SE274 Data Structure

Lecture 5: Heaps, Priority Queues

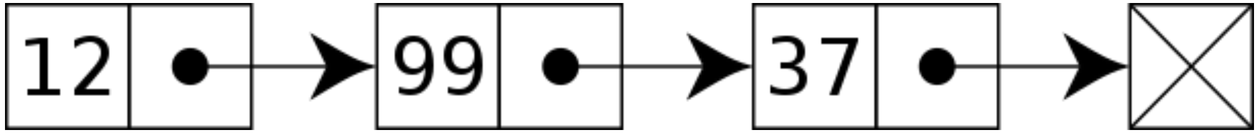
Mar 30, 2020

Instructor: Sunjun Kim

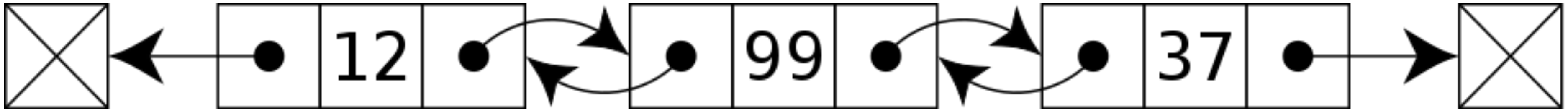
Information&Communication Engineering, DGIST

Recap: Linked List

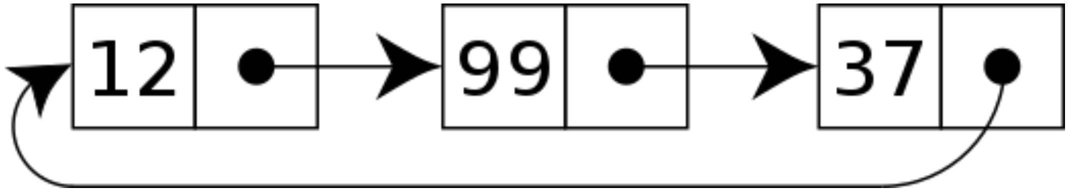
- Singly linked list



- Doubly linked list



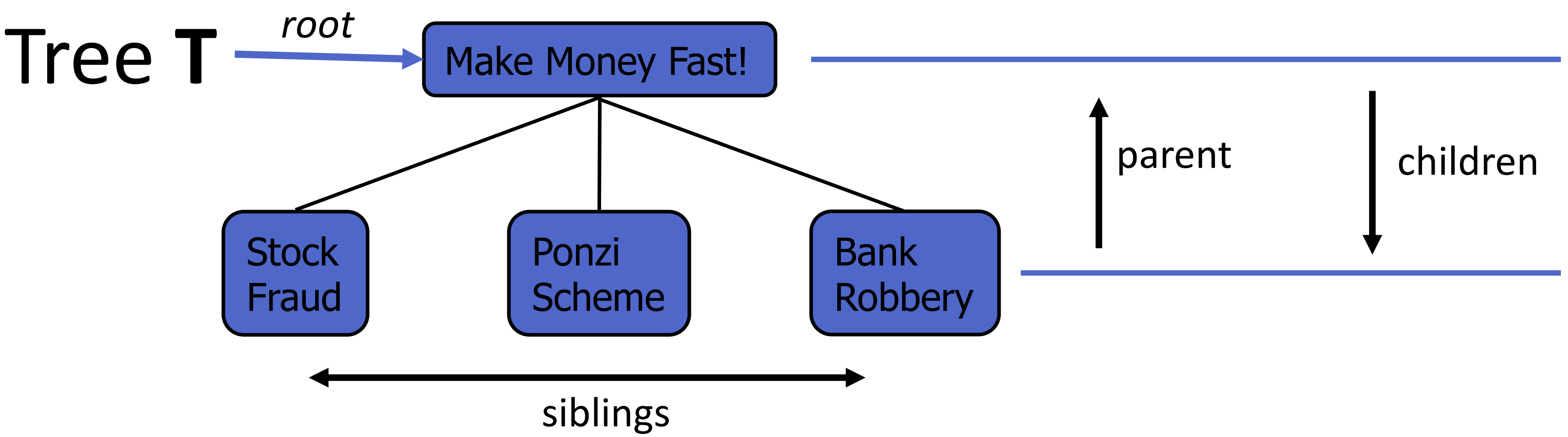
- Circular linked list



* Image credit: https://en.wikipedia.org/wiki/Linked_list

Recap: Tree

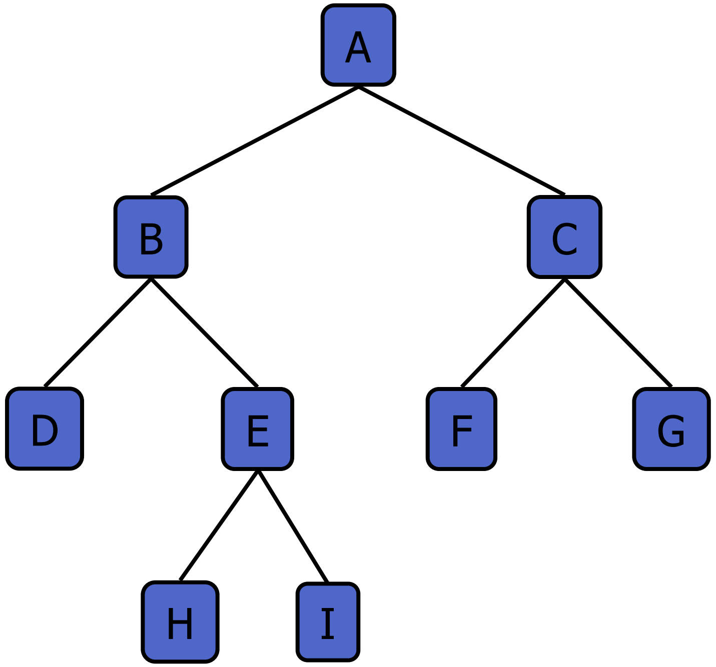
- We define a *tree* **T** as a set of ***nodes*** storing elements such that ***Nodes*** have **a *parent-child*** relationship, that satisfies:
  - If **T** is nonempty, it has a special node, called the ***root*** of **T**.
  - Each node ***v*** of **T** different from the root has a unique *parent node* ***w***; every node with parent ***w*** is a child of ***w,*** nodes that share the same parent are called *siblings*.



Recap: Binary Trees

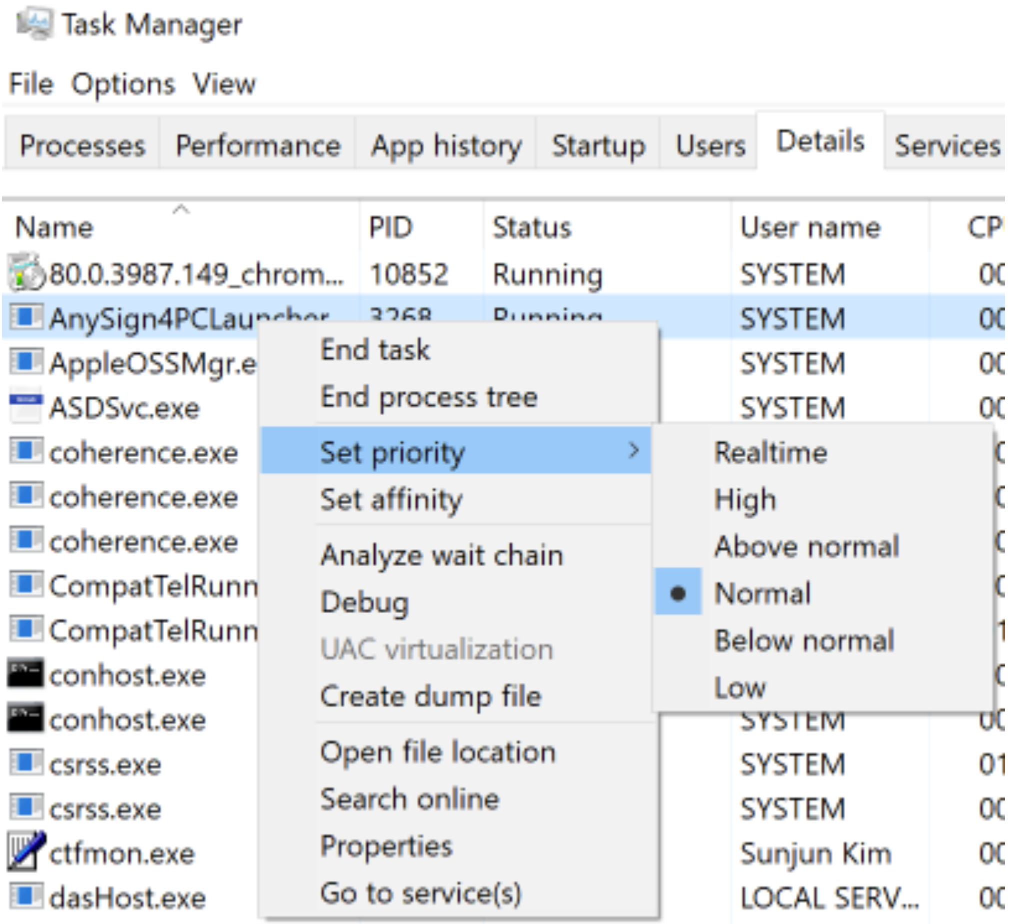
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - 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

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



Priority Queues

Priority Queue: practical examples



Priority Queue: idea

- Assigning “priority” to Queue items.
- Each element is associated with a ***key***, which represents the **priority** of the element.
    - Convention: lower key value = higher its priority.
        - 3 < 5
        - abc < fes
        - 🐭<🐯<🐷
        - 나<너
    - Keys are not necessarily unique.

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined as a key with a ***total order*** relation
- Two distinct entries in a priority queue can have the same key
- ***This prevents confliction between two items.***

- Mathematical concept of ***total order relation*** $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

- ***Example of total ordered set)***
  - The letters of Alphabet, in dictionary order.
  - Set of integers, real numbers, etc.
  - Ordered fields

Priority Queue ADT

- A priority queue stores a collection of items
- Each item is a pair
  (key, value)
- Main methods of the Priority Queue ADT
  - **add** (k, x)
    inserts an item with key k and value x
  - **remove_min**()
    removes and returns the item with smallest key
- Additional methods
  - **min**()
    returns, but does not remove, an item with smallest key
  - **len**(P), **is_empty**()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Priority Queue Example

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.min()	(3,B)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_min()	(3,B)	{(5,A), (7,D), (9,C)}
P.remove_min()	(5,A)	{(7,D), (9,C)}
len(P)	2	{(7,D), (9,C)}
P.remove_min()	(7,D)	{(9,C)}
P.remove_min()	(9,C)	{ }
P.is_empty()	True	{ }
P.remove_min()	"error"	{ }

Composition Design Pattern

- An item in a priority queue is simply a key-value pair
- Priority queues store items to allow for efficient insertion and removal based on keys

```python
class PriorityQueueBase:
  """Abstract base class for a priority queue."""

  class _Item:
    """Lightweight composite to store priority queue items."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __lt__(self, other):
      return self._key < other._key          # compare items based on their keys

  def is_empty(self):                         # concrete method assuming abstract len
    """Return True if the priority queue is empty."""
    return len(self) == 0
```

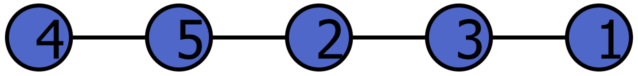
__lt__() = larger than

→ Is called when …

A = _Item(aa, ab)
B = _Item(ba, bb)
if A < B:
.....

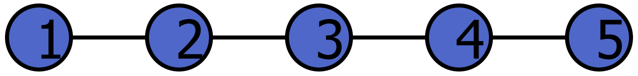
Sequence-based Priority Queue

- Implementation with an unsorted list



- Performance:
  - add takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - Remove_min and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list



- Performance:
  - add takes $O(n)$ time since we have to find the place where to insert the item
  - remove_min and min take $O(1)$ time, since the smallest key is at the beginning

Note: Positional List

- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position acts as a marker or token within the broader positional list.
- A position p is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
  - p.element( ): Return the element stored at position p.

Note: Positional Accessor Operations

L.first(): Return the position of the first element of L, or None if L is empty.

L.last(): Return the position of the last element of L, or None if L is empty.

L.before(p): Return the position of L immediately before position p, or None if p is the first position.

L.after(p): Return the position of L immediately after position p, or None if p is the last position.

L.is_empty(): Return True if list L does not contain any elements.

len(L): Return the number of elements in the list.

iter(L): Return a forward iterator for the *elements* of the list. See Section 1.8 for discussion of iterators in Python.

Note: Positional Update Operations

- L.add_first(e): Insert a new element e at the front of L, returning the position of the new element.
- L.add_last(e): Insert a new element e at the back of L, returning the position of the new element.
- L.add_before(p, e): Insert a new element e just before position p in L, returning the position of the new element.
- L.add_after(p, e): Insert a new element e just after position p in L, returning the position of the new element.
- L.replace(p, e): Replace the element at position p with element e, returning the element formerly at position p.
- L.delete(p): Remove and return the element at position p in L, invalidating the position.

Unsorted List Implementation

```python
class UnsortedPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with an unsorted list."""

  def _find_min(self):                    # nonpublic utility
    """Return Position of item with minimum key."""
    if self.is_empty():                   # is_empty inherited from base class
      raise Empty('Priority queue is empty')
    small = self._data.first()
    walk = self._data.after(small)
    while walk is not None:
      if walk.element() < small.element():
        small = walk
      walk = self._data.after(walk)
    return small

  def __init__(self):
    """Create a new empty Priority Queue."""
    self._data = PositionalList()

  def __len__(self):
    """Return the number of items in the priority queue."""
    return len(self._data)

  def add(self, key, value):
    """Add a key-value pair."""
    self._data.add_last(self._Item(key, value))

  def min(self):
    """Return but do not remove (k,v) tuple with minimum key."""
    p = self._find_min()
    item = p.element()
    return (item._key, item._value)

  def remove_min(self):
    """Remove and return (k,v) tuple with minimum key."""
    p = self._find_min()
    item = self._data.delete(p)
    return (item._key, item._value)
```

Sorted List Implementation

```
class SortedPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with a sorted list."""

  def __init__(self):
    """Create a new empty Priority Queue."""
    self._data = PositionalList()

  def __len__(self):
    """Return the number of items in the priority queue."""
    return len(self._data)

  def add(self, key, value):
    """Add a key-value pair."""
    newest = self._Item(key, value)              # make new item instance
    walk = self._data.last()       # walk backward looking for smaller key
    while walk is not None and newest < walk.element():
      walk = self._data.before(walk)
    if walk is None:
      self._data.add_first(newest)               # new key is smallest
    else:
      self._data.add_after(walk, newest)         # newest goes after walk

  def min(self):
    """Return but do not remove (k,v) tuple with minimum key."""
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    p = self._data.first()
    item = p.element()
    return (item._key, item._value)

  def remove_min(self):
    """Remove and return (k,v) tuple with minimum key."""
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    item = self._data.delete(self._data.first())
    return (item._key, item._value)
```

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of add operations
  2. Remove the elements in sorted order with a series of remove_min 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***

$P \leftarrow$ priority queue with comparator C

while $\neg$*S.is_empty* ()

  $e \leftarrow$ *S.remove_first* ()

  P.add (*e*, $\varnothing$)

while $\neg$*P.is_empty*()

  $e \leftarrow$ *P.removeMin*().*key*()

  S.add_last(*e*)

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 insert 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

Selection-Sort Example

	Sequence S		Priority Queue P
Input:	(7,4,8,2,5,3,9)		()
Phase 1			
(a)	(4,8,2,5,3,9)		(7)
(b)	(8,2,5,3,9)	(7,4)	
..	..	..	
(g)	()		(7,4,8,2,5,3,9)
Phase 2			
(a)	(2)		(7,4,8,5,3,9)
(b)	(2,3)		(7,4,8,5,9)
(c)	(2,3,4)		(7,8,5,9)
(d)	(2,3,4,5)		(7,8,9)
(e)	(2,3,4,5,7)	(8,9)	
(f)	(2,3,4,5,7,8)		(9)
(g)	(2,3,4,5,7,8,9)		()

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 $\boldsymbol{n}$ insert operations takes time proportional to

$$1 + 2 + \ldots + \boldsymbol{n}$$

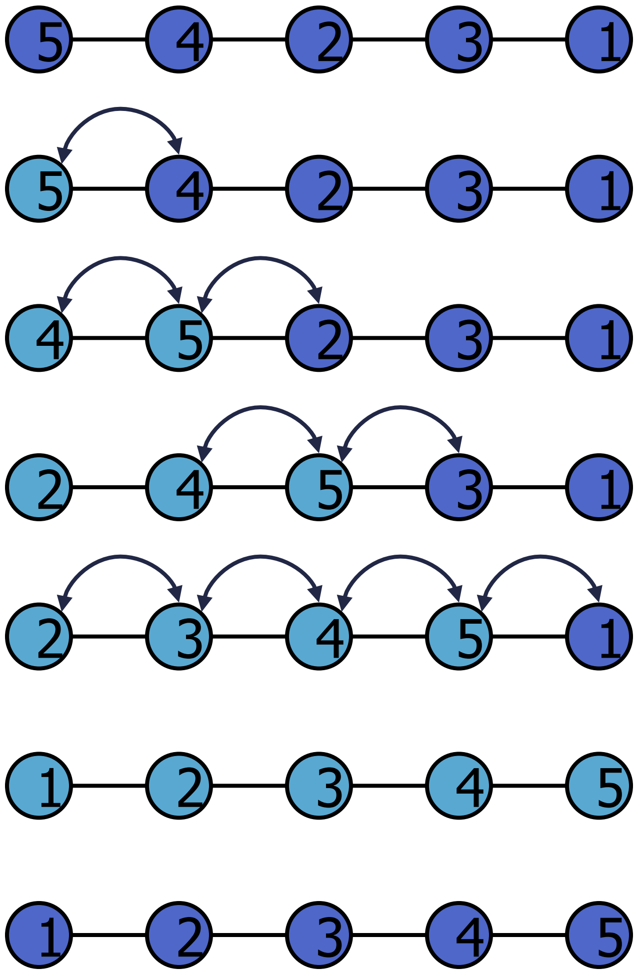
  2. Removing the elements in sorted order from the priority queue with a series of $\boldsymbol{n}$ removeMin operations takes $\boldsymbol{O}(\boldsymbol{n})$ time
- Insertion-sort runs in $\boldsymbol{O}(\boldsymbol{n}^2)$ time

Insertion-Sort Example

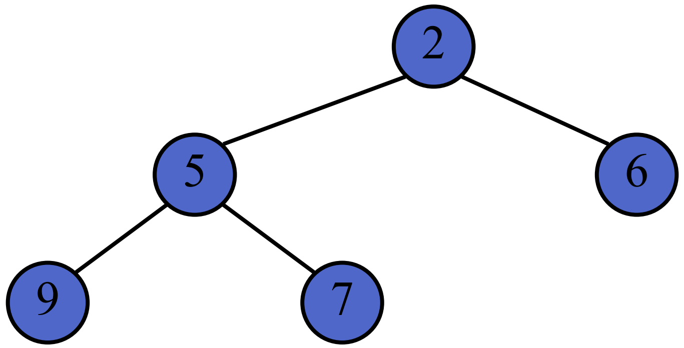
	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

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 swaps instead of modifying the sequence



Heaps



Recall Priority Queue ADT

- A priority queue stores a collection of items
- Each item is a pair
  (key, value)
- Main methods of the Priority Queue ADT
  - add(k, x)
    inserts an item with key k and value x
  - remove_min()
    removes and returns the item with smallest key
- Additional methods
  - min()
    returns, but does not remove, an item with smallest key
  - len(), is_empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Recall PQ Sorting

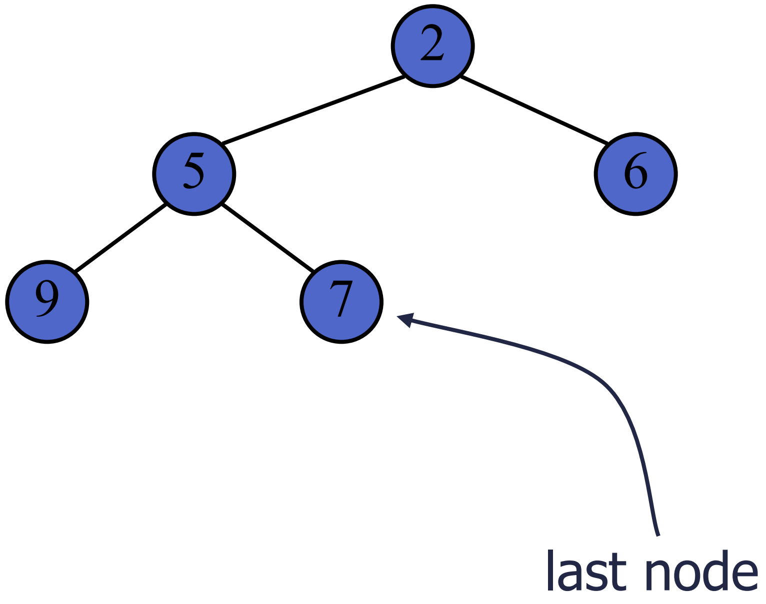
- We use a priority queue
  - Insert the elements with a series of add operations
  - Remove the elements in sorted order with a series of remove_min operations
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives selection-sort: O(n^2) time
  - Sorted sequence gives insertion-sort: O(n^2) time
- Can we do better?

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.is_empty*** ()
    e ← ***S.remove*** (***S. first*** ())
    P.add(***e***, ***e***)
  while ¬***P.is_empty***()
    e ← ***P.remove_min***().***key***()
    S.add_last(***e***)

Heaps

- A heap is a binary tree storing keys at its 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 *(fully filled)*
  - at depth h, the remaining nodes are residing in the leftmost possible positions at that level.

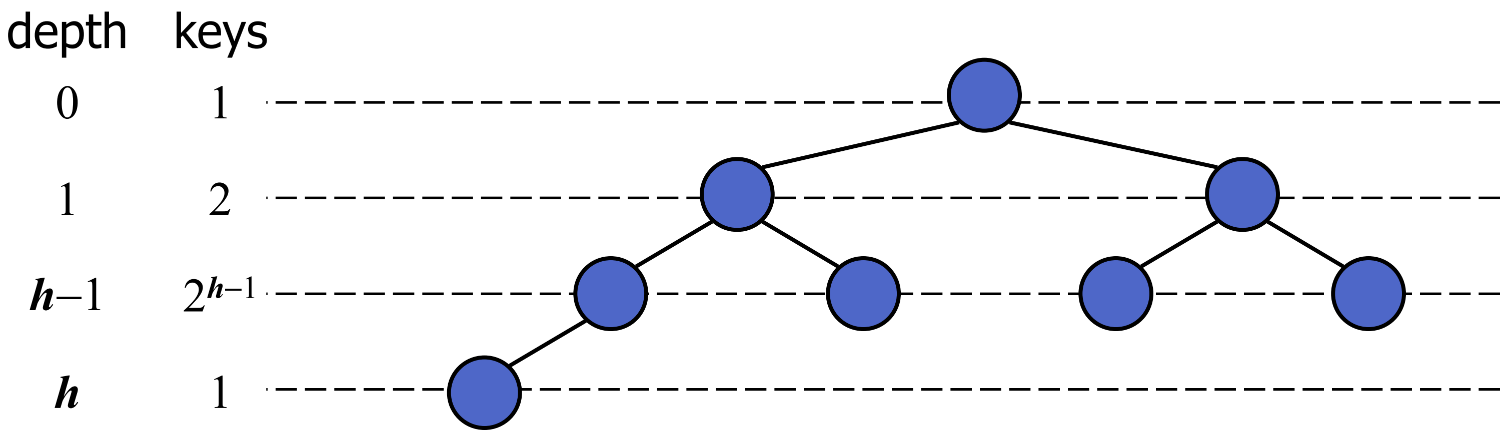
- The last node of a heap is the rightmost node of maximum depth



Height of a Heap

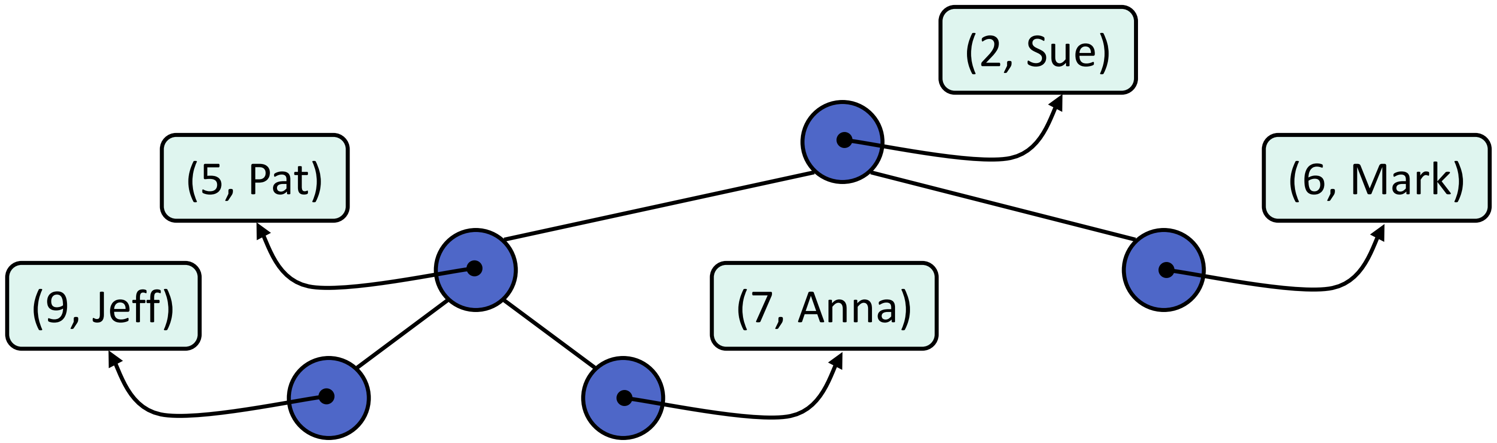
- Theorem: A heap storing n keys has height $O(\log n)$

  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 $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$



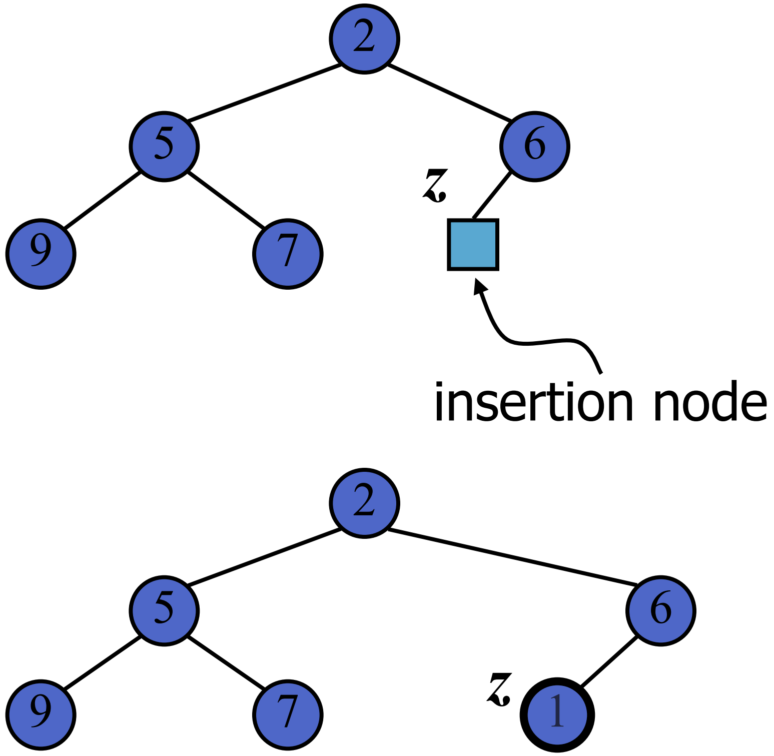
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each node
- We keep track of the **position of the last node**



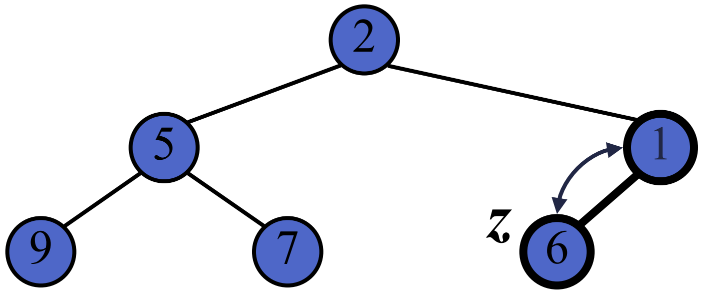
Insertion into a Heap

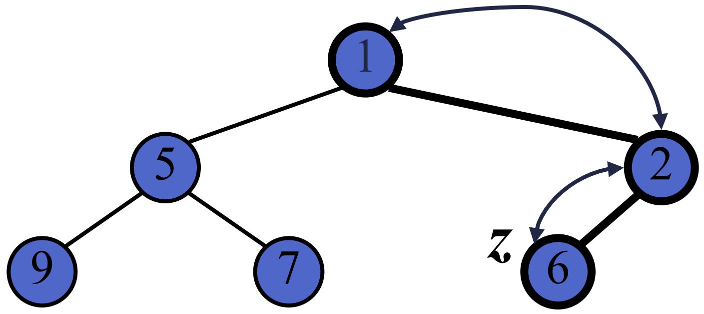
- Method add of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)



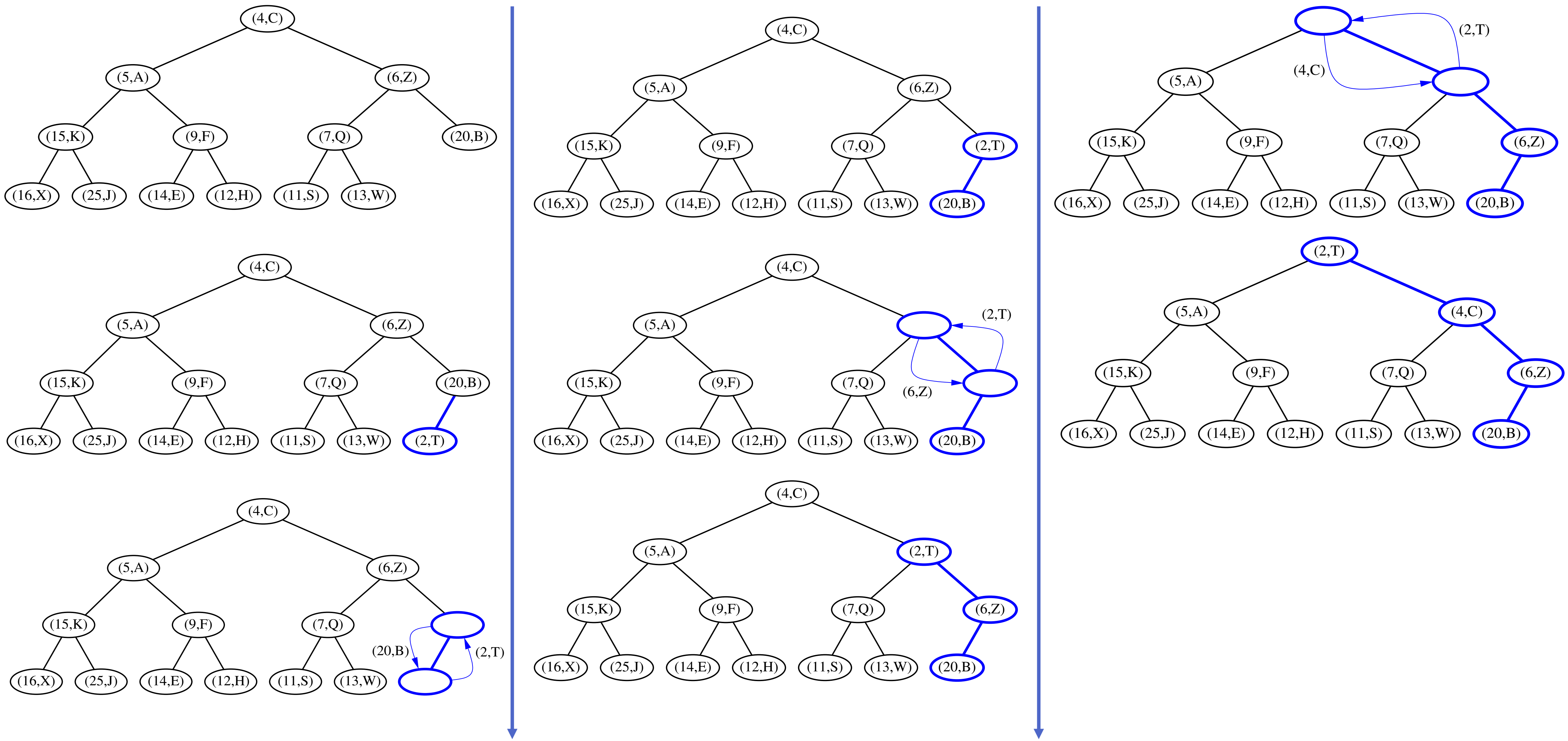
Upheap

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



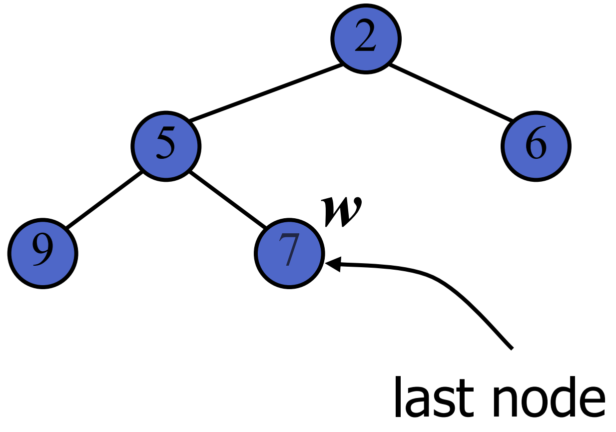


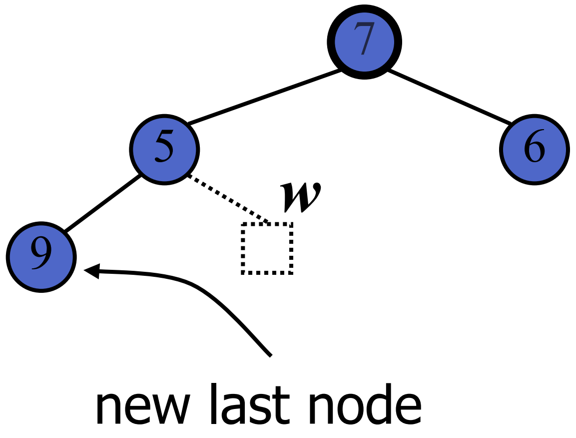
Upheap example



Removal from a Heap

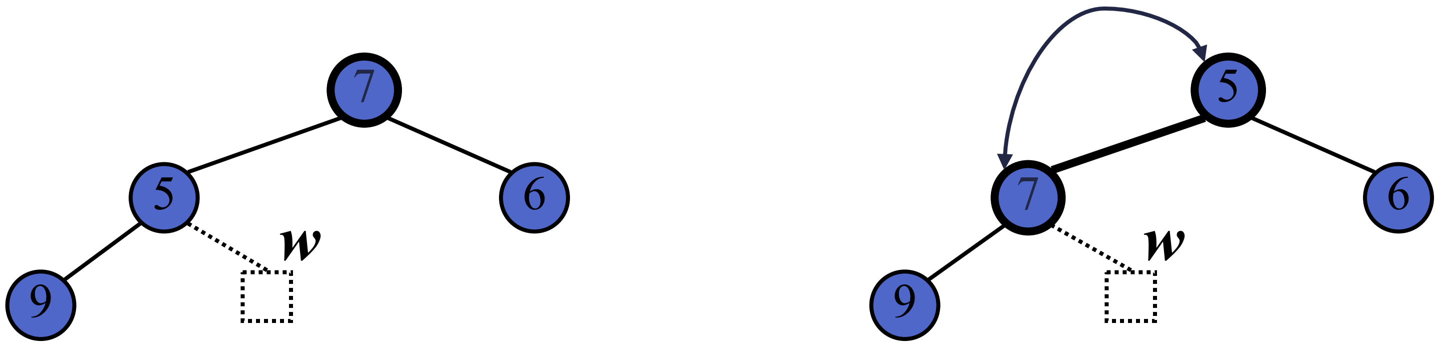
- Method remove_min of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)



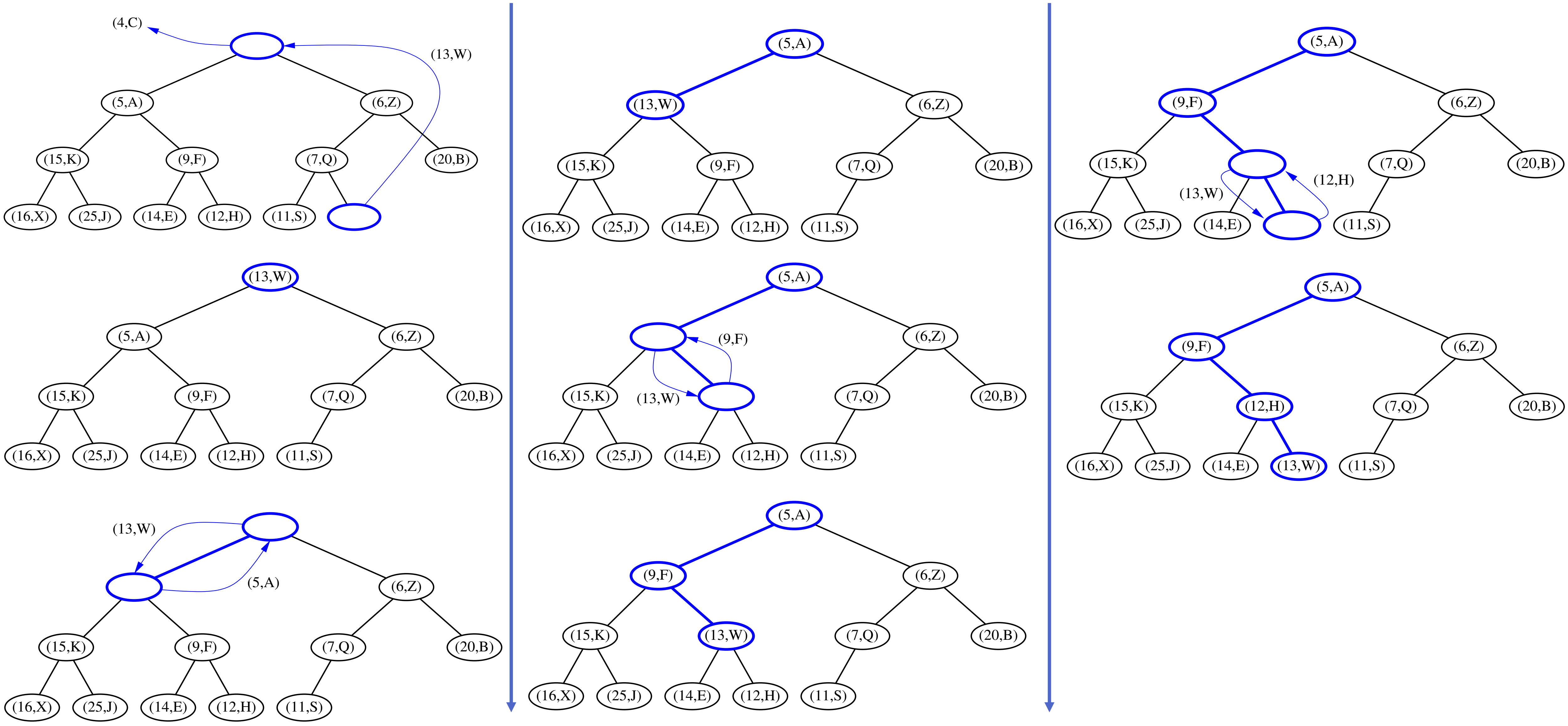


Downheap

- After replacing the root key with the key $\boldsymbol{k}$ of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key $\boldsymbol{k}$ along a downward path from the root
  - For swapping, select a child node with the minimal key (=higher priority)
- Downheap terminates when key $\boldsymbol{k}$ reaches a leaf or a node whose children have keys greater than or equal to $\boldsymbol{k}$
- Since a heap has height $\boldsymbol{O}(\log \boldsymbol{n})$, downheap runs in $\boldsymbol{O}(\log \boldsymbol{n})$ time

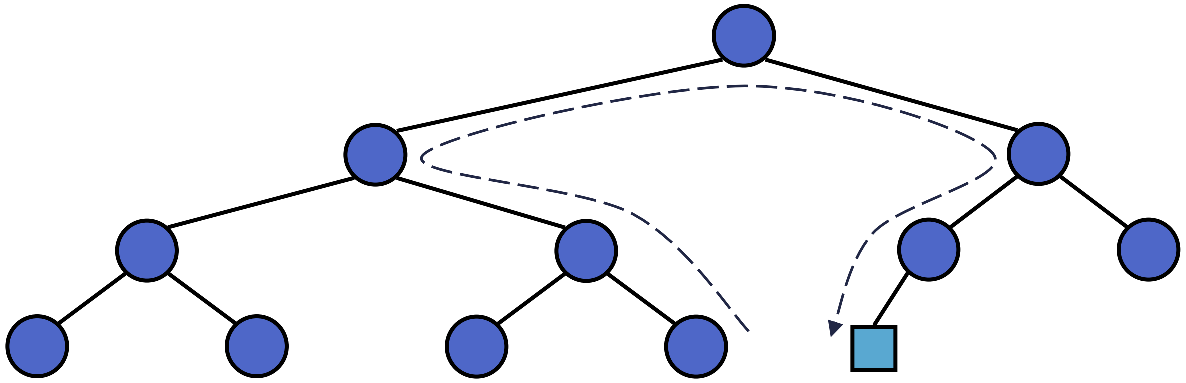


Downheap example

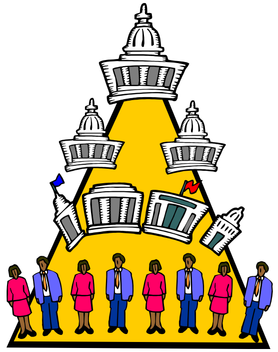


Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
    - If a left child is reached, go to the right child
    - If a root is reached, stay there.
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal



Heap-Sort

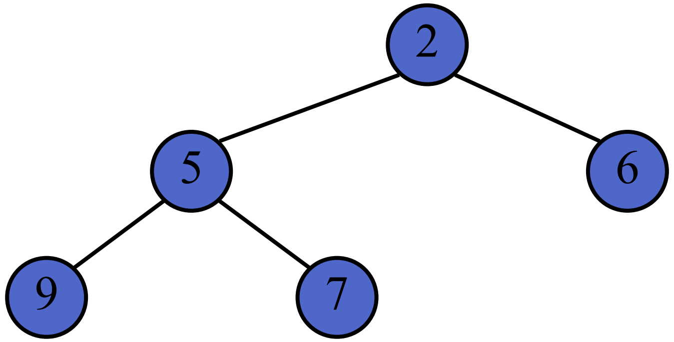


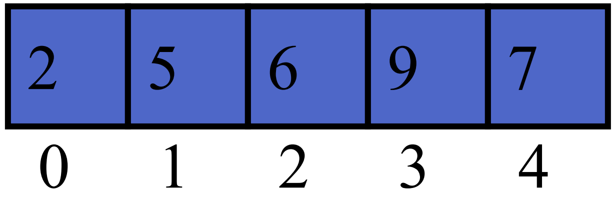
- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods add and remove_min take $O(\log n)$ time
  - methods len, is_empty, and min take time $O(1)$ time

- 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

Array-based Heap Implementation

- We can represent a heap with $\boldsymbol{n}$ keys by means of an array of length $\boldsymbol{n}$
- For the node at rank $\boldsymbol{i}$
  - the left child is at rank $2\boldsymbol{i} + 1$
  - the right child is at rank $2\boldsymbol{i} + 2$
- Links between nodes are not explicitly stored
- Operation add corresponds to inserting at rank $\boldsymbol{n} + 1$
- Operation remove_min corresponds to removing at rank $\boldsymbol{n}$
- Yields in-place heap-sort



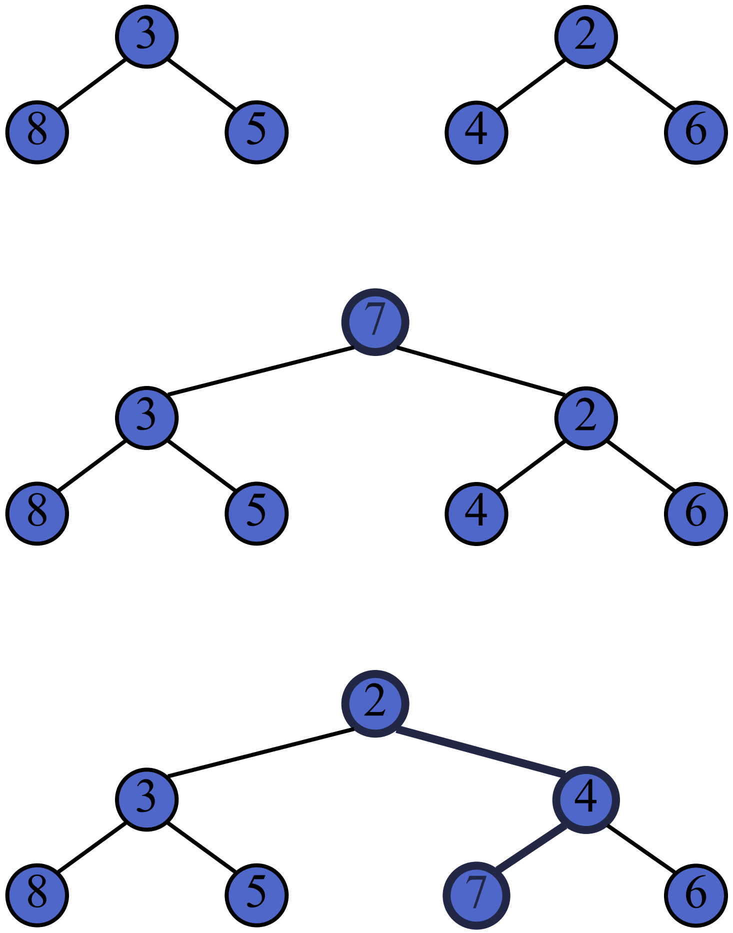


Python Heap Implementation

```python
class HeapPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with a binary heap."""
  #------------------------------ nonpublic behaviors ------------------------------
  def _parent(self, j):
    return (j-1) // 2

  def _left(self, j):
    return 2*j + 1

  def _right(self, j):
    return 2*j + 2

  def _has_left(self, j):
    return self._left(j) < len(self._data)      # index beyond end of list?

  def _has_right(self, j):
    return self._right(j) < len(self._data)     # index beyond end of list?

  def _swap(self, i, j):
    """Swap the elements at indices i and j of array."""
    self._data[i], self._data[j] = self._data[j], self._data[i]

  def _upheap(self, j):
    parent = self._parent(j)
    if j > 0 and self._data[j] < self._data[parent]:
      self._swap(j, parent)
      self._upheap(parent)                      # recur at position of parent

  def _downheap(self, j):
    if self._has_left(j):
      left = self._left(j)
      small_child = left                        # although right may be smaller
      if self._has_right(j):
        right = self._right(j)
        if self._data[right] < self._data[left]:
          small_child = right
      if self._data[small_child] < self._data[j]:
        self._swap(j, small_child)
        self._downheap(small_child)             # recur at position of small child

  #------------------------------ public behaviors ------------------------------
  def __init__(self):
    """Create a new empty Priority Queue."""
    self._data = [ ]

  def __len__(self):
    """Return the number of items in the priority queue."""
    return len(self._data)

  def add(self, key, value):
    """Add a key-value pair to the priority queue."""
    self._data.append(self._Item(key, value))
    self._upheap(len(self._data) - 1)           # upheap newly added position

  def min(self):
    """Return but do not remove (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    item = self._data[0]
    return (item._key, item._value)

  def remove_min(self):
    """Remove and return (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    self._swap(0, len(self._data) - 1)          # put minimum item at the end
    item = self._data.pop( )                    # and remove it from the list;
    self._downheap(0)                           # then fix new root
    return (item._key, item._value)
```

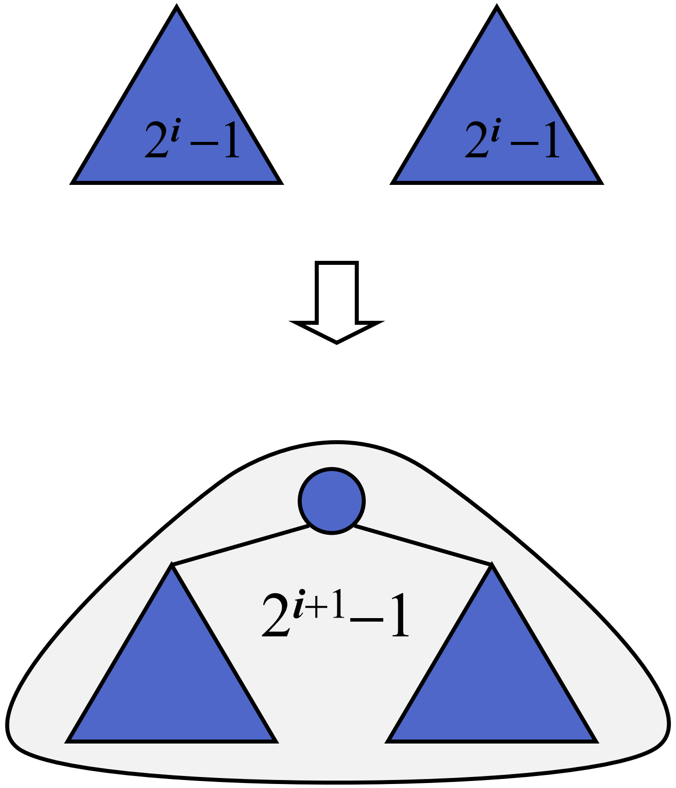
Merging Two Heaps

- We are given 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

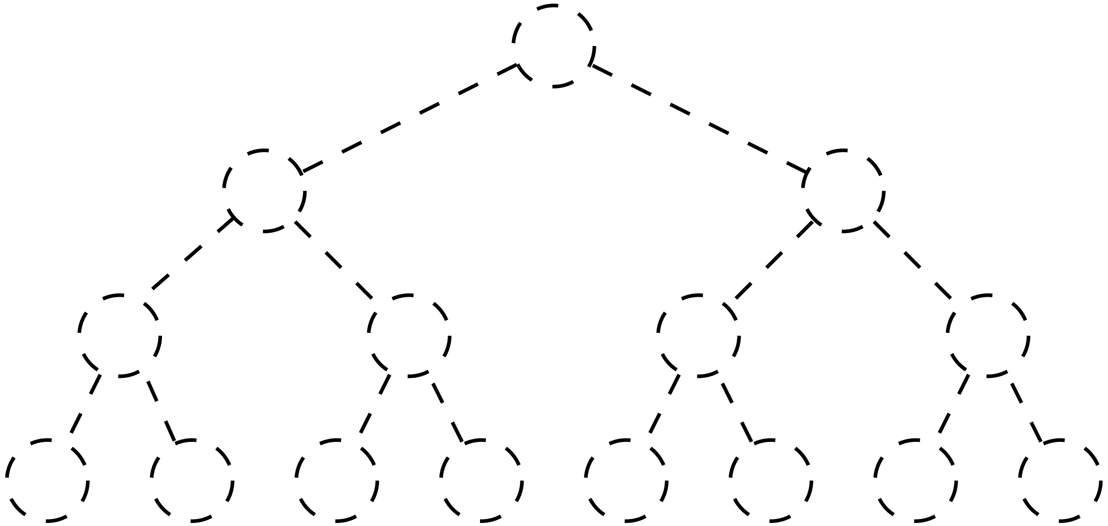


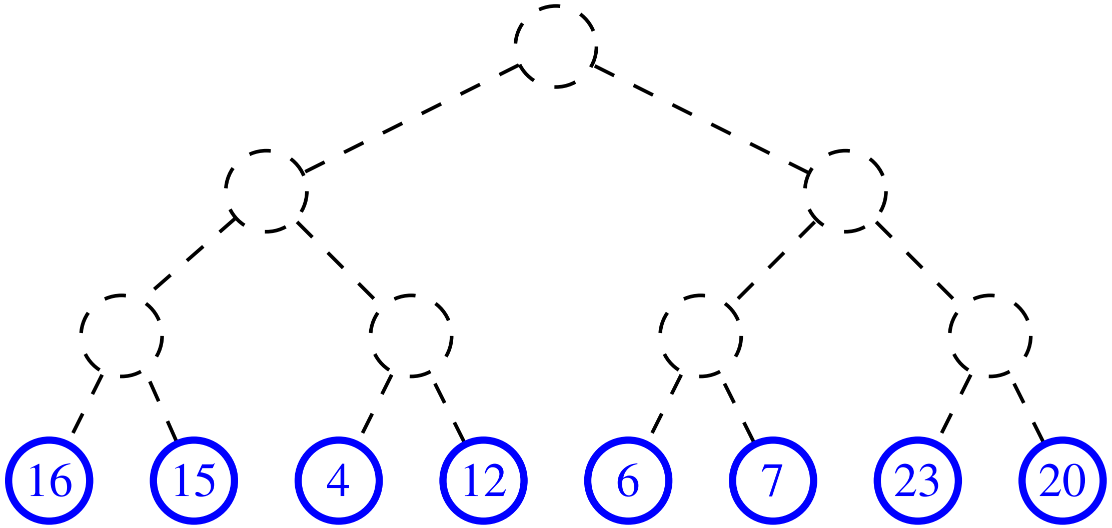
Bottom-up Heap Construction

- 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

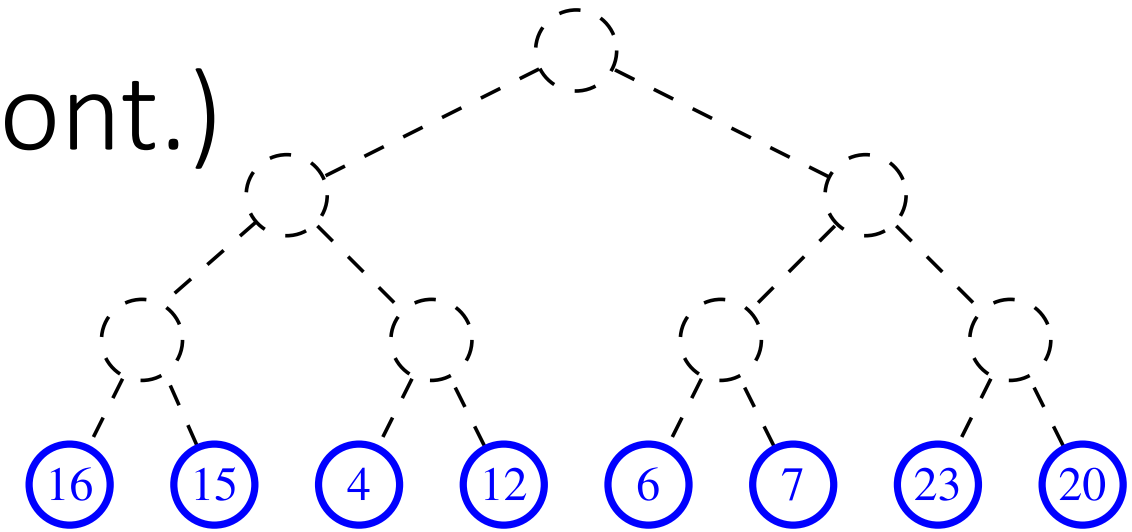


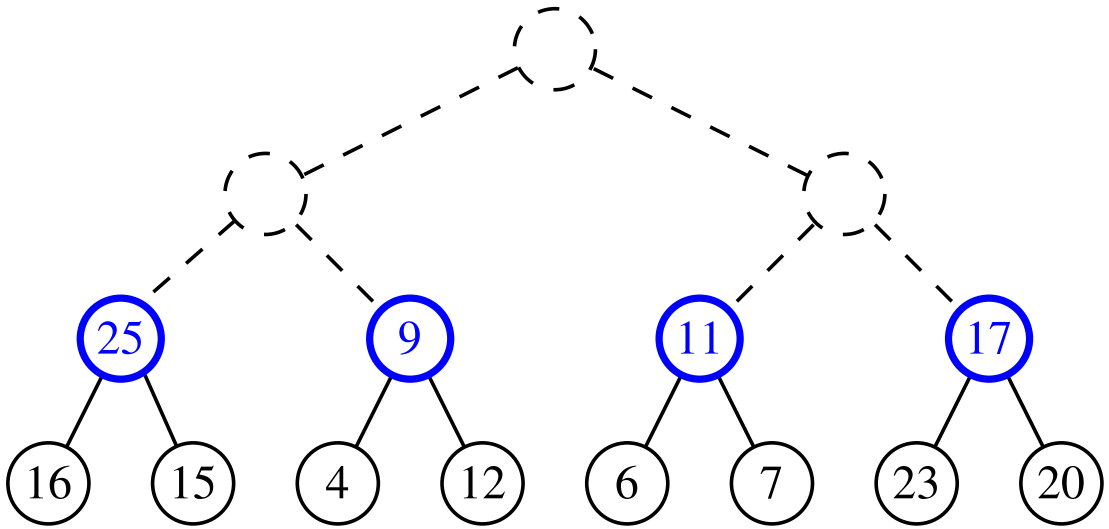
Example



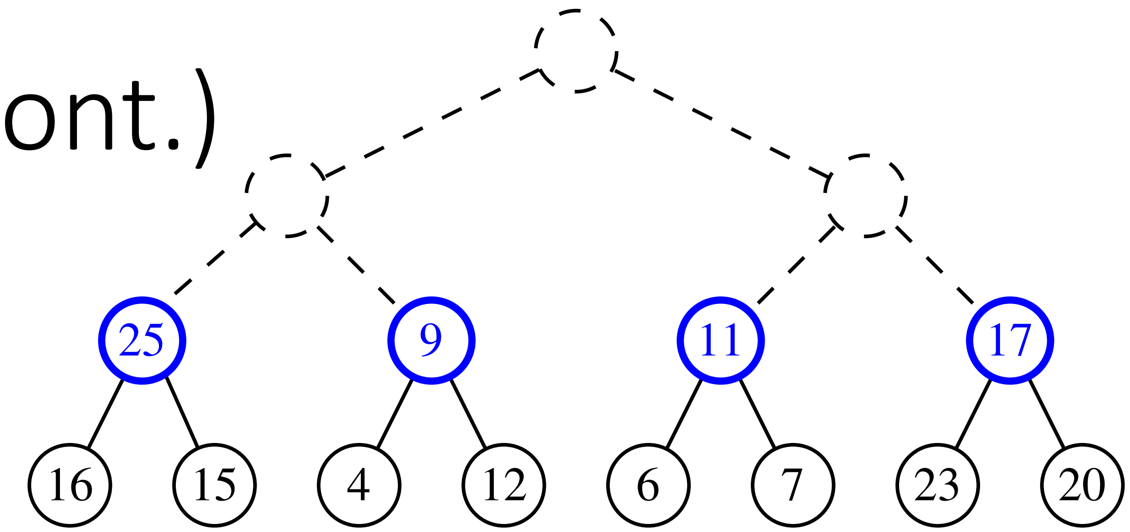


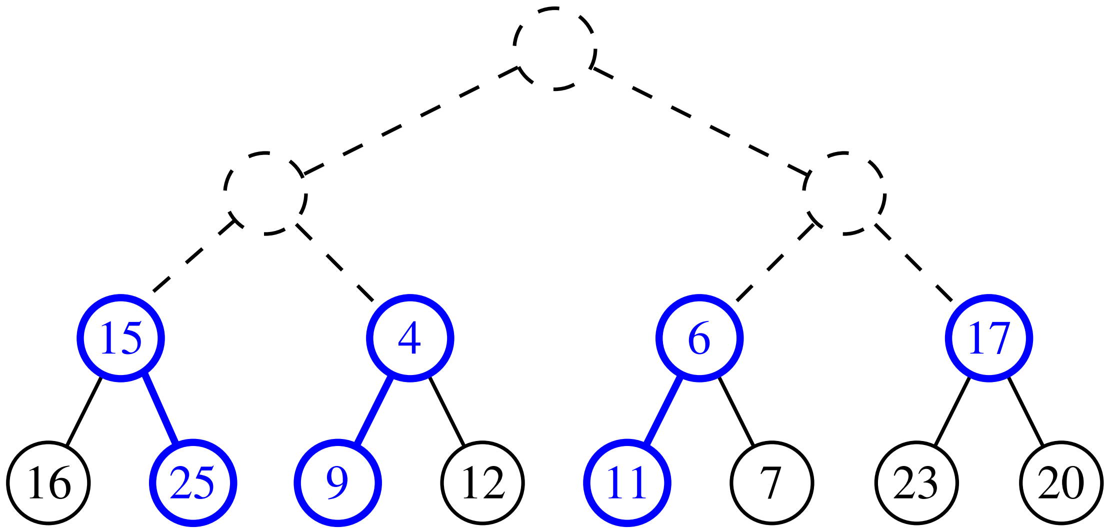
Example (cont.)



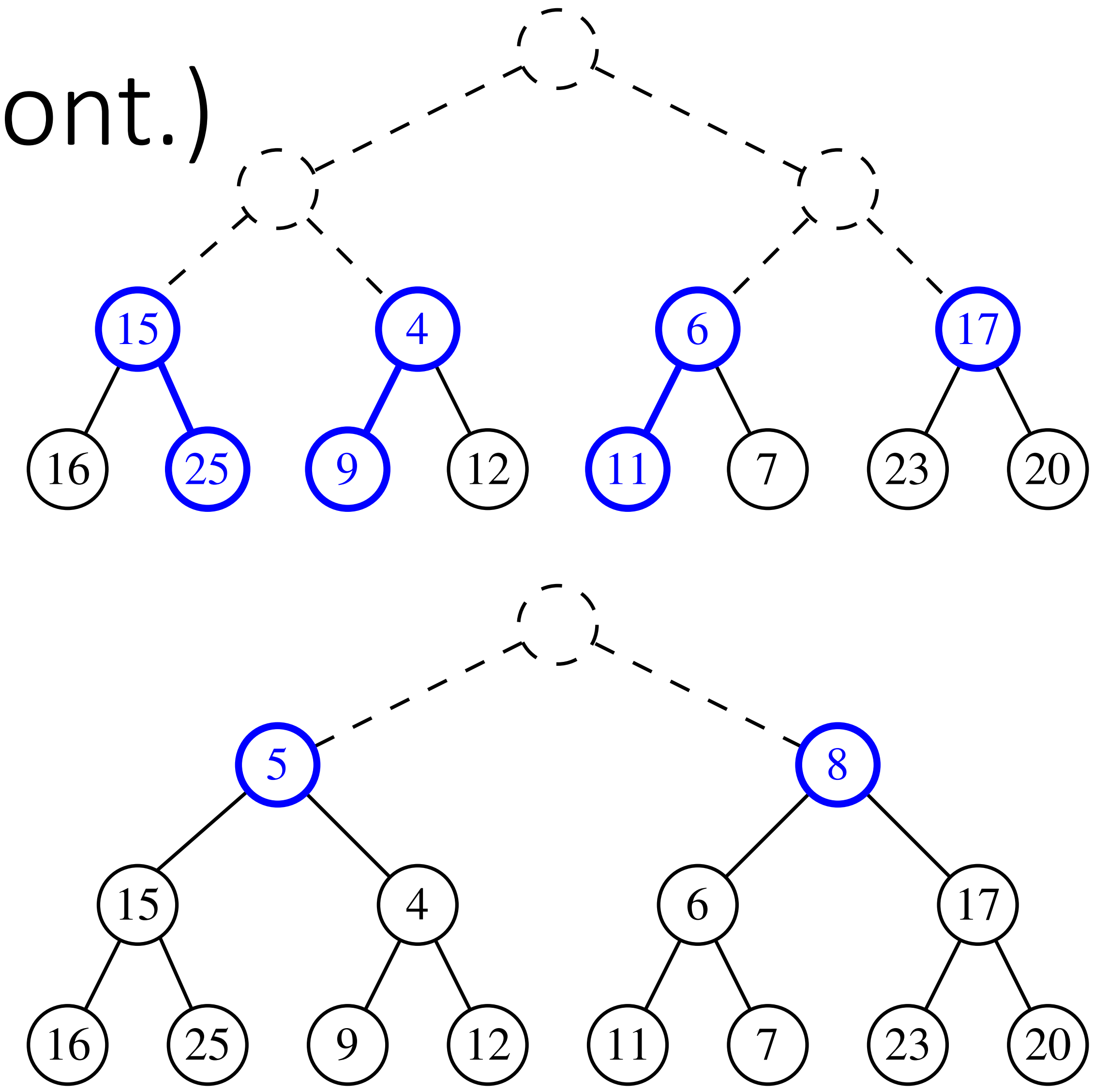


Example (cont.)

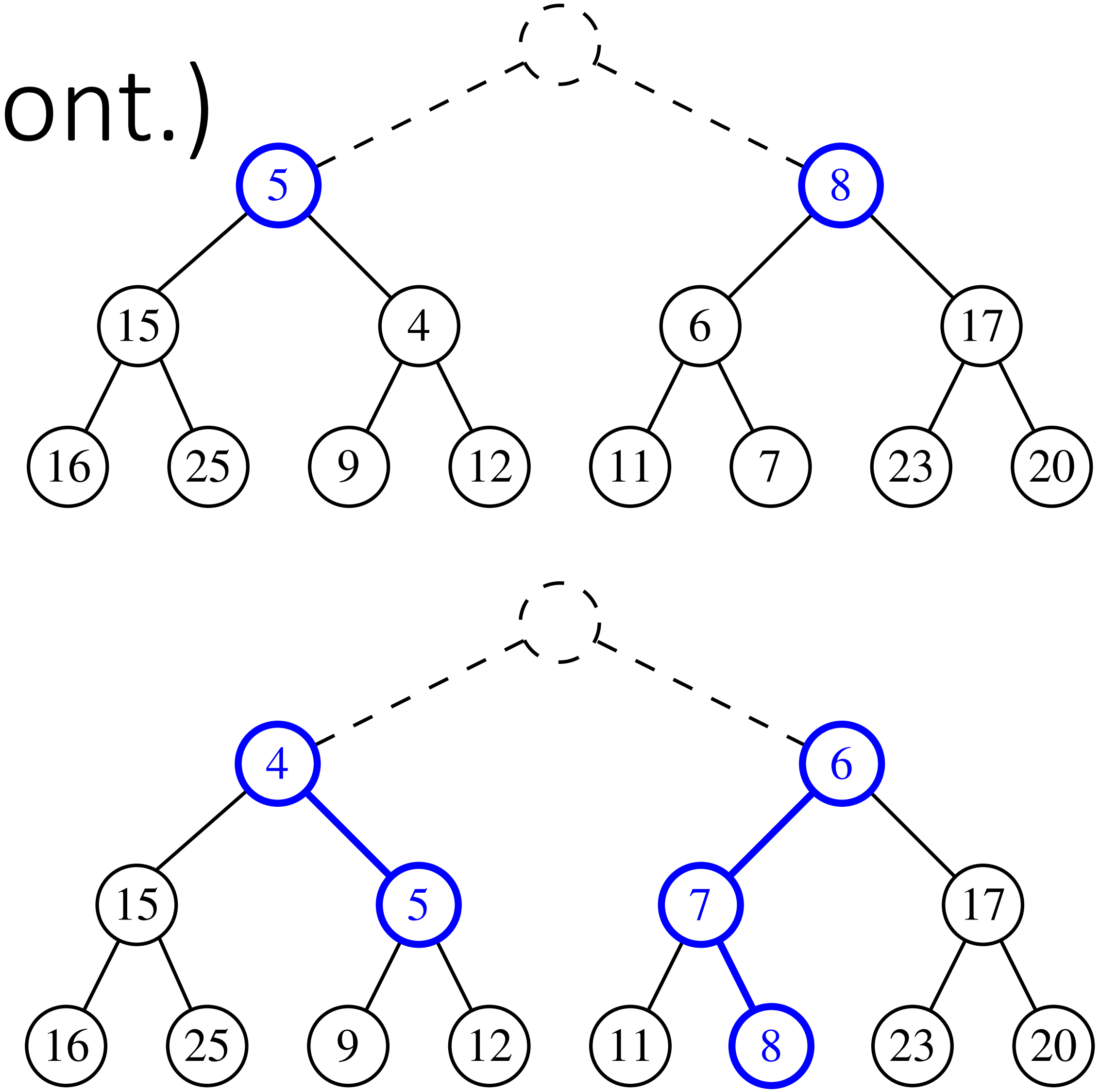




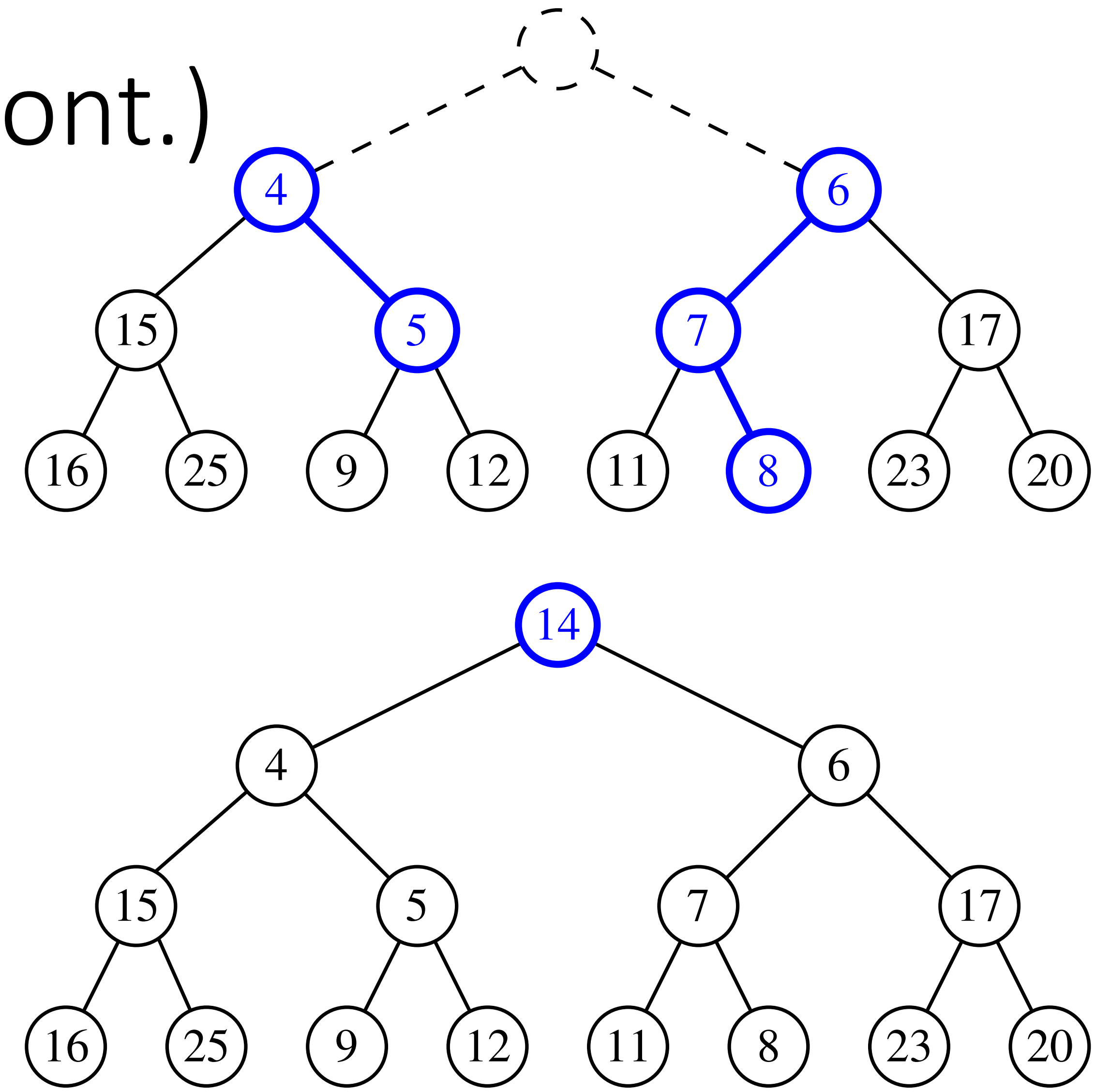
Example (cont.)



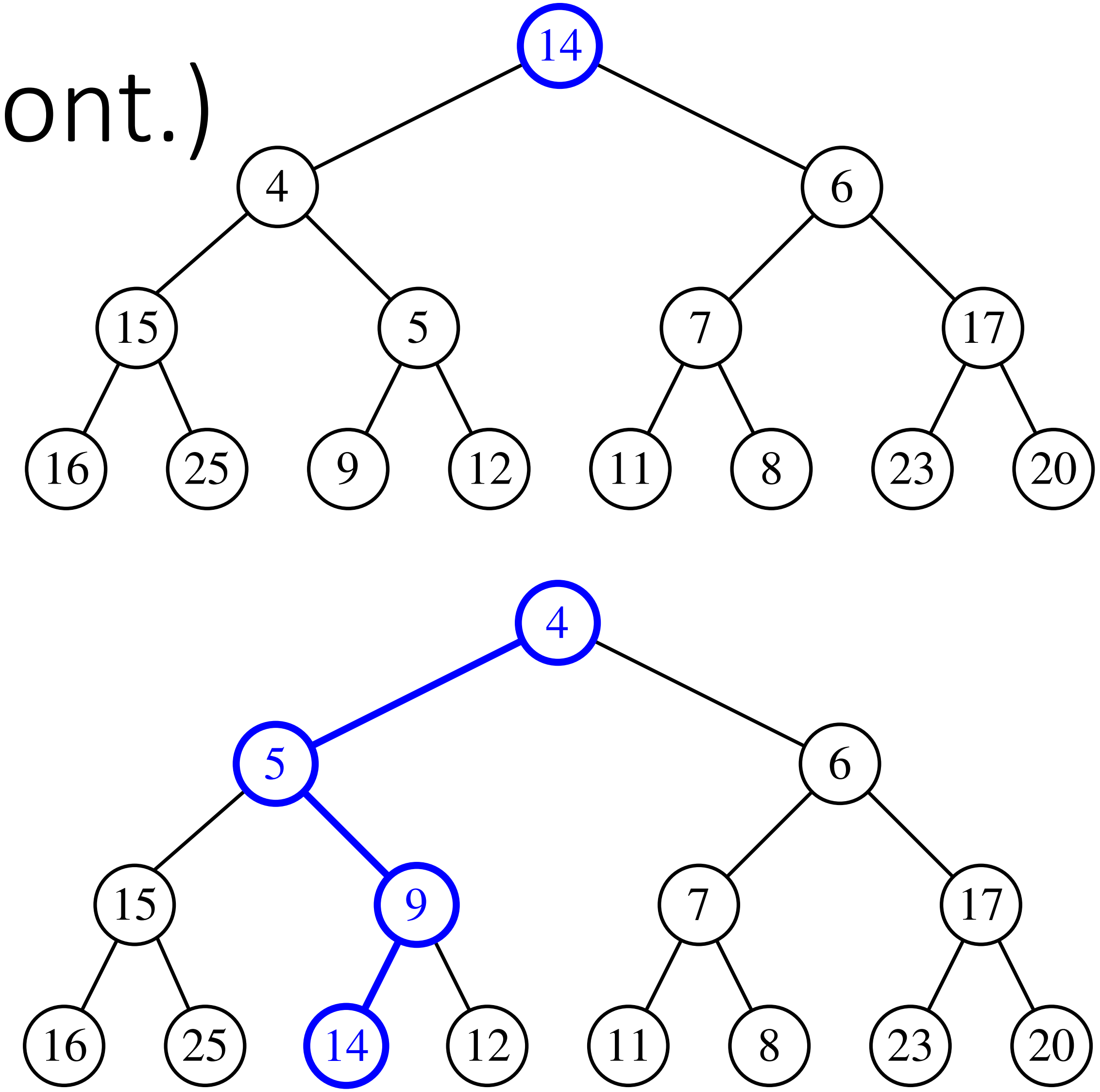
Example (cont.)



Example (cont.)

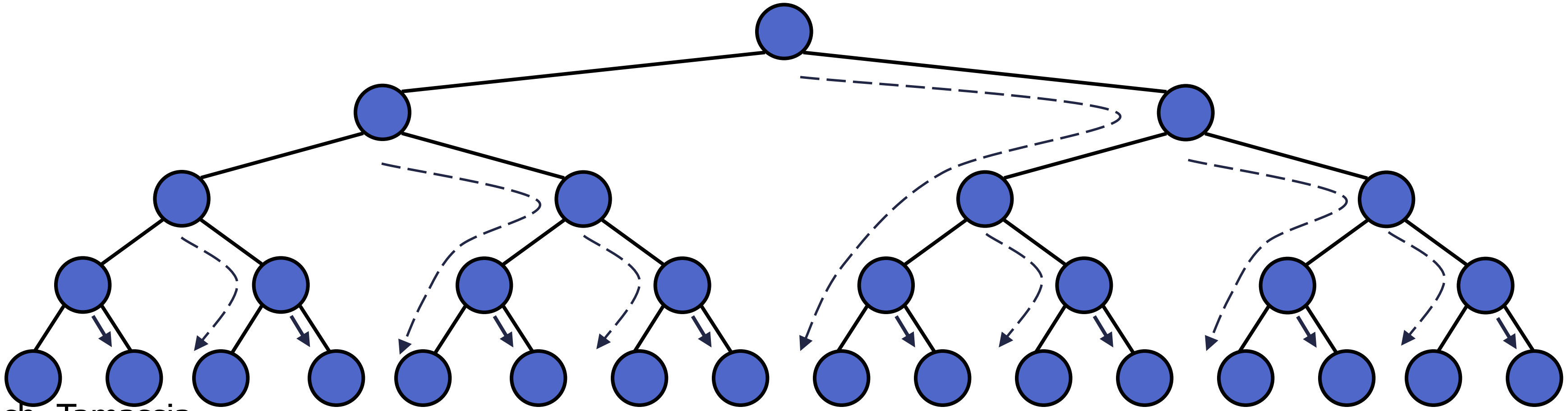


Example (cont.)



Analysis of Heap Construction

- 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



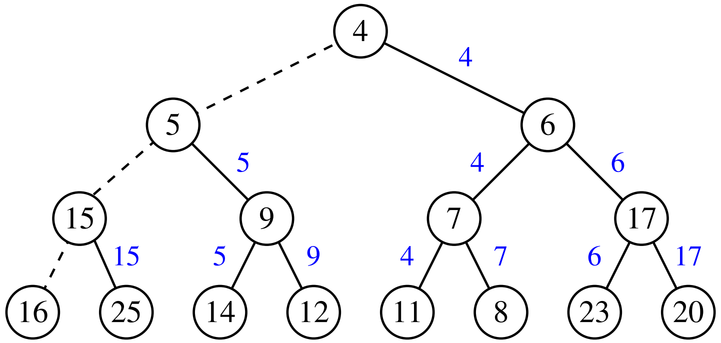
Exercise (optional) (textbook p383)

Asymptotic Analysis of Bottom-Up Heap Construction

Bottom-up heap construction is asymptotically faster than incrementally inserting n keys into an initially empty heap. Intuitively, we are performing a single down-heap operation at each position in the tree, rather than a single up-heap operation from each. Since more nodes are closer to the bottom of a tree than the top, the sum of the downward paths is linear, as shown in the following proposition.

Proposition 9.3: *Bottom-up construction of a heap with n entries takes $O(n)$ time, assuming two keys can be compared in $O(1)$ time.*

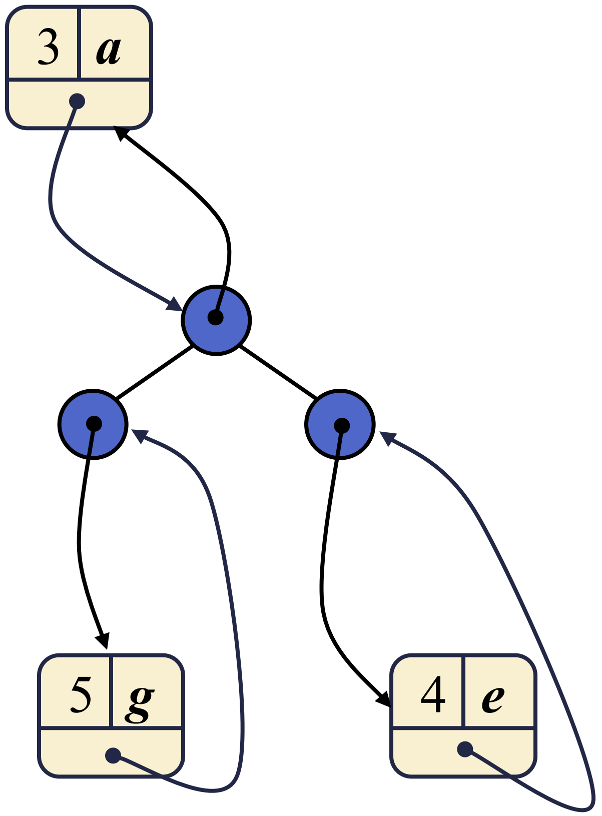
Exercise : answer



Justification: The primary cost of the construction is due to the down-heap steps performed at each nonleaf position. Let π_v denote the path of T from nonleaf node v to its "inorder successor" leaf, that is, the path that starts at v, goes to the right child of v, and then goes down leftward until it reaches a leaf. Although, π_v is not necessarily the path followed by the down-heap bubbling step from v, the length $\|\pi_v\|$ (its number of edges) is proportional to the height of the subtree rooted at v, and thus a bound on the complexity of the down-heap operation at v. We can bound the total running time of the bottom-up heap construction algorithm based on the sum of the sizes of paths, $\sum_v \|\pi_v\|$. For intuition, Figure 9.6 illustrates the justification "visually," marking each edge with the label of the nonleaf node v whose path π_v contains that edge.

We claim that the paths π_v for all nonleaf v are edge-disjoint, and thus the sum of the path lengths is bounded by the number of total edges in the tree, hence $O(n)$. To show this, we consider what we term "right-leaning" and "left-leaning" edges (i.e., those going from a parent to a right, respectively left, child). A particular right-leaning edge e can only be part of the path π_v for node v that is the parent in the relationship represented by e. Left-leaning edges can be partitioned by considering the leaf that is reached if continuing down leftward until reaching a leaf. Each nonleaf node only uses left-leaning edges in the group leading to that nonleaf node's inorder successor. Since each nonleaf node must have a different inorder successor, no two such paths can contain the same left-leaning edge. We conclude that the bottom-up construction of heap T takes $O(n)$ time. ■

Adaptable Priority Queues



Items and Priority Queues

- An item stores a (key, value) pair
- Item fields:
  - _key: the key associated with this item
  - _value: the value paired with the key associated with this item

- Priority Queue ADT:
  - add(k, x)
    inserts an item with key k and value x
  - remove_min()
    removes and returns the item with smallest key
  - min()
    returns, but does not remove, an item with smallest key
  - len(P), is_empty()

Example

- Online trading system where orders to purchase and sell a stock are stored in two priority queues (one for sell orders and one for buy orders) as (p,s) entries:
  - The key, p, of an order is the price
  - The value, s, for an entry is the number of shares
  - A buy order (p,s) is executed when a sell order (p',s') with price $p' \le p$ is added (the execution is complete if $s' \ge s$)
  - A sell order (p,s) is executed when a buy order (p',s') with price $p' \ge p$ is added (the execution is complete if $s' \ge s$)
- What if someone wishes to cancel their order before it executes?
- What if someone wishes to update the price or number of shares for their order?

Methods of the Adaptable Priority Queue ADT

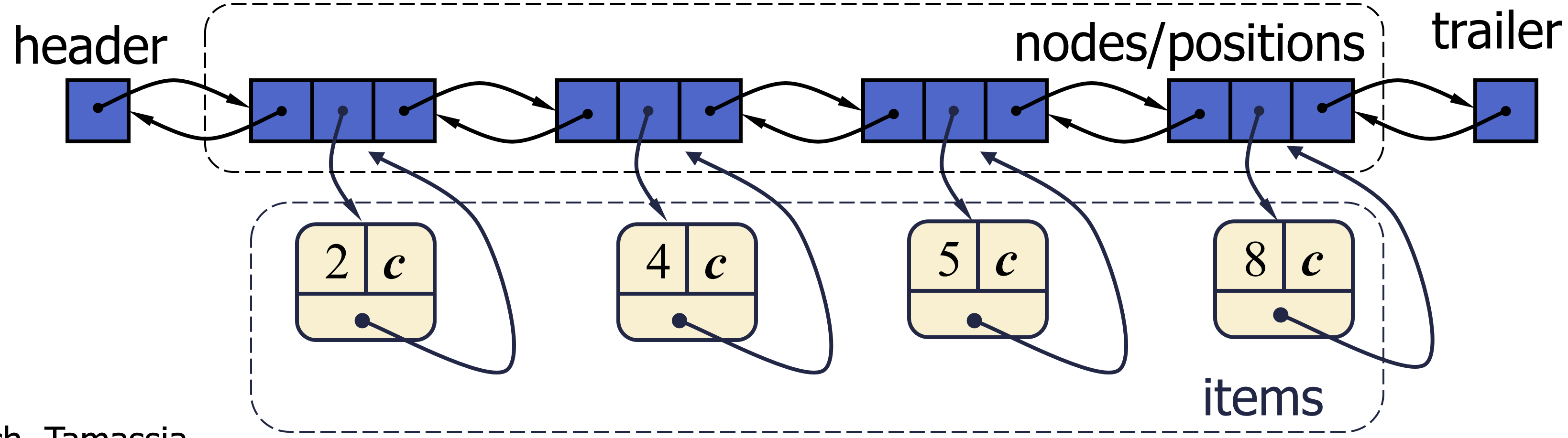
- remove(loc): Remove from P and return item e for locator loc.
- update(loc,k,v): Replace the key-value pair for locator, loc, with (k,v).

Locators

- A locator-aware item identifies and tracks the location of its (key, value) object within a data structure
- Intuitive notion:
  - Coat claim check
  - Valet claim ticket
  - Reservation number
- Main idea:
  - Since items are created and returned from the data structure itself, it can return location-aware items, thereby making future updates easier

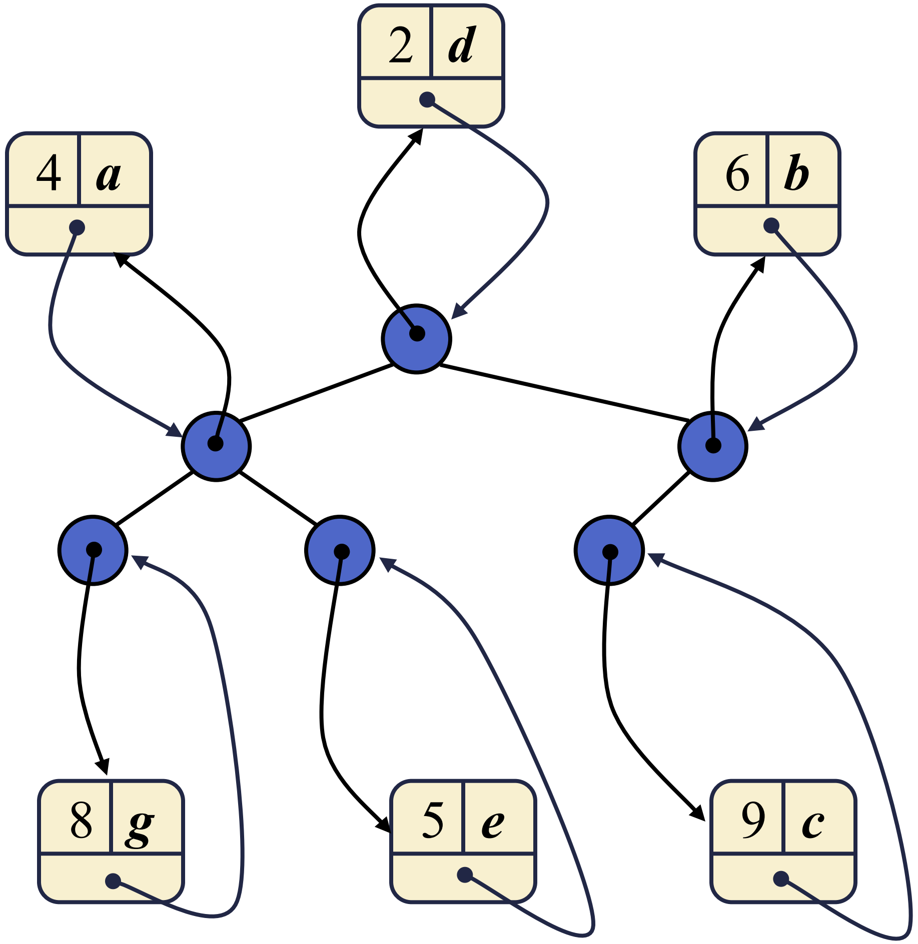
List Implementation

- A location-aware list item is an object storing
  - key
  - value
  - position (or rank) of the item in the list
- In turn, the position (or array cell) stores the entry
- Back pointers (or ranks) are updated during swaps



Heap Implementation

- A location-aware heap item is an object storing
  - key
  - value
  - position of the item in the underlying heap
- In turn, each heap position stores an item
- Back pointers are updated during item swaps



Performance

- Improved times thanks to location-aware items are highlighted in red

Method	**Unsorted List**	**Sorted List**	**Heap**
len, is_empty	$O(1)$	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
remove_min	$O(n)$	$O(1)$	$O(\log n)$
remove	$\boldsymbol{O}(1)$	$\boldsymbol{O}(1)$	$\boldsymbol{O}(\log \boldsymbol{n})$
update	$\boldsymbol{O}(1)$	$O(n)$	$\boldsymbol{O}(\log \boldsymbol{n})$