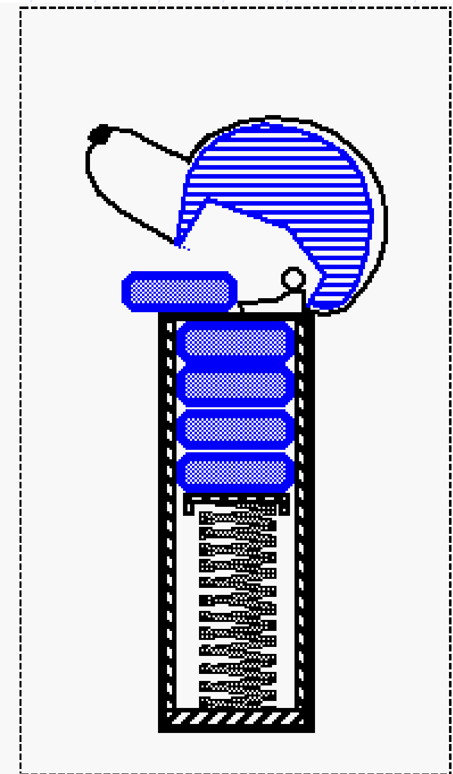
Basic Data Structures

Priority Queues, Heaps, Dictionaries, and Hash Tables



Priority Queues

Sell	100	IBM	$122
Sell	300	IBM	$120
Buy	500	IBM	$119
Buy	400	IBM	$118

Priority Queue ADT

- A priority queue stores a collection of items
- An item is a pair (key, element)
- Main methods of the Priority Queue ADT
  - insertItem(k, o) inserts an item with key k and element o
  - removeMin() removes the item with smallest key and returns its element
- Additional methods
  - minKey() returns, but does not remove, the smallest key of an item
  - minElement() returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Total Order Relation

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct items in a priority queue can have the same key
- Mathematical concept of total order relation $\leq$
  - The comparison rule is defined for every pair x and y.
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- Methods of the Comparator ADT, all with Boolean return type
  - isLessThan(x, y)
  - isLessThanOrEqualTo(x,y)
  - isEqualTo(x,y)
  - isGreaterThan(x, y)
  - isGreaterThanOrEqualTo(x,y)
  - isComparable(x)

Sorting with a Priority Queue

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insertItem(e, e) operations
  2. Remove the elements in sorted order with a series of removeMin() operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort*(*S*, *C*)
  Input sequence *S*, comparator *C* for the elements of *S*
  Output sequence *S* sorted in increasing order according to *C*
  P ← priority queue with comparator *C*
  while ¬*S.isEmpty* ()
    e ← *S.remove* (*S. first* ())
    P.insertItem(*e*, *e*)
  while ¬*P.isEmpty*()
    e ← *P.removeMin*()
    S.insertLast(*e*)

Sequence-based Priority Queue

- Implementation with an unsorted sequence
  - Store the items of the priority queue in a list-based sequence, in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin, minKey and minElement take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted sequence
  - Store the items of the priority queue in a sequence, sorted by key
- Performance:
  - insertItem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey and minElement take $O(1)$ time since the smallest key is at the beginning of the sequence

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insertItem operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

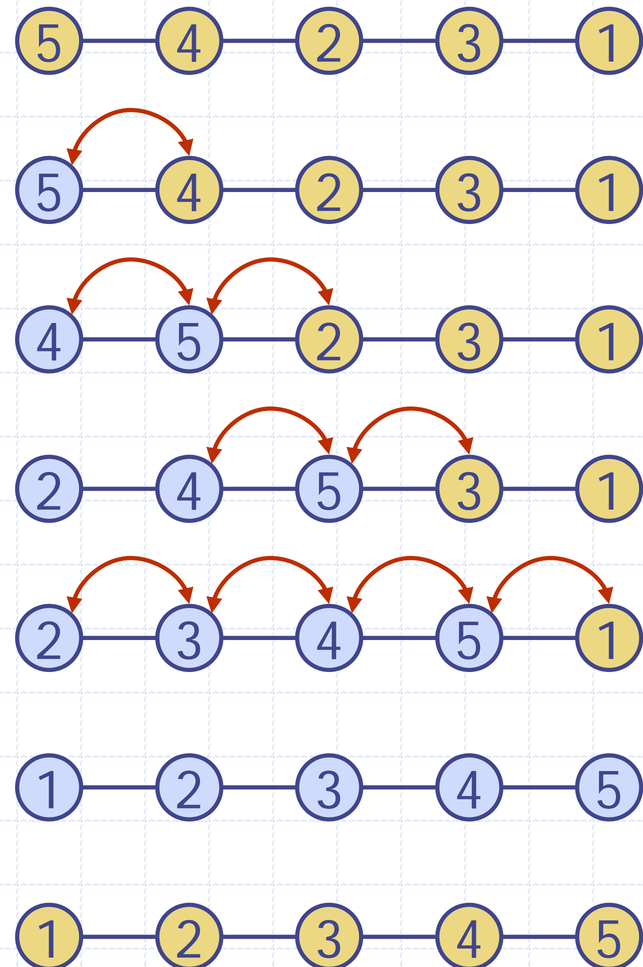
- Selection-sort runs in $O(n^2)$ time

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insertItem operations takes time proportional to
     $$1 + 2 + \ldots + n$$
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

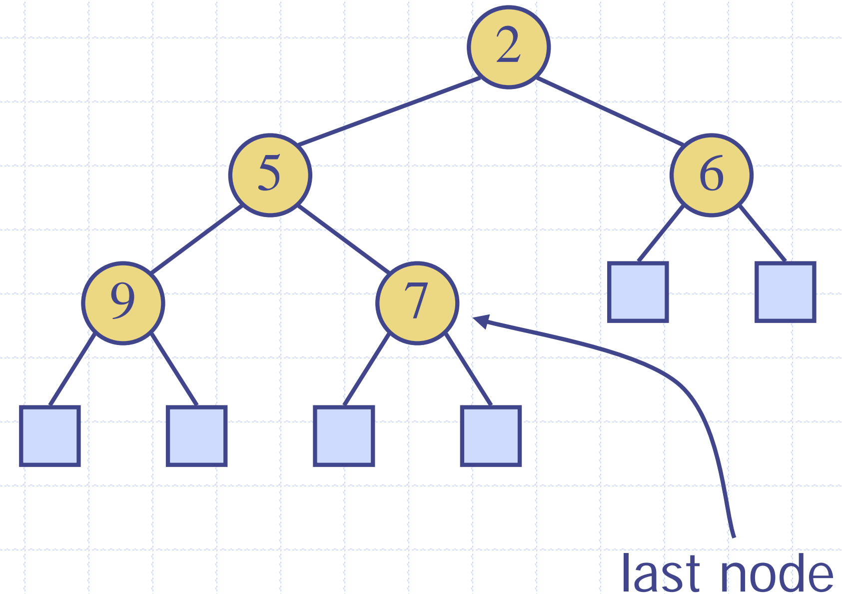
In-place Insertion-sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swapElements instead of modifying the sequence



What is a heap (§2.4.3)

- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root,
    $key(v) \geq key(parent(v))$
  - 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 $h - 1$, the internal nodes are to the left of the external nodes
- The last node of a heap is the rightmost internal node of depth $h - 1$

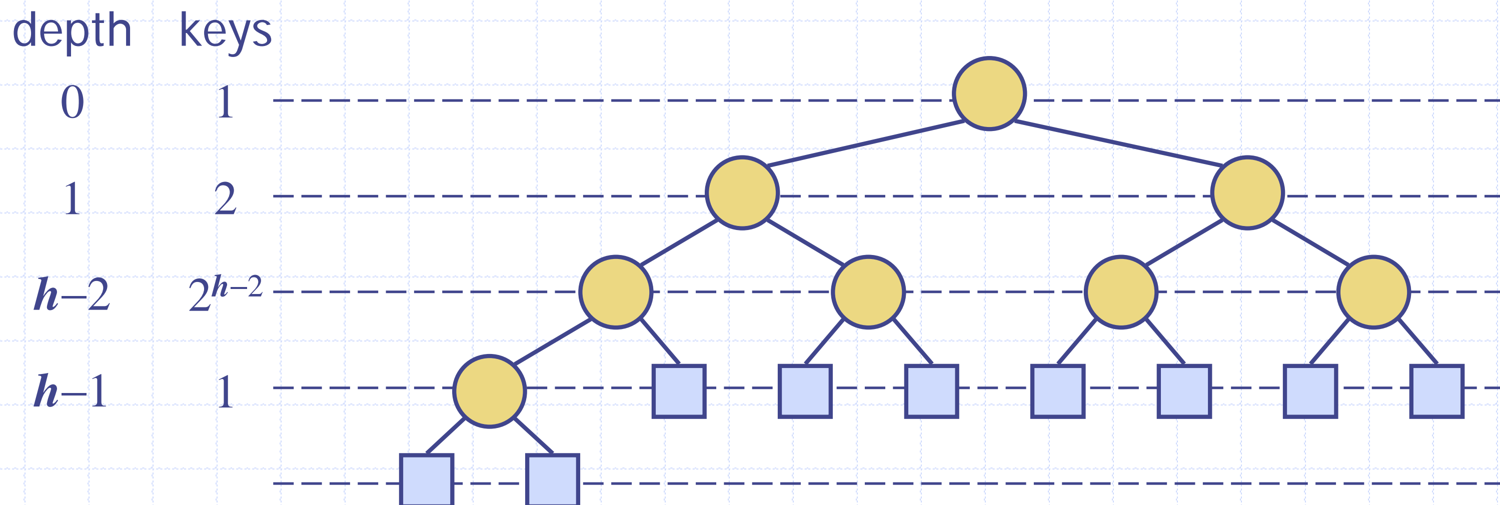


Height of a Heap (§2.4.3)

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

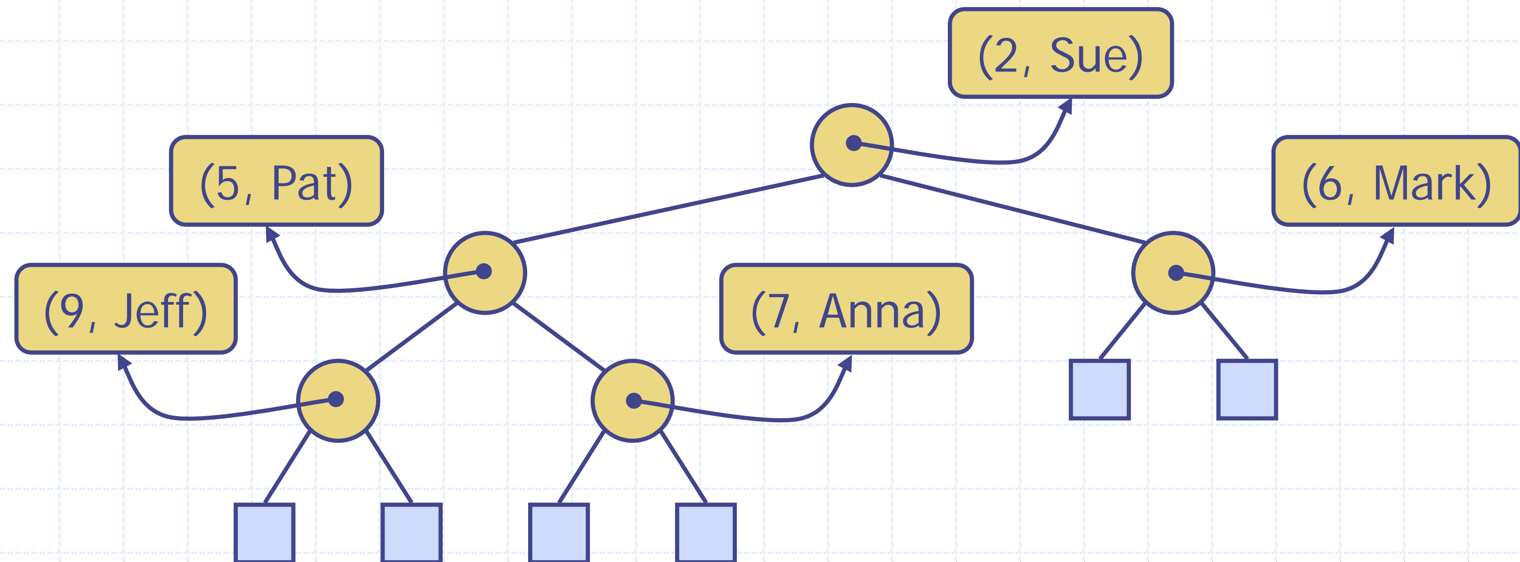
Proof: (we apply the complete binary tree property)

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



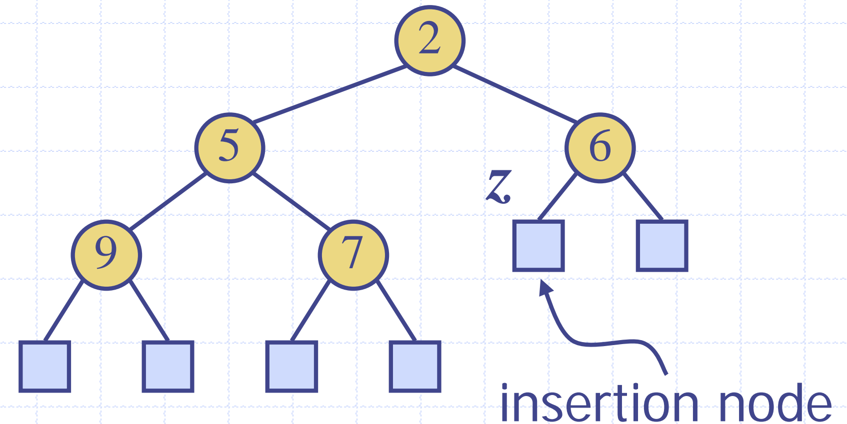
Heaps and Priority Queues

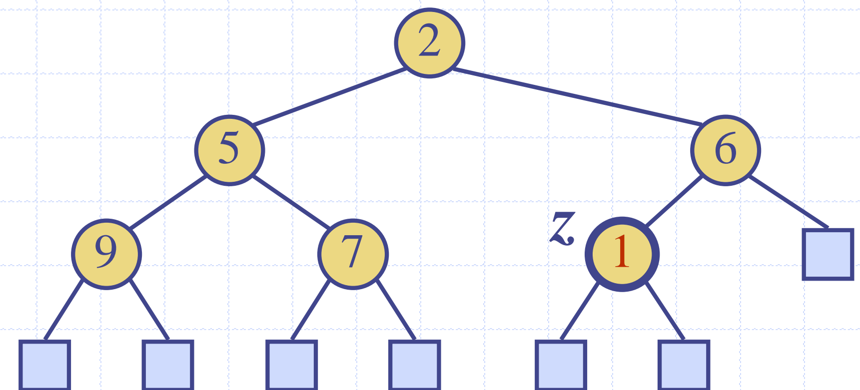
- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the pictures



Insertion into a Heap (§2.4.3)

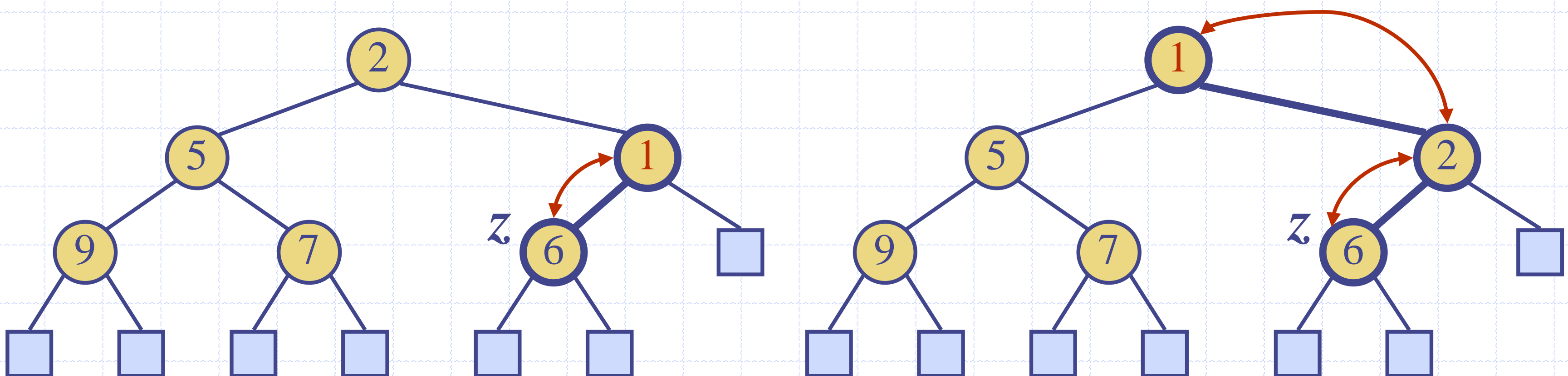
- Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z and expand z into an internal node
  - Restore the heap-order property (discussed next)





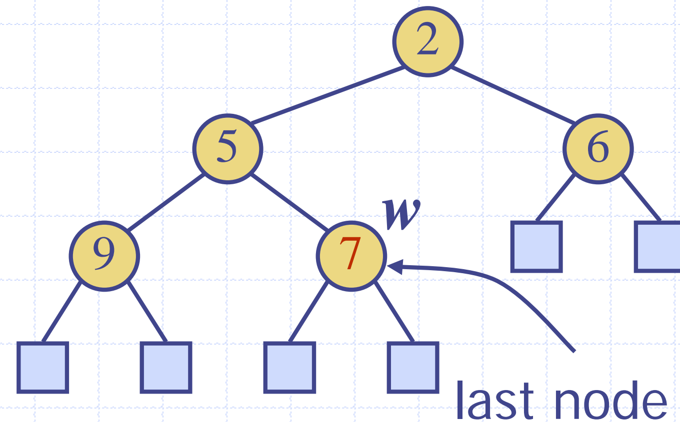
Upheap

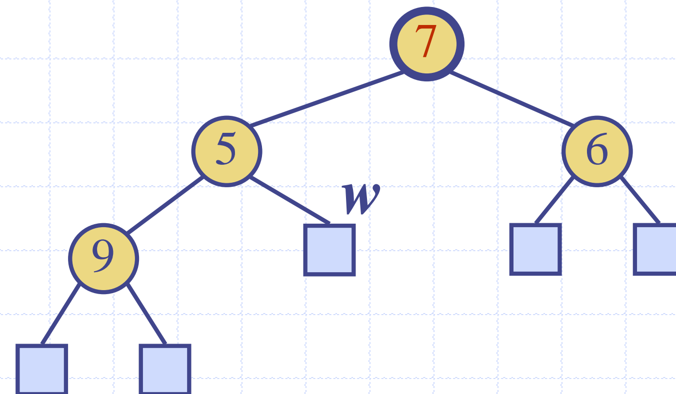
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



Removal from a Heap (§2.4.3)

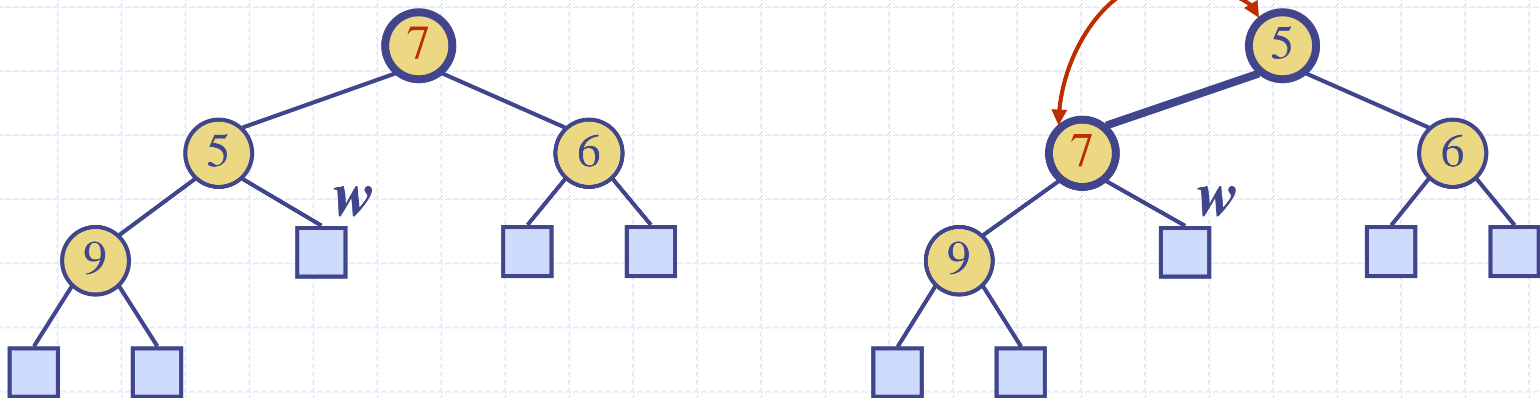
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Compress w and its children into a leaf
  - Restore the heap-order property (discussed next)





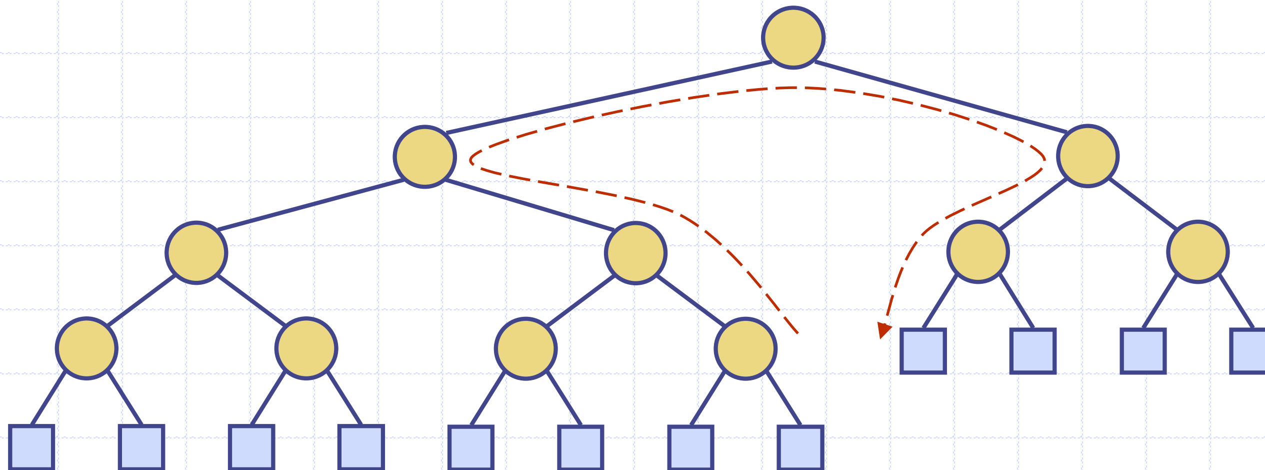
Downheap

- 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 along a downward path from the root
- Upheap 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

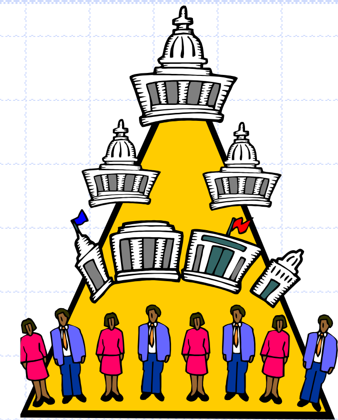


Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - While the current node is a right child, go to the parent node
  - If the current node is a left child, go to the right child
  - While the current node is internal, go to the left child
- Similar algorithm for updating the last node after a removal



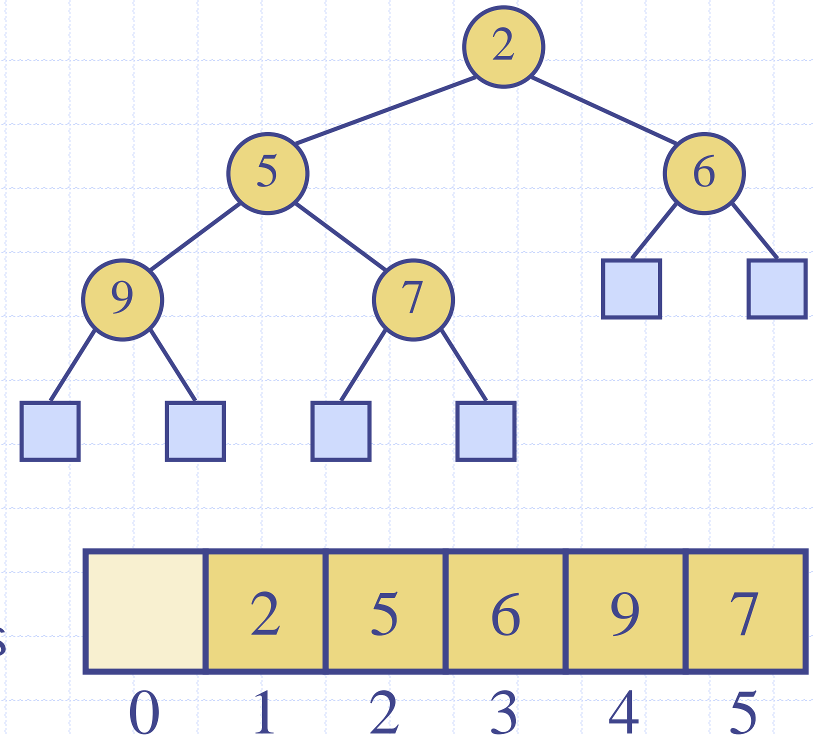
Heap-Sort (§2.4.4)



- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insertItem and removeMin take $O(\log n)$ time
  - methods size, isEmpty, minKey, and minElement take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

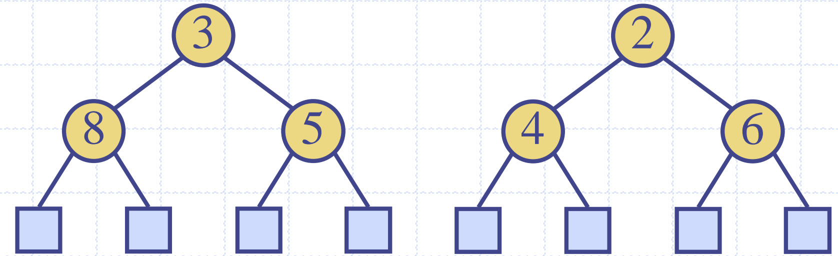
Vector-based Heap Implementation (§2.4.3)

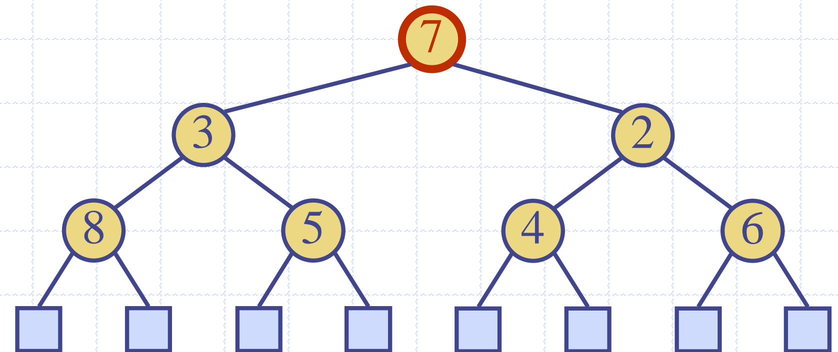
- We can represent a heap with n keys by means of a vector of length $n + 1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- The leaves are not represented
- The cell at rank 0 is not used
- Operation insertItem corresponds to inserting at rank $n + 1$
- Operation removeMin corresponds to removing at rank n
- Yields in-place heap-sort

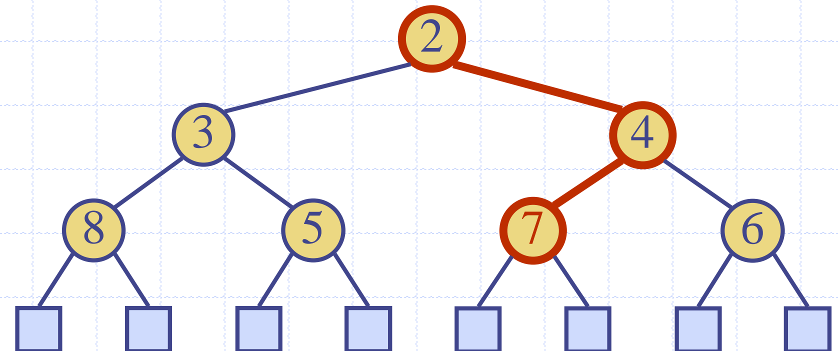


Merging Two Heaps

- We are given two two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property

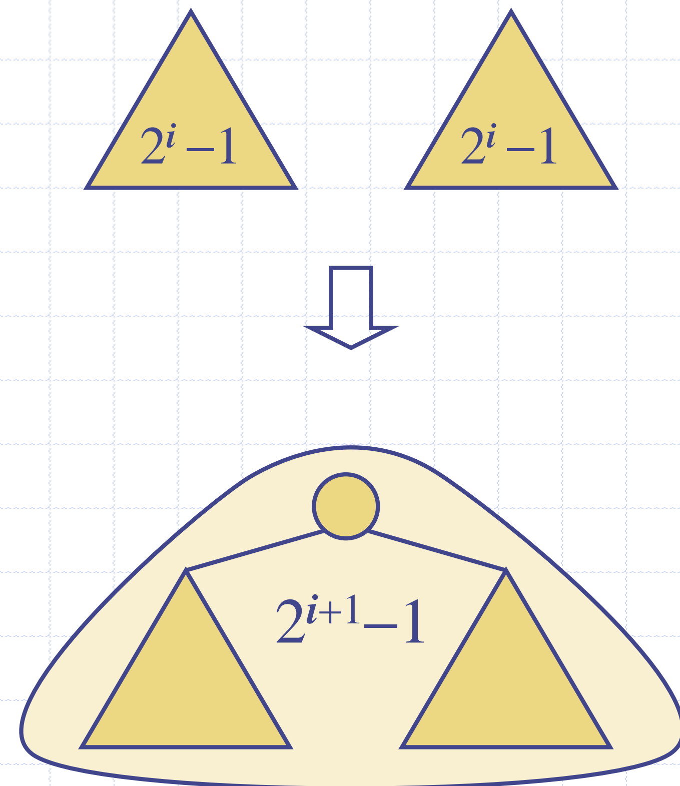




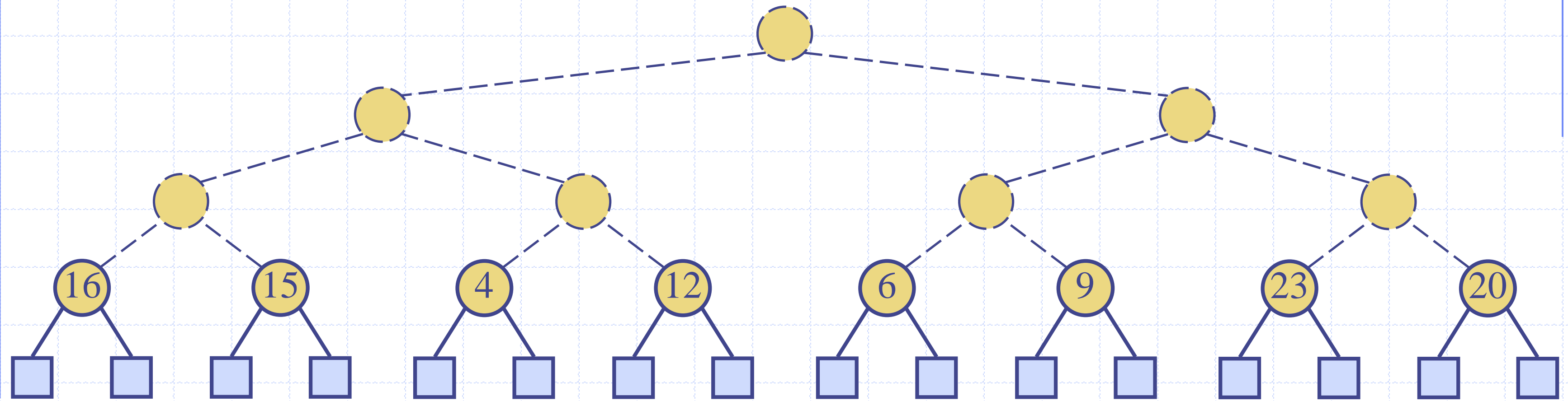


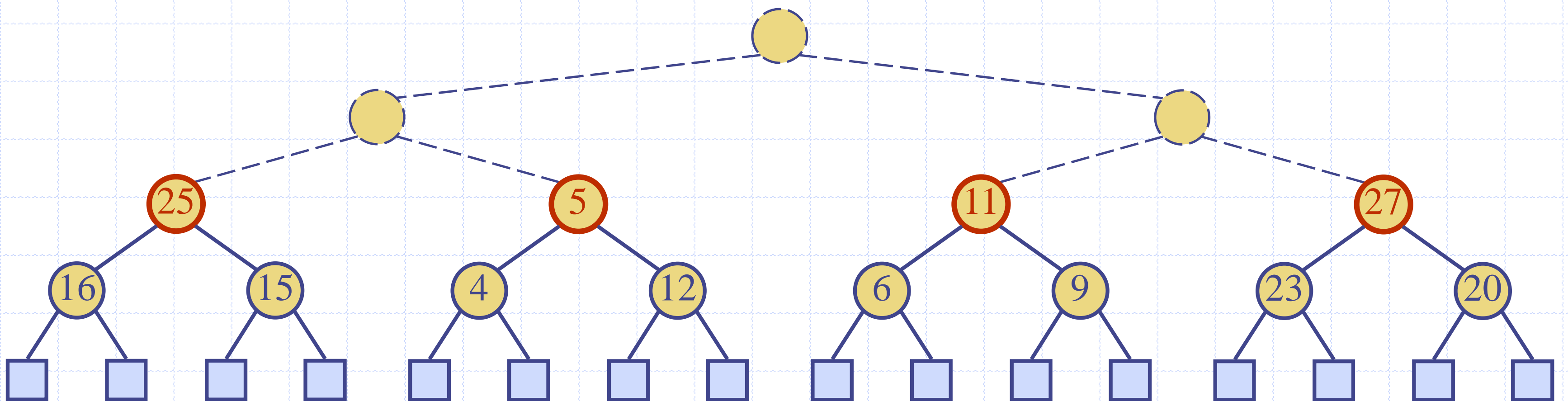
Bottom-up Heap Construction

- We can construct a heap storing n given keys in 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

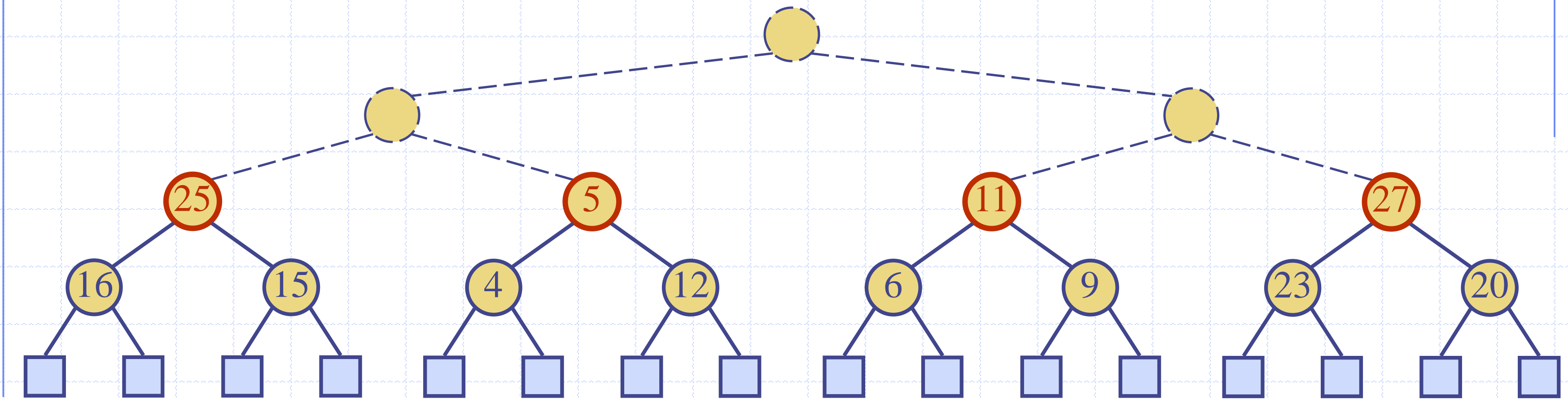


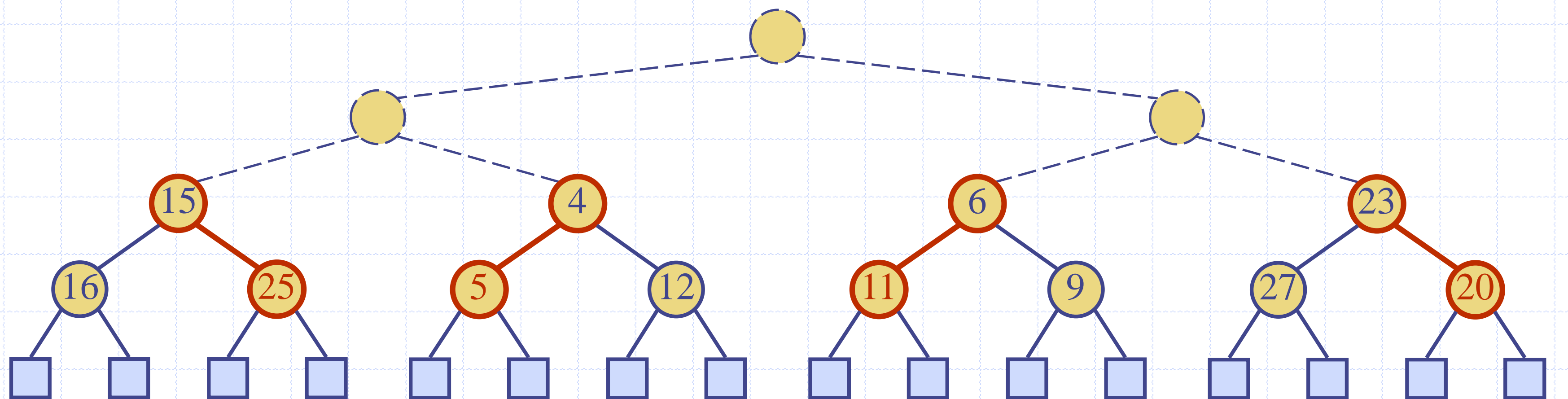
Example



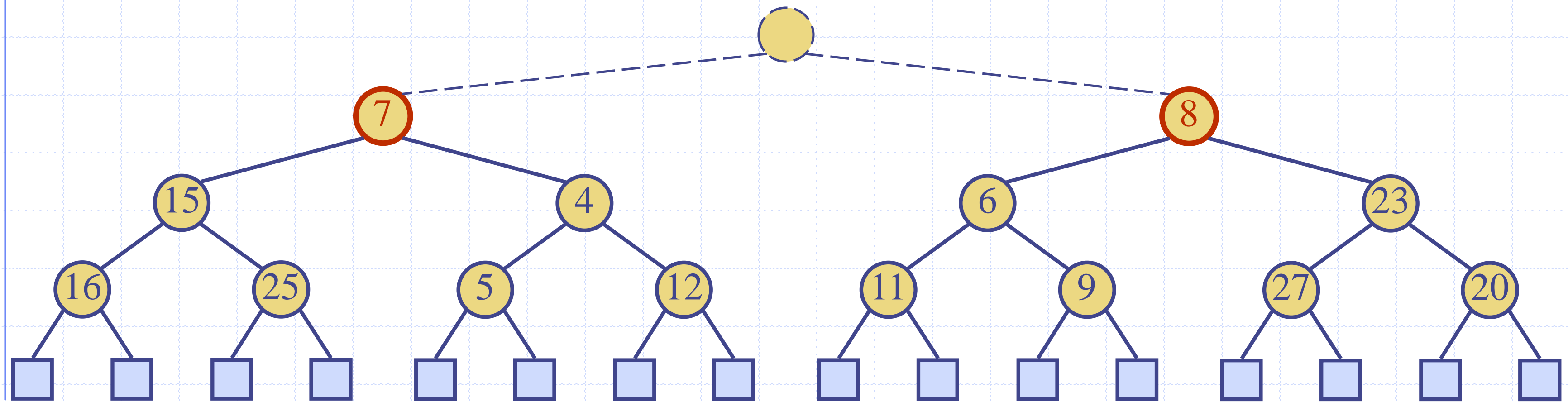


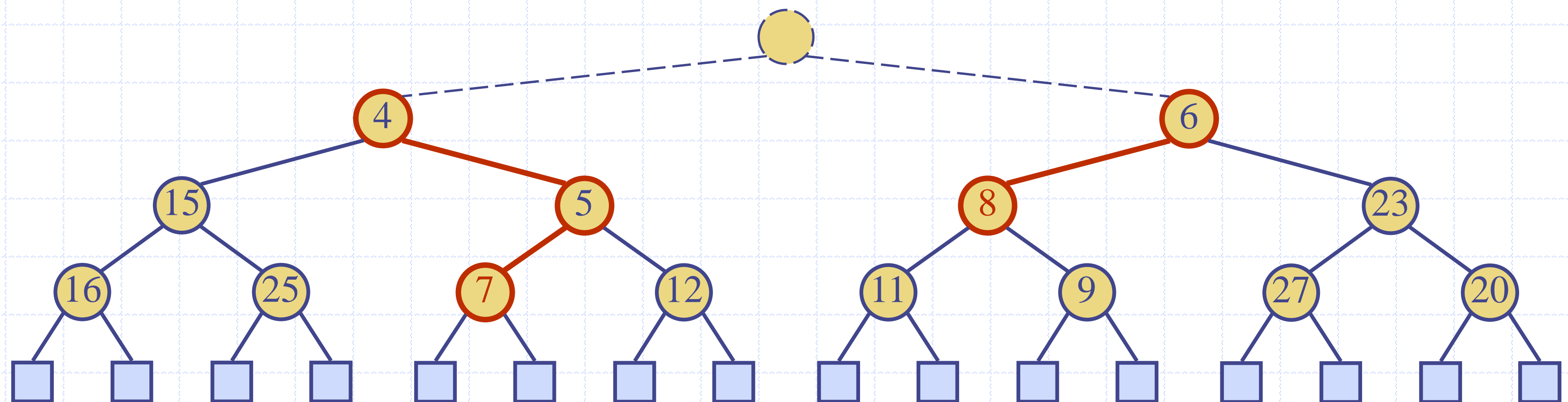
Example (contd.)



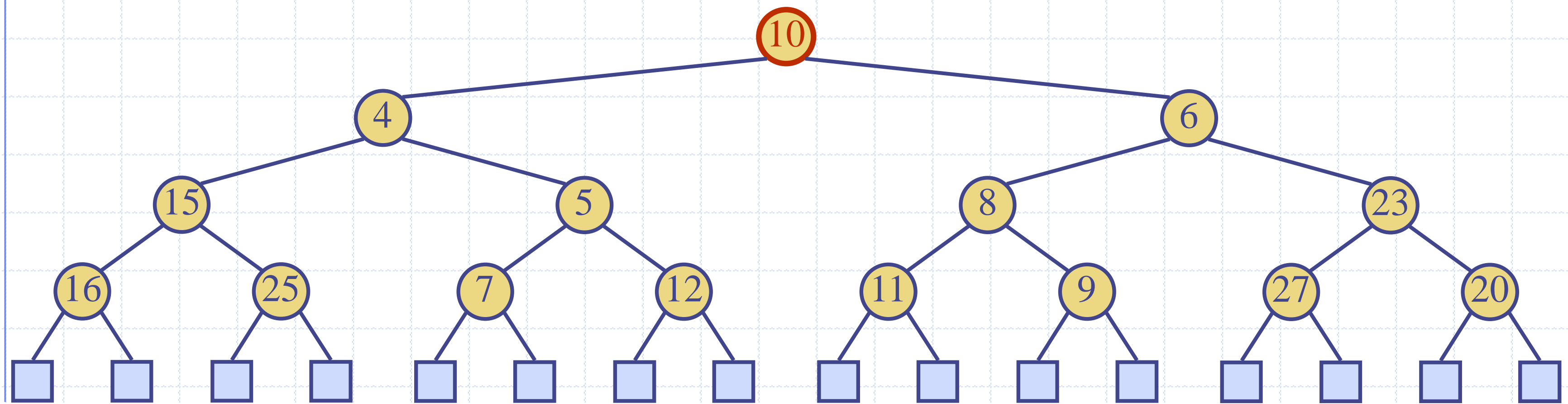


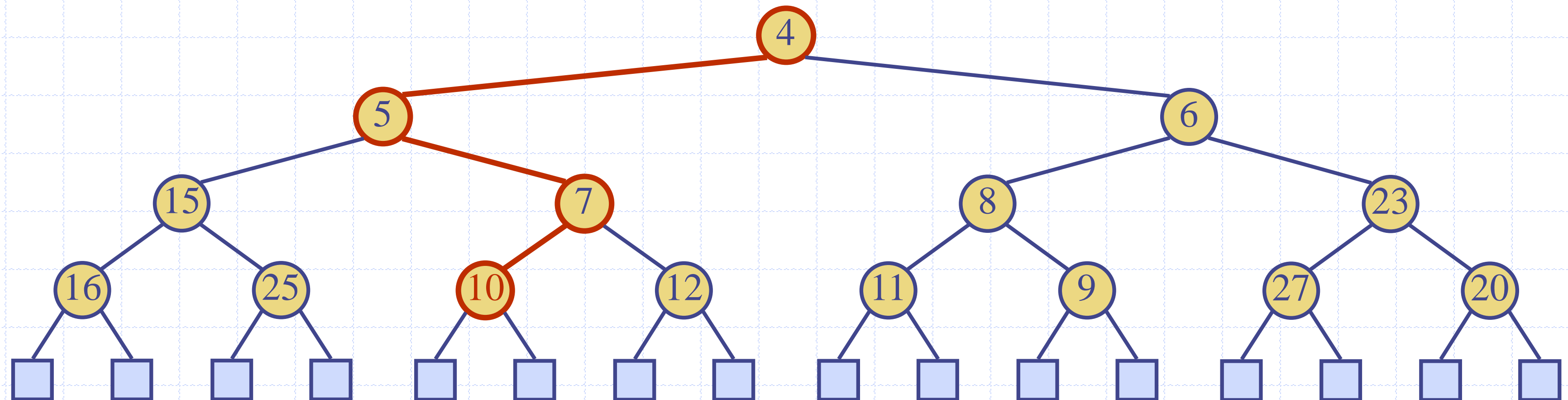
Example (contd.)



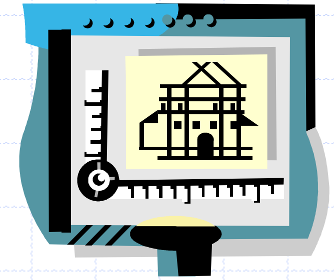


Example (end)

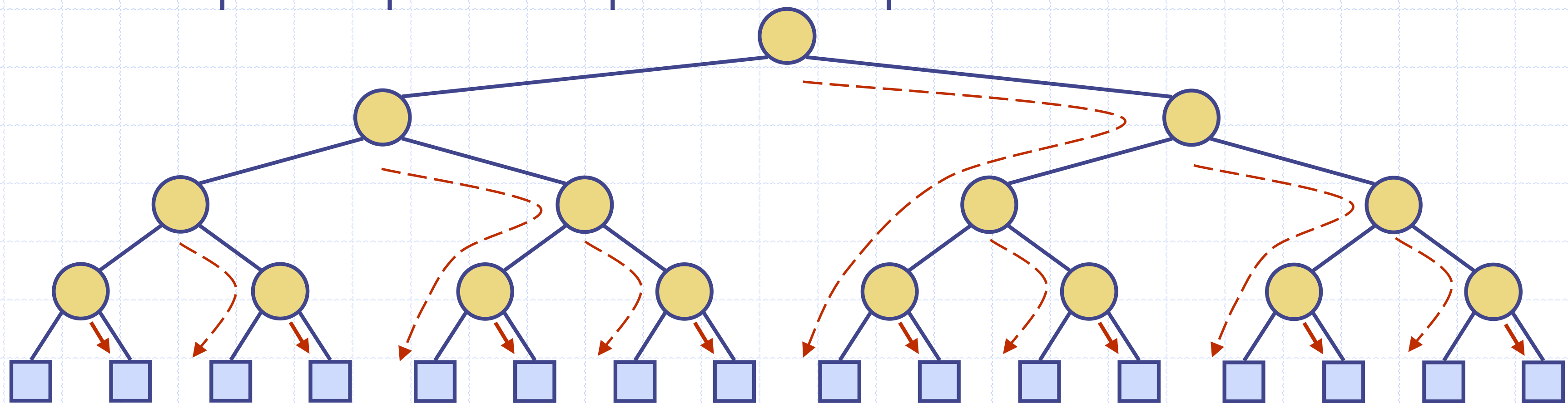




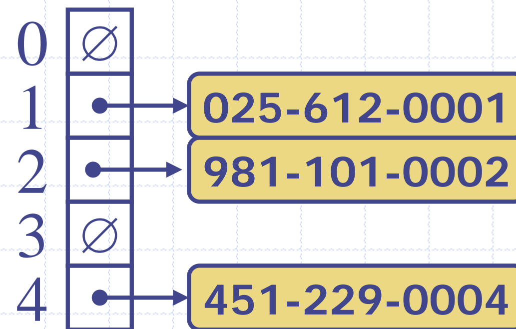
Analysis



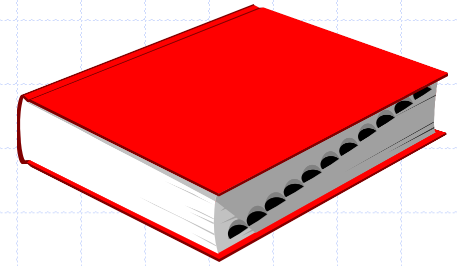
- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort



Dictionaries and Hash Tables



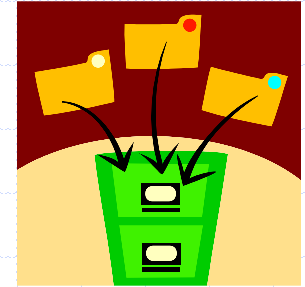
Dictionary ADT (§2.5.1)



- The dictionary ADT models a searchable collection of key-element items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)

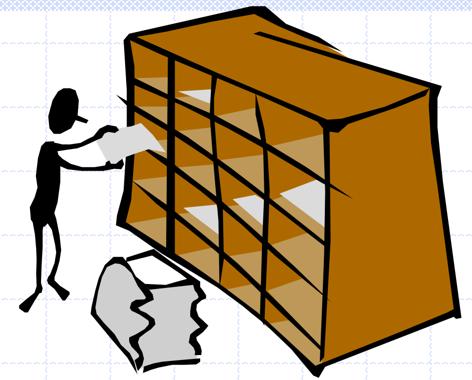
- Dictionary ADT methods:
  - findElement(k): if the dictionary has an item with key k, returns its element, else, returns the special element NO_SUCH_KEY
  - insertItem(k, o): inserts item (k, o) into the dictionary
  - removeElement(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns the special element NO_SUCH_KEY
  - size(), isEmpty()
  - keys(), elements()

Log File (§2.5.1)



- A log file is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked lists or a circular array), in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - findElement and removeElement take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Hash Functions and Hash Tables (§2.5.2)



- A hash function h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example:

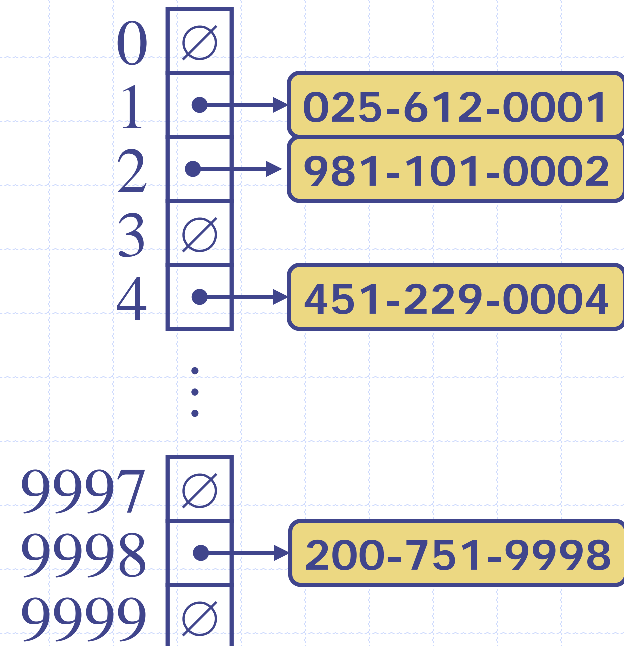
  $$h(x) = x \bmod N$$

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

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a dictionary with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

- We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10,000$ and the hash function $h(x) = \text{last four digits of } x$



Hash Functions (§ 2.5.3)

- A hash function is usually specified as the composition of two functions:

  Hash code map:

  h_1: keys $\rightarrow$ integers

  Compression map:

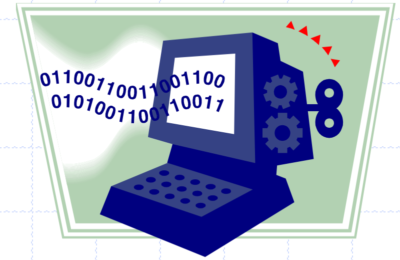
  h_2: integers $\rightarrow [0, N-1]$

- The hash code map is applied first, and the compression map is applied next on the result, i.e.,

  $$h(x) = h_2(h_1(x))$$

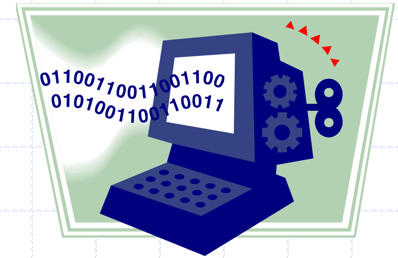
- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Code Maps (§2.5.3)



- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Code Maps (cont.)

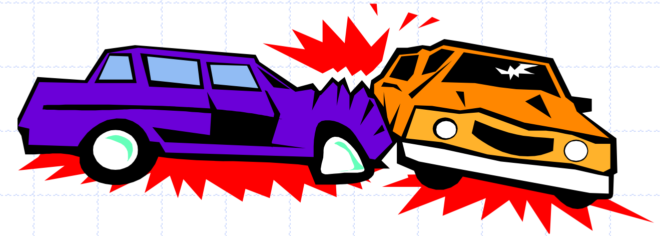


- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $$a_0 a_1 \ldots a_{n-1}$$
  - We evaluate the polynomial
    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots \ldots + a_{n-1} z^{n-1}$$
    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \ldots, n-1)$$
- We have $p(z) = p_{n-1}(z)$

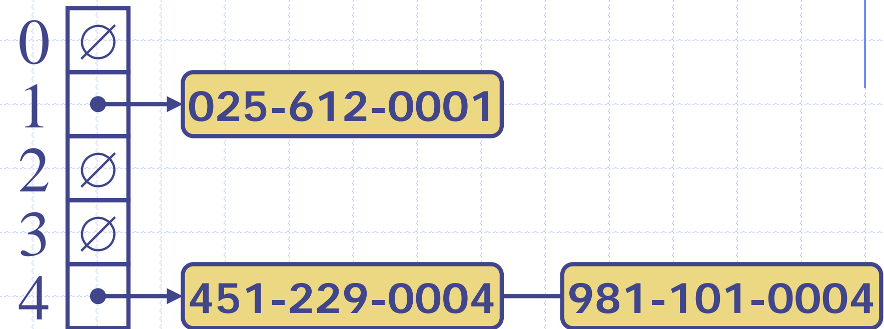
Compression Maps (§2.5.4)

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Collision Handling (§ 2.5.5)



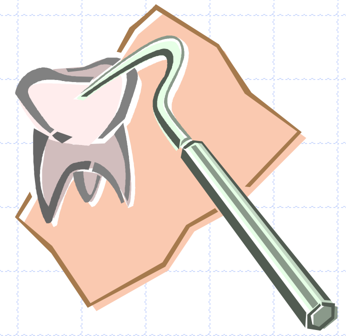
- Collisions occur when different elements are mapped to the same cell



- **Chaining**: let each cell in the table point to a linked list of elements that map there

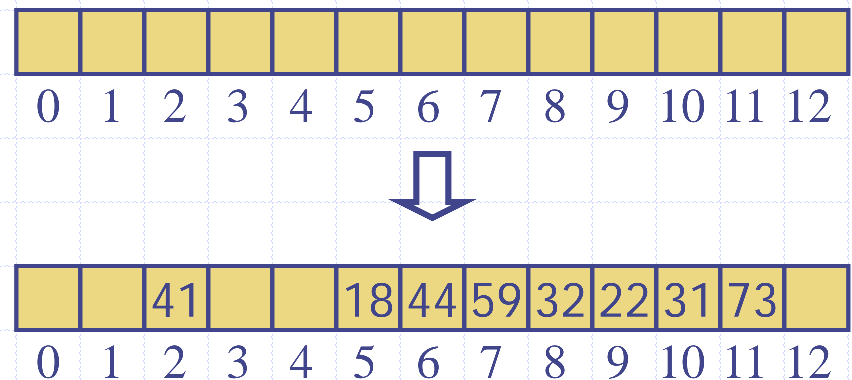
- Chaining is simple, but requires additional memory outside the table

Linear Probing (§2.5.5)

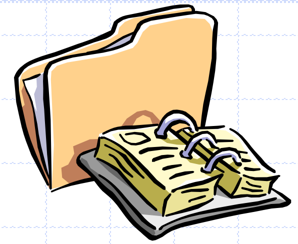


- Open addressing: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

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



Search with Linear Probing



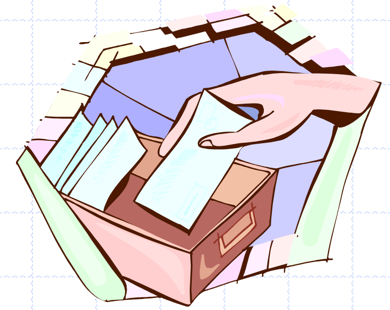
- Consider a hash table A that uses linear probing
- findElement(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Algorithm *findElement*(k)
  $i \leftarrow h(k)$
  $p \leftarrow 0$
  repeat
    $c \leftarrow A[i]$
    if $c = \varnothing$
      return *NO_SUCH_KEY*
    else if $c.key() = k$
      return $c.element()$
    else
      $i \leftarrow (i + 1) \bmod N$
      $p \leftarrow p + 1$
  until $p = N$
  return *NO_SUCH_KEY*

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- removeElement(k)
  - We search for an item with key k
  - If such an item (k, o) is found, we replace it with the special item ***AVAILABLE*** and we return element o
  - Else, we return ***NO_SUCH_KEY***
- insert Item(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores ***AVAILABLE***, or
    - N cells have been unsuccessfully probed
  - We store item (k, o) in cell i

Double Hashing



- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

$$(i + jd(k)) \bmod N$$

for $j = 0, 1, \ldots, N - 1$

- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Common choice of compression map for the secondary hash function:

$$d_2(k) = q - k \bmod q$$

where

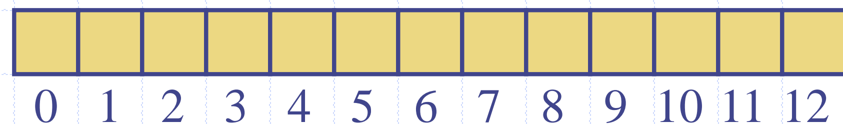
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are

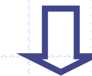
$$1, 2, \ldots, q$$

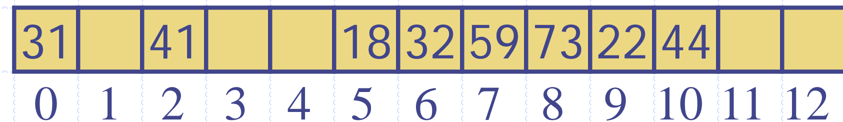
Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

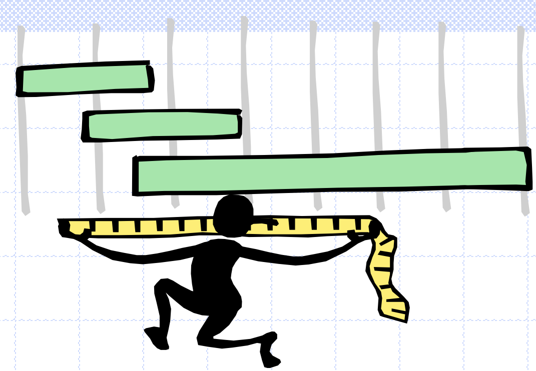
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		







Performance of Hashing



- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is

$$1 / (1 - \alpha)$$

- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Universal Hashing (§ 2.5.6)

- A family of hash functions is **universal** if, for any $0 \leq i,j \leq M-1$,
  $Pr(h(j)=h(k)) \leq 1/N$.
- Choose p as a prime between M and 2M.
- Randomly select $0<a<p$ and $0 \leq b<p$, and define $h(k)=(ak+b \bmod p) \bmod N$

- Theorem: The set of all functions, h, as defined here, is universal.

Proof of Universality (Part 1)

- Let f(k) = ak+b mod p
- Let g(k) = k mod N
- So h(k) = g(f(k)).
- f causes no collisions:
  - Let f(k) = f(j).
  - Suppose k<j. Then
- So a(j-k) is a multiple of p
- But both are less than p
- So a(j-k) = 0. I.e., j=k. (contradiction)
- Thus, f causes no collisions.

$$aj+b-\left\lfloor \frac{aj+b}{p} \right\rfloor p = ak+b-\left\lfloor \frac{ak+b}{p} \right\rfloor p$$

$$a(j-k) = \left(\left\lfloor \frac{aj+b}{p} \right\rfloor - \left\lfloor \frac{ak+b}{p} \right\rfloor \right) p$$

Proof of Universality (Part 2)

- If f causes no collisions, only g can make h cause collisions.
- Fix a number x. Of the p integers y=f(k), different from x, the number such that g(y)=g(x) is at most $\lceil p/N \rceil - 1$
- Since there are p choices for x, the number of h's that will cause a collision between j and k is at most

$$p(\lceil p/N \rceil - 1) \leq \frac{p(p-1)}{N}$$

- There are p(p-1) functions h. So probability of collision is at most

$$\frac{p(p-1)/N}{p(p-1)} = \frac{1}{N}$$

- Therefore, the set of possible h functions is universal.