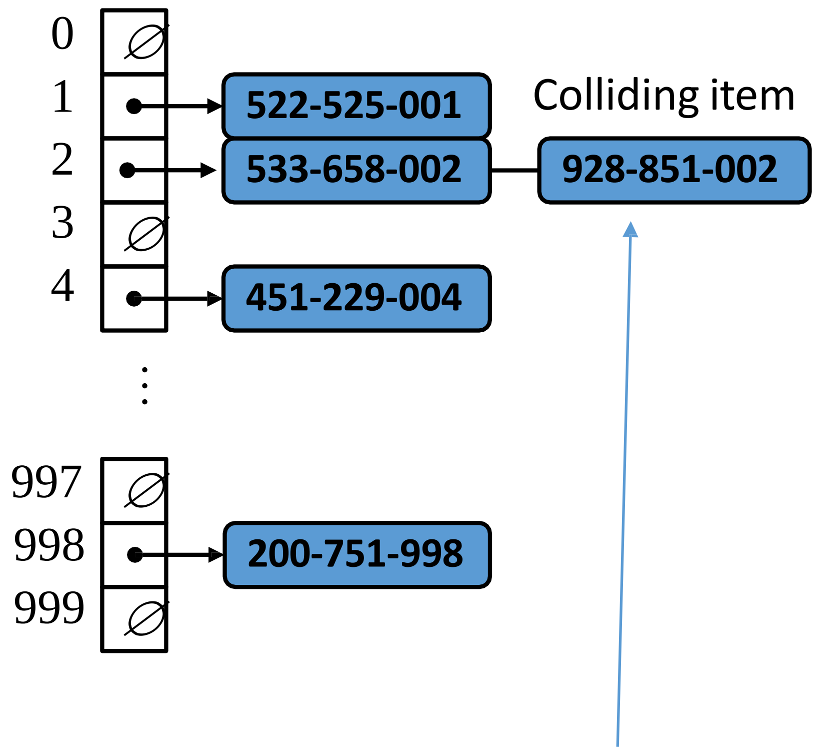
The index of the bucket array given by the last three digits of the SIN

Collisions

- Two objects map to the same cell in the table

Strategies to handle collisions

- Separate chaining
  - each cell in the table point to a linked list of entries that map there
  - It is a simple strategy but requires additional memory
- Open addressing
  - the colliding item is placed in a different cell of the table
  - Main techniques: linear probing, quadratic probing, double hashing

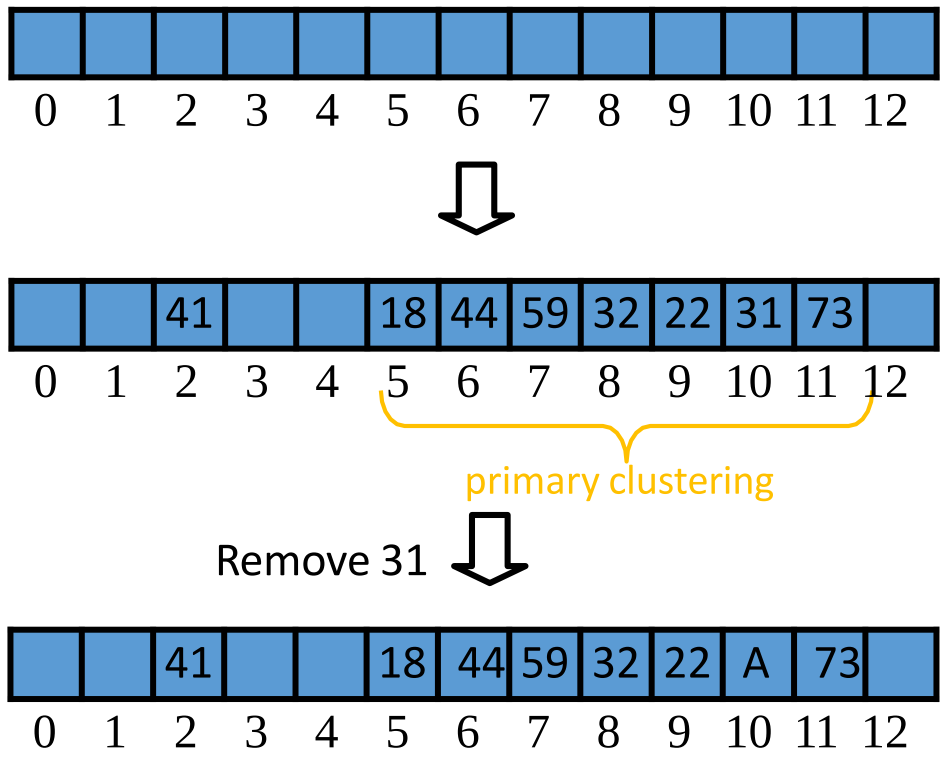


Separate chaining adds colliding item to a linked list

Linear probing

- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Inserting (k,v) into bucket A[i] that is already occupied:
  - Try A[(i+1) mod N], A[(i+2) mod N], and so on, until an empty bucket is found.
- Colliding items lump together (**primary** clustering), causing future collisions to yield a longer sequence of probes
- Removals will place a mark "A" in the removed item

- Example:
  - N = 13
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in that order

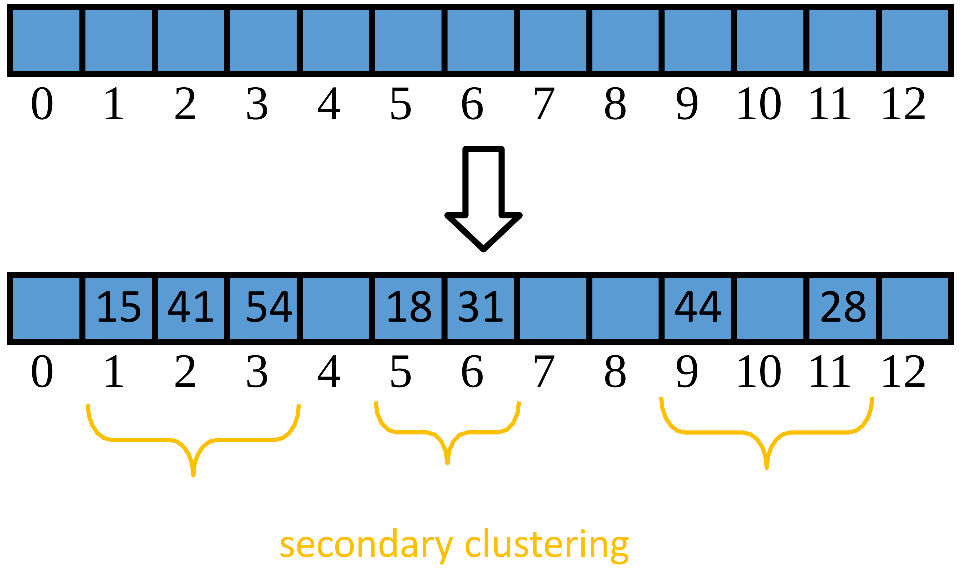


Quadratic probing

- **Quadratic probing** also handles collisions by placing the colliding item in an available bucket
- Inserting (k,v) into bucket A[i] that is already occupied:
  - Try A[(i+j^2) mod N], for j = 1, 2, …, N-1
  - until we find an empty bucket.
- Quadratic probing is not guaranteed to find an empty bucket.
- Colliding items are sparsely located in the table, avoiding **primary** clustering
- However, it causes **secondary** clustering: the set of filled buckets "bounces" around the array on a fixed pattern

Example:

- h(x) = x mod 13
- Insert keys 18, 41, 31, 54, 28, 44, 15, in that order
- Collisions: A[(i+j2) mod N] for j = 1, 2, …, N-1



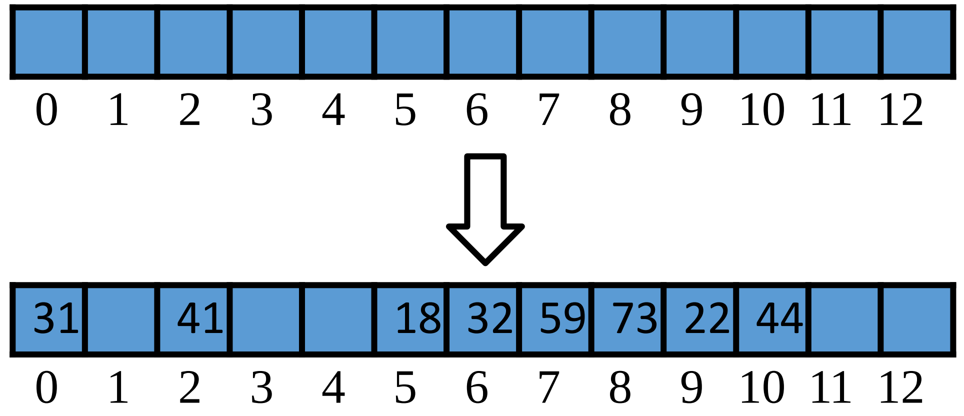
Double hashing

- Double hashing uses a secondary hash function $d_2(k)$ and handles collisions by placing an item in the first available cell of the series
  $(i + jd_2(k)) \bmod N$
  for $j = 1, \ldots, N - 1$
- The secondary hash function $d_2(k)$ should not have zero values
- The table size N must be prime to allow probing of all the cells
- Common choice of compression function for the secondary hash function:
  $d_2(k) = q - k \bmod q$
  where
  - $q < N$
  - q and N are prime
- The possible values for $d_2(k)$ are 1, 2, …, q

Example:

- $N = 13$
- $h(k) = k \bmod 13$
- $d_2(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in that order

k	$h(k)$	$d2(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

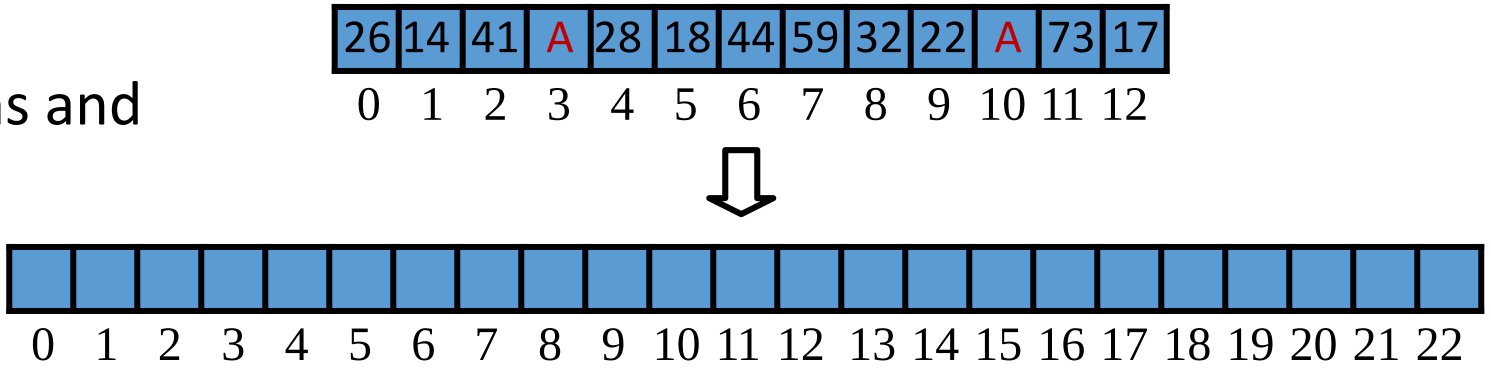


Performance of Hashing

- The load factor ***a*** = ***n***/***N*** affects the performance of a hash table
- The number of probes for a search in linear probing is $p = \frac{1}{2}(1 + \frac{1}{1-\alpha})$
- If $\alpha \rightarrow 0$, p is a constant
- If $\alpha \rightarrow 1$, p$\rightarrow \infty$
- Ideal load factor is α<0.5
- Then, expected running time of a search is O(1)
- The same is true for insertions and removals

Rehashing

- When the hash table becomes full or load factor is too high
- Rehashing is the best option:
  - Create a new empty hash table
  - Remove all elements from the old table, one at a time
  - Insert them into the new hash table, one at a time
- Example: ***h***(***x***) = ***x*** mod 13 $\Rightarrow$ ***h***(***x***) = ***x*** mod 23



Hash tables in Java

- HashMap implements a Hash table in Java
- Uses separate chaining
- Constructor allows to specify capacity and load factor

```
//Constructs an empty HashMap with the specified initial
capacity and load factor.

public HashMap(int initialCapacity, float loadFactor)
```

- Default capacity is 16 and load factor is 0.75
- It is an implementation of the Map interface:
  - Allows to specify the type of keys and values

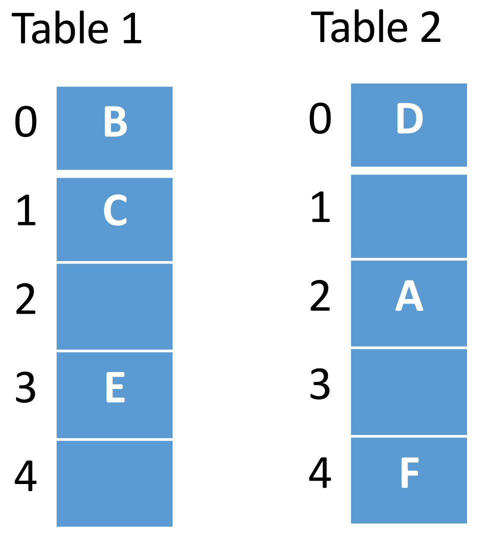
```
public HashMap(Map<? extends K,? extends V> m)
```

Cuckoo hashing

Main Idea

- Keep two hash tables
- Insert key k in Table 1
- If cell in Table 1 is occupied by k’, displace k’ (insert it in Table 2)
- If cell in Table 2 is occupied by k”, insert k” in Table 1, and so on
- Until an empty cell is found
- If load factor is low, it is unlikely that more than O(log N) displacements will occur
- After a number of displacements occur, the hash table can be rebuilt

Example



A:	0, 2
B:	0, 0
D:	1, 0
C:	1, 4
F:	3, 4
E:	3, 2

Advanced hashing

- Perfect Hashing
  - Uses separate chaining (each bin contains a linked list)
  - If the number of keys in the list is at most *constant*, then a search will run in *worst-case* O(1)
  - It may not be the case unless the keys are uniformly distributed
- Hopscotch hashing
  - Uses the idea of linear probing
  - After collision, item is placed no mar than max_dist of initial probe
  - It is a new algorithm – now at an experimental level
- Universal hashing – main idea
  - Hash function must be computed in constant time
  - Items are uniformly distributed across array cells
  - Thus, insertions and searches are done in O(1)
- Extendible hashing
  - It applies to large datasets that are too large to fit in main memory
  - Uses disk blocks of M records to store hash entries
  - Aims at achieving searches in at most two disk accesses
  - Insertions may require a few disk accesses
  - Uses a directory to access disk blocks – more details in Ch. 8

Priority queues

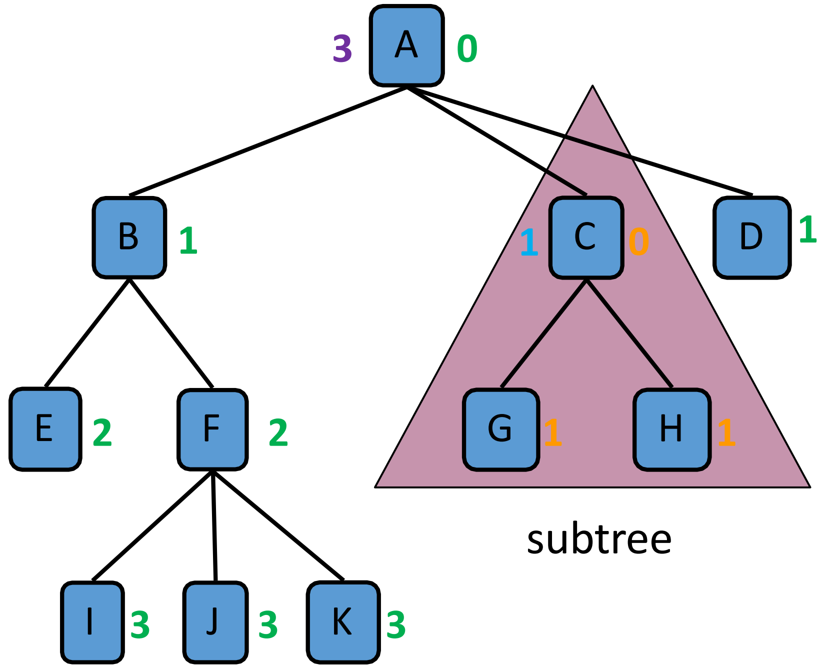
- A priority queue (PQ) stores a collection of entries
- Each **entry** is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, x)
    inserts an entry with key k and value x
  - insertLast(k)
    insert at the end of the priority queue
  - removeMin()
    removes and returns the entry with the smallest key
  - min()
    returns, but does not remove, an entry with the smallest key
- Applications
  - Huffman coding tree
  - Search techniques
  - Operating systems
  - Sorting/selection

- Comparison defines a total order
- The primary method of the Comparator ADT:
  - compare(a, b): Returns an integer *i* such that:
  - $i < 0$ if $a < b$
  - $i = 0$ if $a = b$
  - $i > 0$ if $a > b$
  - an error occurs if *a* and *b* cannot be compared.
- Can be implemented in
  - sorted list: min can be retrieved in O(1)
  - unsorted list:
    - ➢ Heap: min can be retrieved in O(log n)
    - ➢ Any order: min can be retrieved in O(n)

Trees – definition, terminology

- Graph theory: A tree T is an undirected acyclic graph
- A tree can be either unrooted or rooted. This chapter covers rooted trees
- Informally: A tree T is a set of nodes, which have a parent-child relationship
- Subtree: tree consisting of a node of T and its descendants
- Properties:
  - If T is nonempty, it has a single node called the root of T
  - Each node v of T (except the root) has a unique parent
  - Every node v with parent w is a child of w
  - A tree T can be empty

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (or leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree (or a subtree): maximum depth of any node
- Descendant of a node: child, grandchild, grand-grandchild, etc.



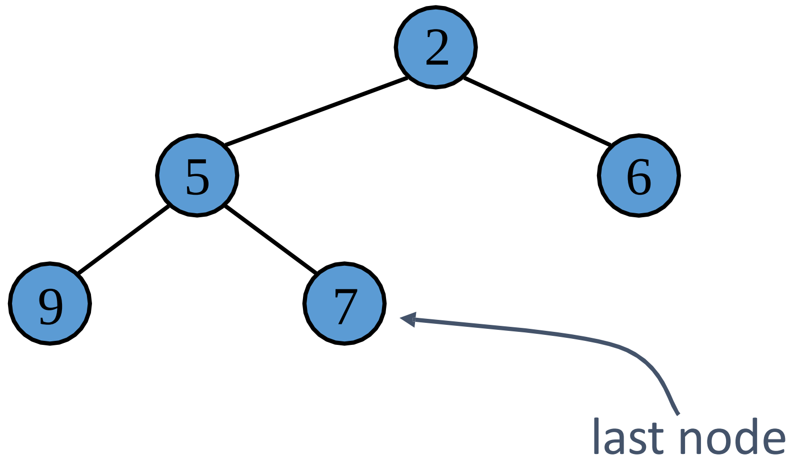
Binary tree

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for **proper** binary trees)
  - The children of a node are in an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - an empty tree,
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

Binary heap

- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
  - Heap-Order: for every node v other than the root

    $$\text{key}(v) \geq \text{key}(\text{parent}(v))$$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
    - at depth h, the external nodes are arranged to the left of the tree
- The last node of a heap is the rightmost node of depth h



Height of a Heap

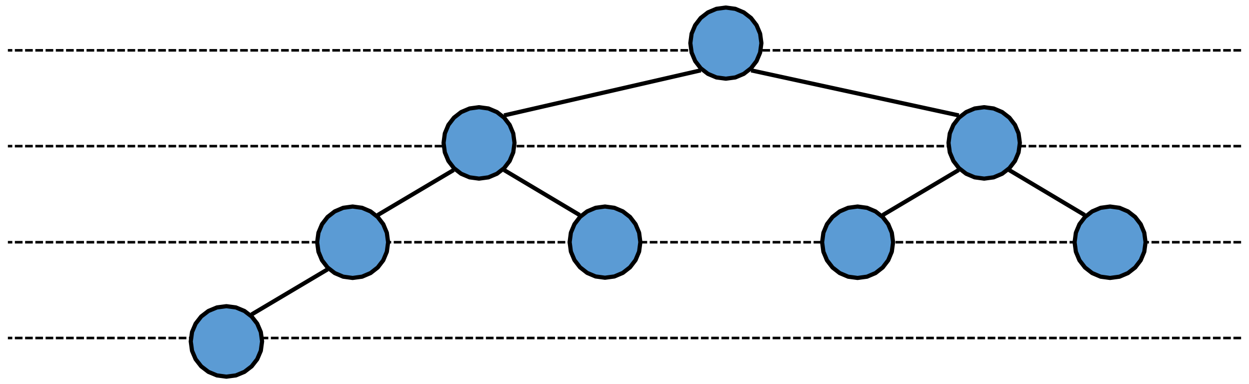
- Theorem: A heap storing $\boldsymbol{n}$ keys has height $\boldsymbol{O}(\log \boldsymbol{n})$

  Proof: Follows the complete binary tree property
  - Let $\boldsymbol{h}$ be the height of a heap storing $\boldsymbol{n}$ keys
  - Since there are $2^{\boldsymbol{i}}$ keys at depth $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 1$ and at least one key at depth $\boldsymbol{h}$, we have $\boldsymbol{n} \geq 1 + 2 + 4 + \ldots + 2^{\boldsymbol{h}-1} + 1$
  - Thus, $\boldsymbol{n} \geq 2^{\boldsymbol{h}}$, i.e., $\boldsymbol{h} \leq \log \boldsymbol{n}$

$2^h \le n \quad \Rightarrow \quad \log 2^h \le \log n$

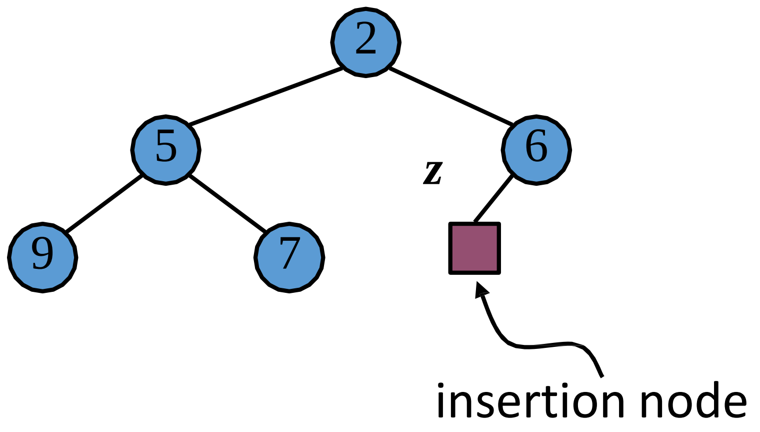
$\Rightarrow \quad h \log 2 \le \log n \quad \Rightarrow \quad h \le \log n$

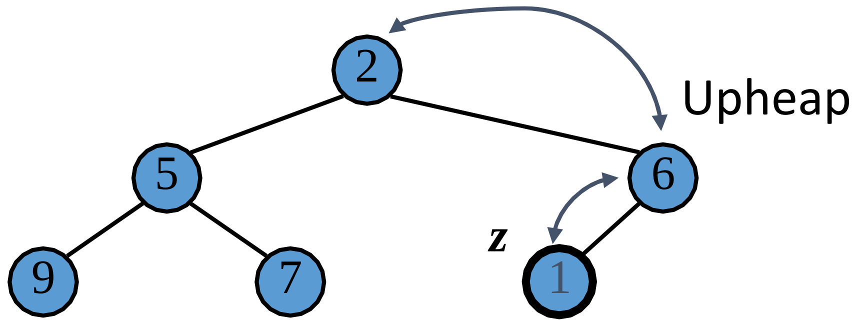
depth	keys
0	1
1	2
$\boldsymbol{h}-1$	$2^{\boldsymbol{h}-1}$
$\boldsymbol{h}$	1



Heap - insertion

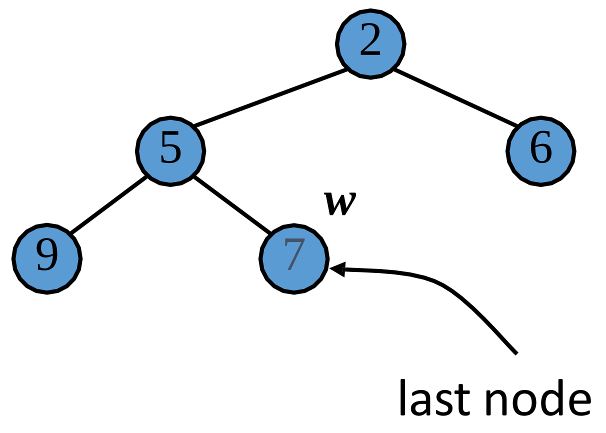
- The insertion algorithm consists of three steps
  - Find the insertion node ***z*** (the new last node)
  - Store ***k*** at ***z***
  - Perform Upheap
- Upheap restores the heap-order property by swapping ***k*** along an upward path from the insertion node
- Upheap terminates when the key ***k*** reaches the root or a node whose parent has a key smaller than or equal to ***k***
- Since a heap has height ***O***(log ***n***), Upheap runs in ***O***(log ***n***) time

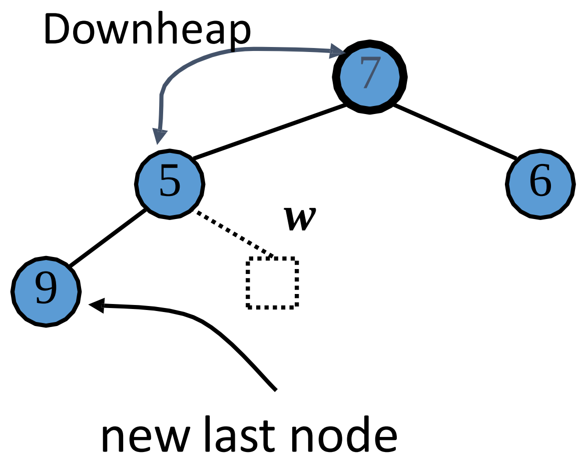




Heap – remove minimum key

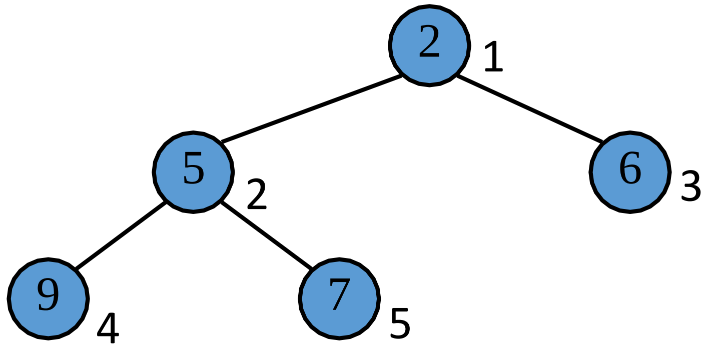
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node ***w***
  - Remove ***w***
  - Restore the heap-order property (Downheap)
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm Downheap restores the heap-order property by swapping key k (with the smallest key of its two children) along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height O(log n), Downheap runs in O(log n) time

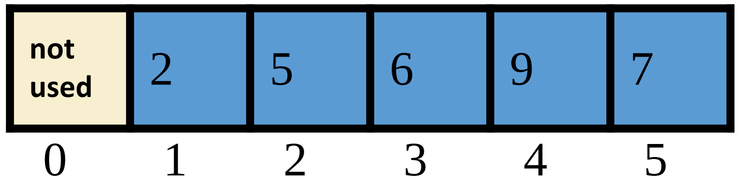




Heap – array implementation

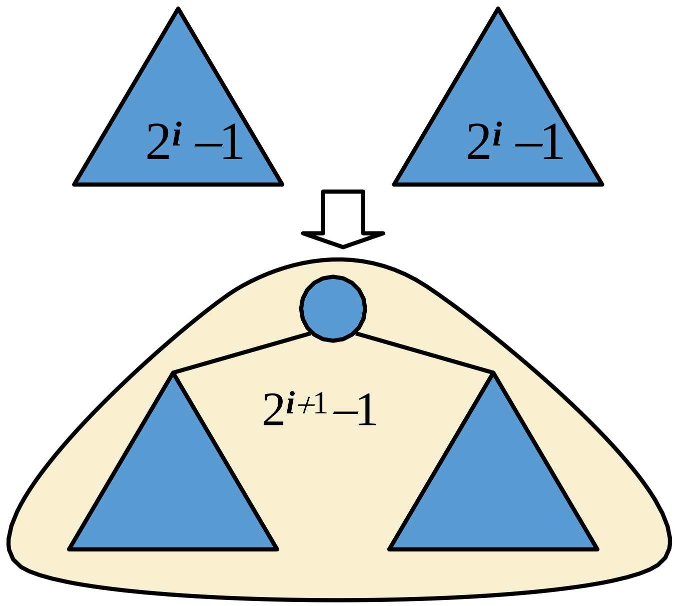
- We represent a heap with $\boldsymbol{n}$ keys on an array of size $\boldsymbol{n}+1$
- For the node at rank $\boldsymbol{i}$
  - the left child is at rank $2\boldsymbol{i}$
  - the right child is at rank $2\boldsymbol{i}+1$
  - its parent is at rank $\lfloor \boldsymbol{i}/2 \rfloor$
- Links between nodes are not physically stored
- The cell at rank 0 is not used
- Operation insert corresponds to inserting at rank $\boldsymbol{n}+1$
- Operation removeMin corresponds to removing at rank 1
- The key removed (min) is inserted at the end of the sorted list
- removeMin and insert take O(log n)

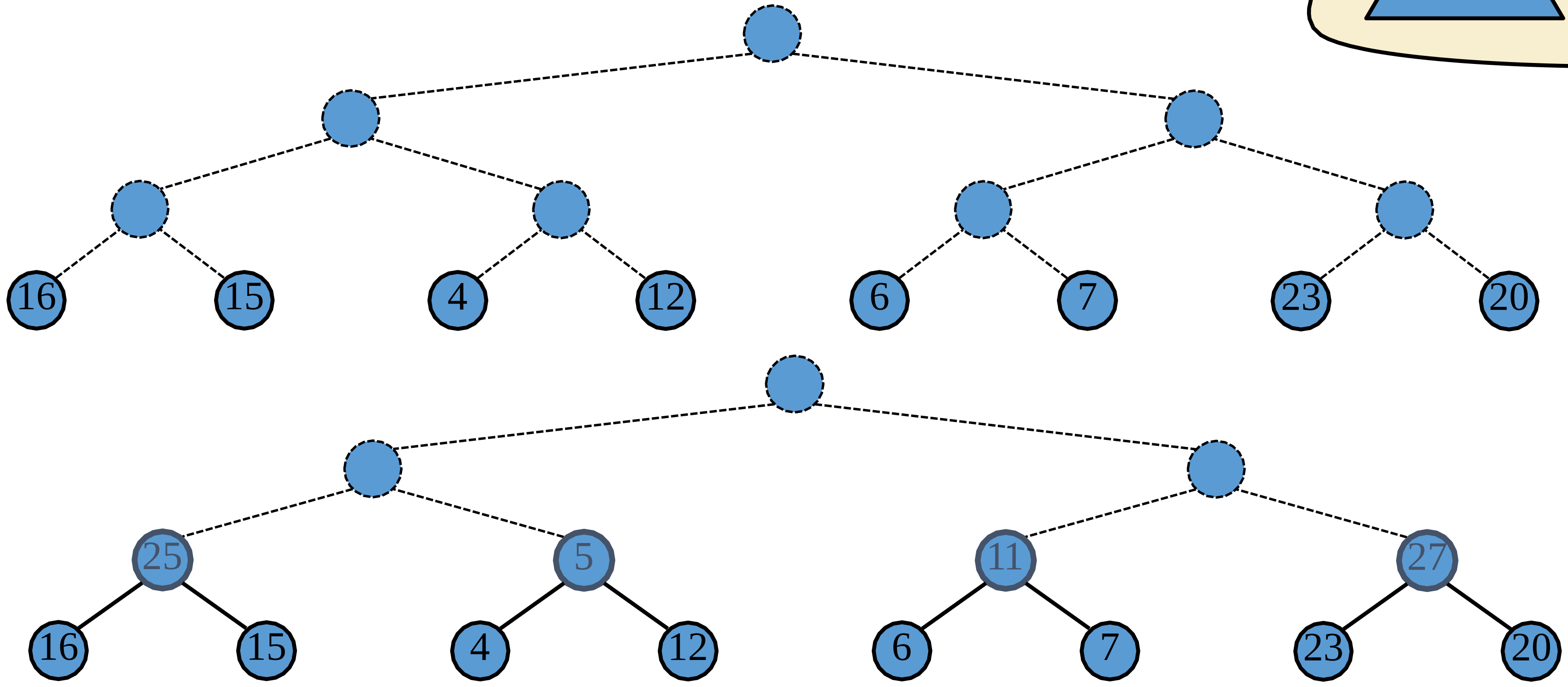




Heap – bottom-up construction

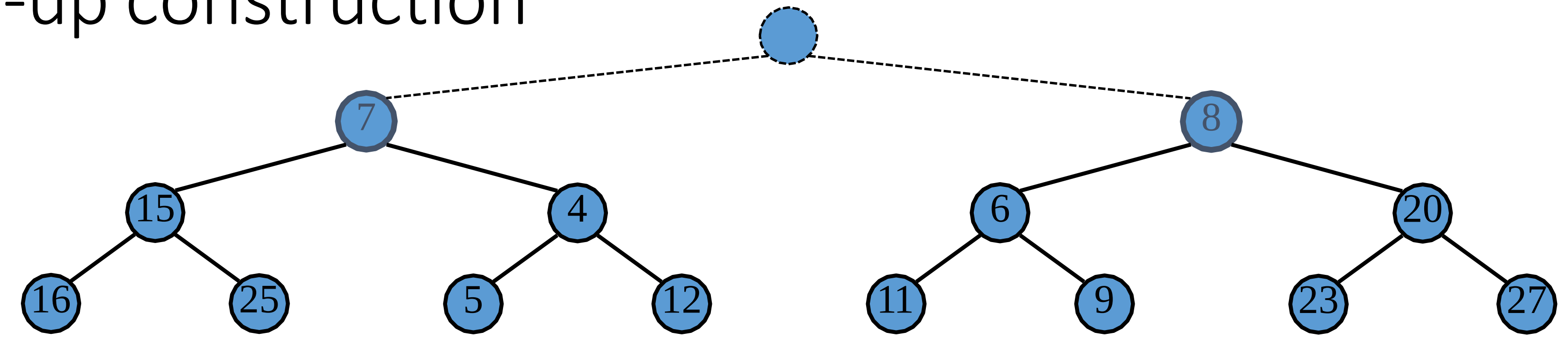
- We can construct a heap storing n keys using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys
- The algorithm runs in O(n)

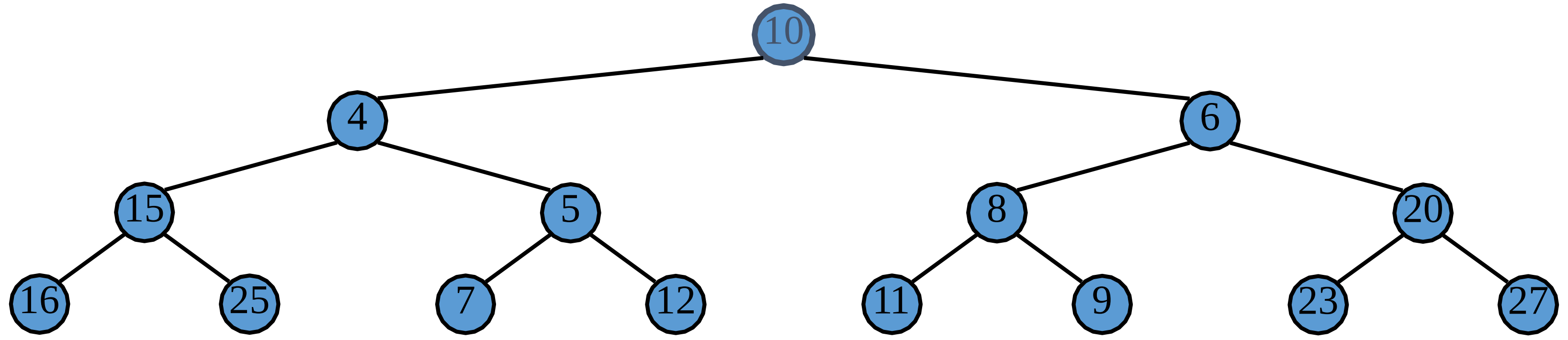


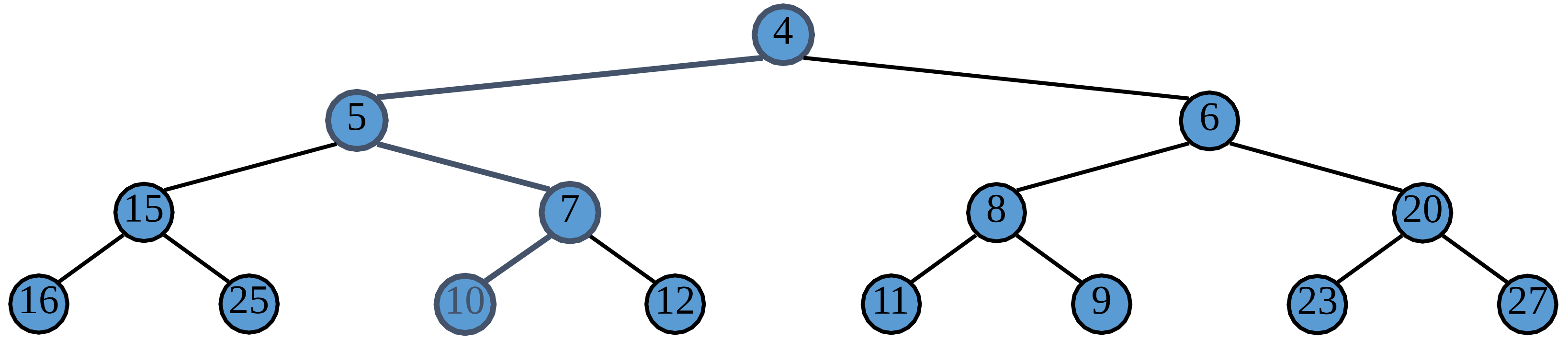


```
Algorithm BottomUpHeap(S)
Input: Seq. S with n = 2^h-1 keys
Output: A heap H
if S is empty then
   return empty heap
k ← S[0]
Split S[1..n-1] into S_1 and S_2
H_1 ← BottomUpHeap(S_1)
H_2 ← BottomUpHeap(S_2)
Create H with k as the root, H_1 left child and H_2 as right child
Downheap(H, root)
return H
```

Bottom-up construction



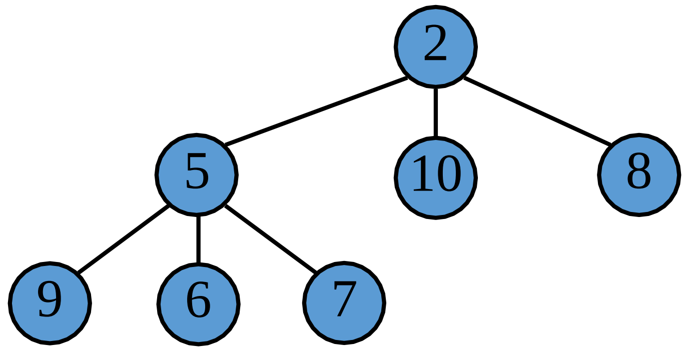




d-Heaps

- A d-Heap is a generalization of the binary heap
- d-Heaps are shallower: height is h = O($\log_d$ n)
- Insertion and removeMin take both O(log n)
- $\log_d n = \frac{\log_2 n}{\log_2 d}$, where $\log_2 d$ is a constant!
- Experimental studies suggest that the 4-Heap may outperform the binary heap

Example: a 3-Heap



Other heaps

- Leftist heap
  - It has the structural and ordering properties of the binary heaps
  - Leftist heaps are not necessarily balanced
- Skew heap
  - It is a self-adjusting version of the leftist heap
  - Heap operations can go up to O(n) worst case
  - But have O(log n) amortized time
- Binomial queue
  - A binomial queue is not a heap in the strict sense but a *collection* of heaps or a *forest*
- Fibonacci heap
  - It is also a *collection* of heaps
  - removeMin and remove have amortized time O(log n)
  - Other operations have amortized time O(1)
- Further reading:
  - Sections 6.5, 6.6, 6.7 and 11.4 of [3]

Heaps in Java

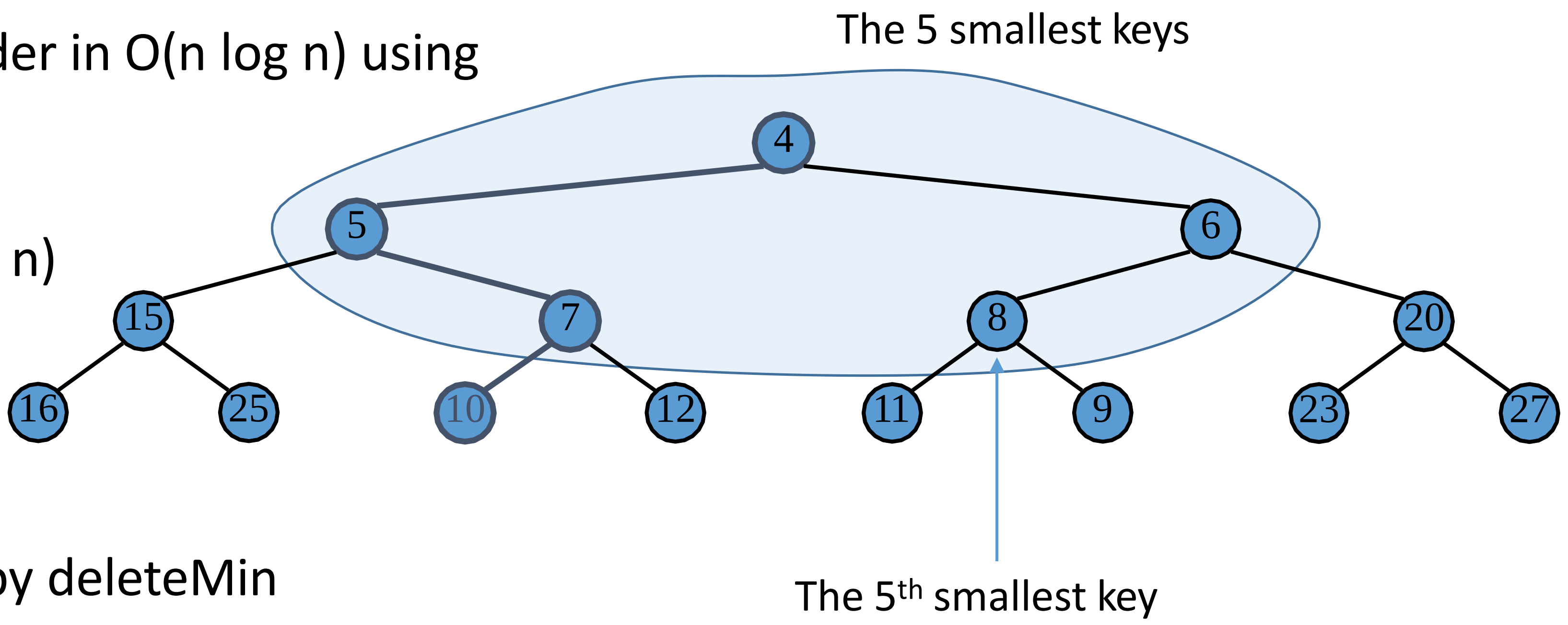
- PriorityQueue class implements a heap ADT
- It's part of java.util
- Allows to specify comparator in constructor

```
public PriorityQueue(int initialCapacity,
      Comparator<? super E> comparator)
```

- Insert a new object: **add**(**E** e)
- Remove the minimum: **poll**()
- Documentation
  - http://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

Application 1: Selection problem

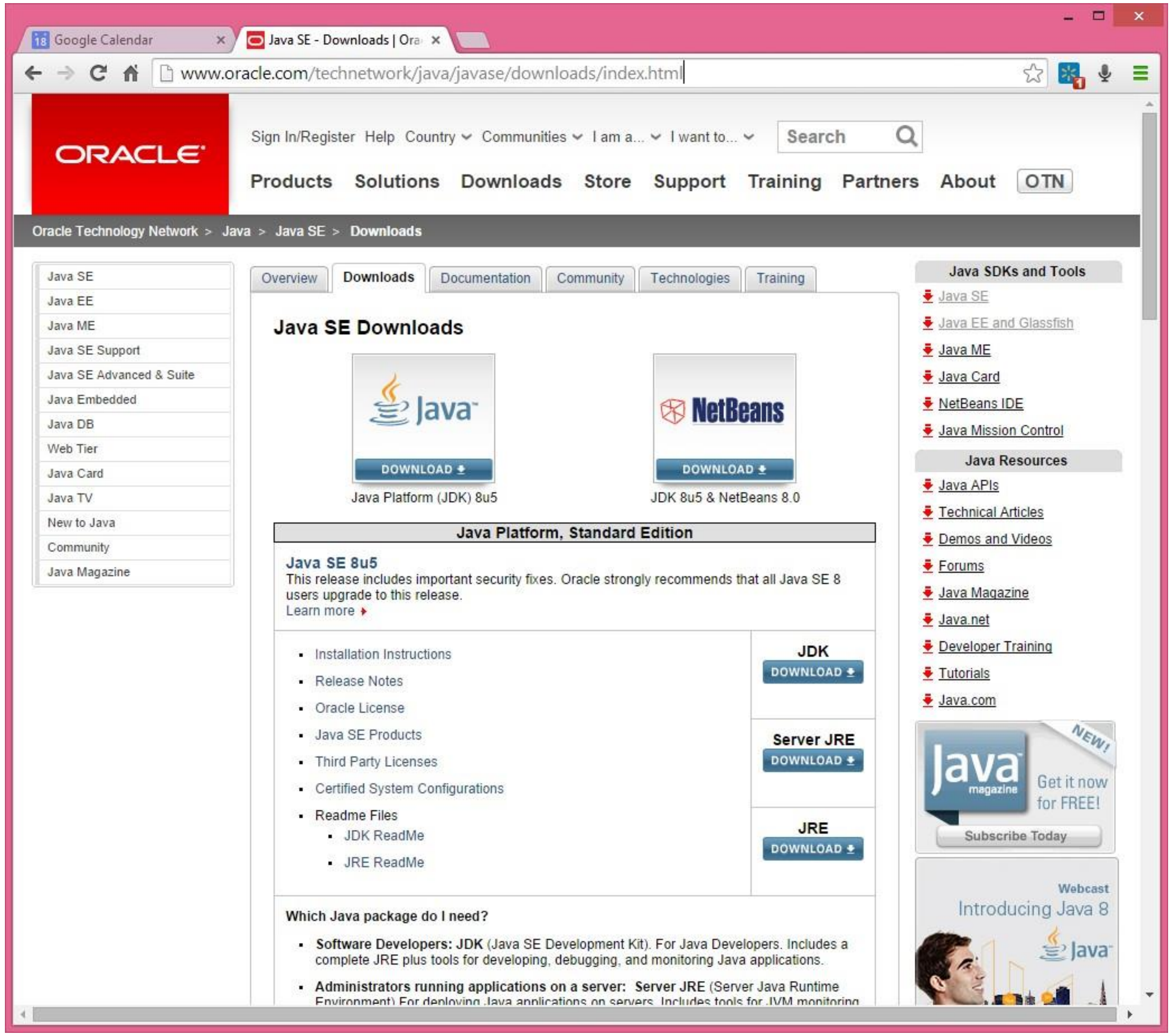
- Problem: given an unsorted list S of n keys, find the kth smallest key (or the k smallest keys)
- Naïve approach
  - Sort the list in increasing order in O(n log n) using Mergesort or Heapsort
  - Extract the kth key
  - Total running time is O(n log n)



- Heap-based approach
  - Construct a heap in O(n)
  - Extract the kth smallest key by deleteMin
  - Each deleteMin operation takes O(log n)
  - Total running time is O(n) + O(k log n) = O(n + k log n)
  - If k <<< n or a constant, this is a much better approach
- Is there a better algorithm?

Application 2: Counting word frequencies

- Problem: given a set of files (html, emails, etc), compute the frequency of each word in each/all files
- Solution: use a hash table in which the keys are the words
- For every word in the document(s)
  - Search in the hash table
  - If not found, insert it with frequency 1
  - If found, add 1 to its frequency
- Other problems:
  - Find the k most frequent words
  - Rank documents by frequent words
  - Find documents based on some keywords


Google Calendar | Java SE - Downloads | Ora

www.oracle.com/technetwork/java/javase/downloads/index.html

ORACLE

Sign In/Register Help Country Communities I am a... I want to... Search

Products Solutions Downloads Store Support Training Partners About OTN

Oracle Technology Network > Java > Java SE > Downloads

Java SE
Java EE
Java ME
Java SE Support
Java SE Advanced & Suite
Java Embedded
Java DB
Web Tier
Java Card
Java TV
New to Java
Community
Java Magazine

Overview | Downloads | Documentation | Community | Technologies | Training

Java SE Downloads

Java DOWNLOAD — Java Platform (JDK) 8u5

NetBeans DOWNLOAD — JDK 8u5 & NetBeans 8.0

Java Platform, Standard Edition

Java SE 8u5
This release includes important security fixes. Oracle strongly recommends that all Java SE 8 users upgrade to this release.
Learn more

- Installation Instructions
- Release Notes
- Oracle License
- Java SE Products
- Third Party Licenses
- Certified System Configurations
- Readme Files
  - JDK ReadMe
  - JRE ReadMe

JDK DOWNLOAD
Server JRE DOWNLOAD
JRE DOWNLOAD

Which Java package do I need?

- **Software Developers: JDK** (Java SE Development Kit). For Java Developers. Includes a complete JRE plus tools for developing, debugging, and monitoring Java applications.
- **Administrators running applications on a server: Server JRE** (Server Java Runtime Environment) For deploying Java applications on servers. Includes tools for JVM monitoring

Java SDKs and Tools
- Java SE
- Java EE and Glassfish
- Java ME
- Java Card
- NetBeans IDE
- Java Mission Control

Java Resources
- Java APIs
- Technical Articles
- Demos and Videos
- Forums
- Java Magazine
- Java.net
- Developer Training
- Tutorials
- Java.com

Java magazine NEW! Get it now for FREE! Subscribe Today

Webcast Introducing Java 8

Review and Further Reading

- Hashing:
  - Perfect, Cuckoo, hopscotch, universal and extendible hashing: Sections 5.7, 5.8, 5.9 [3]
  - Maps and hashing: Sections 10.1, 10.2 of [2], and Chapter 6 of [1]
  - Sorted maps: Section 10.3 of [2]
- Priority queues
  - Heaps: Section 9.3 of [2]
  - Binary heap: 6.3 of [3]
  - Bottom-up construction: 2.4 of [1]
  - d-Heaps, leftist, skew, binomial, Fibonacci heaps: 6.5, 6.6, 6.7, 6.8, 11.4 of [3]
- Hash tables in Java:
  - http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
- Heaps in Java:
  - http://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html
- Binary search in Java:
  - http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

References

1. Algorithm design and applications by M. Goodrich and R. Tamassia, Wiley, 2015.
2. Data Structures and Algorithms in Java, 6th Edition, by M. Goodrich and R. Tamassia, Wiley, 2014.
3. Data Structures and Algorithm Analysis in Java, 3rd Edition, by M. Weiss, Addison-Wesley, 2012.
4. Algorithm Design by J. Kleinberg and E. Tardos, Addison-Wesley, 2006.
5. Introduction to Algorithms, 2nd Edition, by T. Cormen et al., McGraw-Hill, 2001.
6. Java documentation: http://docs.oracle.com/javase/8/docs/api/overview-summary.html

Lab – Practice

- Hashing:
  - Create three hash tables that contain 1,000,000 random keys: Cuckoo, QuadraticProbing, SeparateChaining
  - Do 100,000 searches and record the time of all searches and the average time for each search. Use random keys.
  - Do the same with removals.
  - Compare the running times of the three hash tables, and with the asymptotic notation discussed in class.
- Priority queues:
  - Create a Binary Heap
  - Insert 1,000,000 random integer keys, and record the total and average time per insertion.
  - Do removeMin and immediately after insert a random key. Do this 100,000 times.
  - Keep track of the average time that it takes for removeMin and insert (separately).
  - Compare your results with the time complexities for these operations.

Exercises

1. In the hash table of page 6, insert two keys that yield a single cluster.
2. What are the advantages of quadratic probing? Can you find an example of keys that give a single cluster?
3. Discuss the conditions under which a hash table with linear probing will insert/search in O(1) average time. What is the worst case running of these operations?
4. What are the conditions for double hashing?
5. Write a program that given a set of html files, it creates a hash table with the frequencies of each word in the files.
6. For the problem in #5, write a program that finds the top 10 frequencies in linear time, using a Heap.
7. *Write a program that spellchecks a given document using a dictionary implemented in hash table. Show an error when the word is not found and ask the user if he/she wanted to add the word. If so, add the word to the hash table.
8. Describe the comparator ADT used in priority queues.
9. Discuss the asymptotic running times of the main operations in a priority queue implemented on a Heap.
10. *Write an algorithm for the insertion and removeMin operations in a Binary heap.
11. Write a program that implements the binary heap on an array.
12. Write a program that implements the bottom-up construction of the binary heap on an array. Assume an arbitrary number of keys, i.e., not necessarily $2^h - 1$
13. Show that BottomUpHeap runs in O(n). Not a formal proof but a valid justification is enough.
14. Explain how the insertion and removeMin operations in a d-Heap perform in O(log n) time. Is it the same for a binary heap? Why?
15. If using a d-Heap, describe a strategy to find the best value of d.
16. Describe the main features of leftist, skew, and Fibonacci heaps and their main differences with the binomial heap.
17. *Implement both the naïve and heap-based approaches for the Selection problem (page 26), using a binomial heap. Compare the running times of both algorithms.
18. *Design an algorithm that given a Web page, it counts the number of unique words in the page. Assume the page has been parsed and hence the input to your algorithm is a sequence of consecutive words in the page.