Priority Queues

What is Priority Queue?

A collection of prioritized elements that allows arbitrary element insertion, and allows the removal of the element that has first priority.

When an element is added to a priority queue, the user designates its priority by providing an associated key.

Applications: OS scheduling, Packet routing, customer service, air traffic scheduling, notifications on our phones etc etc.

Priority Queue ADT

P.add(k, v): Insert an item with key k and value v into priority queue P.

P.min( ): Return a tuple, (k,v), representing the key and value of an item in priority queue P with minimum key (but do not remove the item); an error occurs if the priority queue is empty.

P.remove_min( ): Remove an item with minimum key from priority queue P, and return a tuple, (k,v), representing the key and value of the removed item; an error occurs if the priority queue is empty.

P.is_empty( ): Return True if priority queue P does not contain any items.

len(P): Return the number of items in priority queue P.

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"	{ }

Total Order Relations

Keys in a priority queue can be arbitrary objects on which an order is defined

Two distinct entries in a priority queue can have the same key

Mathematical concept of total order relation $\leq$

Reflexive property:

$x \leq x$

Antisymmetric property:

$x \leq y \text{ and } y \leq x \Rightarrow x = y$

Transitive property:

$x \leq y \text{ and } y \leq z \Rightarrow x \leq z$

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

```
1  class PriorityQueueBase:
2    """Abstract base class for a priority queue."""
3
4    class _Item:
5      """Lightweight composite to store priority queue items."""
6      __slots__ = '_key', '_value'
7
8      def __init__(self, k, v):
9        self._key = k
10       self._value = v
11
12     def __lt__(self, other):
13       return self._key < other._key       # compare items based on their keys
14
15   def is_empty(self):                     # concrete method assuming abstract len
16     """Return True if the priority queue is empty."""
17     return len(self) == 0
```

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

Unsorted List Implementation

```
1  class UnsortedPriorityQueue(PriorityQueueBase): # base class defines _Item
2    """A min-oriented priority queue implemented with an unsorted list."""
3
4    def _find_min(self):                        # nonpublic utility
5      """Return Position of item with minimum key."""
6      if self.is_empty():                       # is_empty inherited from base class
7        raise Empty('Priority queue is empty')
8      small = self._data.first()
9      walk = self._data.after(small)
10     while walk is not None:
11       if walk.element( ) < small.element( ):
12         small = walk
13       walk = self._data.after(walk)
14     return small
15
16   def __init__(self):
17     """Create a new empty Priority Queue."""
18     self._data = PositionalList( )
19
20   def __len__(self):
21     """Return the number of items in the priority queue."""
22     return len(self._data)
```

Unsorted List Implementation

```
23
24    def add(self, key, value):
25      """Add a key-value pair."""
26      self._data.add_last(self._Item(key, value))
27
28    def min(self):
29      """Return but do not remove (k,v) tuple with minimum key."""
30      p = self._find_min( )
31      item = p.element( )
32      return (item._key, item._value)
33
34    def remove_min(self):
35      """Remove and return (k,v) tuple with minimum key."""
36      p = self._find_min( )
37      item = self._data.delete(p)
38      return (item._key, item._value)
```

Sorted List Implementation

```
1  class SortedPriorityQueue(PriorityQueueBase): # base class defines _Item
2    """A min-oriented priority queue implemented with a sorted list."""
3
4    def __init__(self):
5      """Create a new empty Priority Queue."""
6      self._data = PositionalList( )
7
8    def __len__(self):
9      """Return the number of items in the priority queue."""
10     return len(self._data)
11
12   def add(self, key, value):
13     """Add a key-value pair."""
14     newest = self._Item(key, value)              # make new item instance
15     walk = self._data.last( )          # walk backward looking for smaller key
16     while walk is not None and newest < walk.element( ):
17       walk = self._data.before(walk)
18     if walk is None:
19       self._data.add_first(newest)               # new key is smallest
20     else:
21       self._data.add_after(walk, newest)         # newest goes after walk
22
```

Sorted List Implementation

```
23    def min(self):
24      """Return but do not remove (k,v) tuple with minimum key."""
25      if self.is_empty( ):
26        raise Empty('Priority queue is empty.')
27      p = self._data.first( )
28      item = p.element( )
29      return (item._key, item._value)
30
31    def remove_min(self):
32      """Remove and return (k,v) tuple with minimum key."""
33      if self.is_empty( ):
34        raise Empty('Priority queue is empty.')
35      item = self._data.delete(self._data.first( ))
36      return (item._key, item._value)
```

Worst-case running times for a priority queue of size n using two types of lists

Operation	Unsorted List	Sorted List
len	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
remove_min	$O(n)$	$O(1)$

Can we do better?

Yes, a more efficient realization of a priority queue can be implemented using a data structure called a binary heap.

Its performance for both insertions and removals in O(log n)

Binary heap achieves this efficiency by using the structure of a binary tree to find a compromise between elements being entirely unsorted and perfectly sorted.

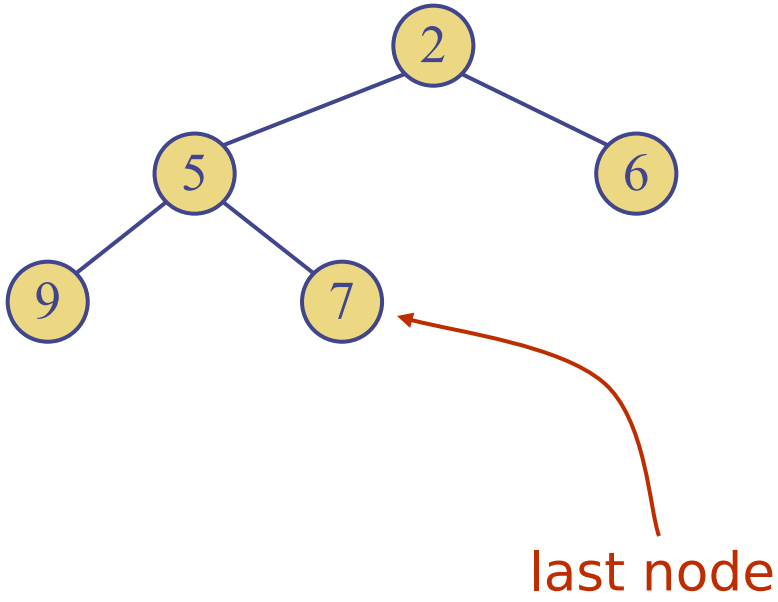
Heaps

A heap is a binary tree storing keys at its nodes and satisfying the following **two properties**:

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

Relational Property:

Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$



Heaps

Structural Property

Complete Binary Tree Property: A heap T with height h is a complete binary tree if levels 0, 1, 2, . . . , h − 1 of T have the maximum number of nodes possible (namely, level i has 2i nodes, for $0 \leq i \leq h - 1$) and the remaining nodes at level h reside in the leftmost possible positions at that level

Height of a Heap

Theorem: A heap T storing n entries has height h = floor(log n)

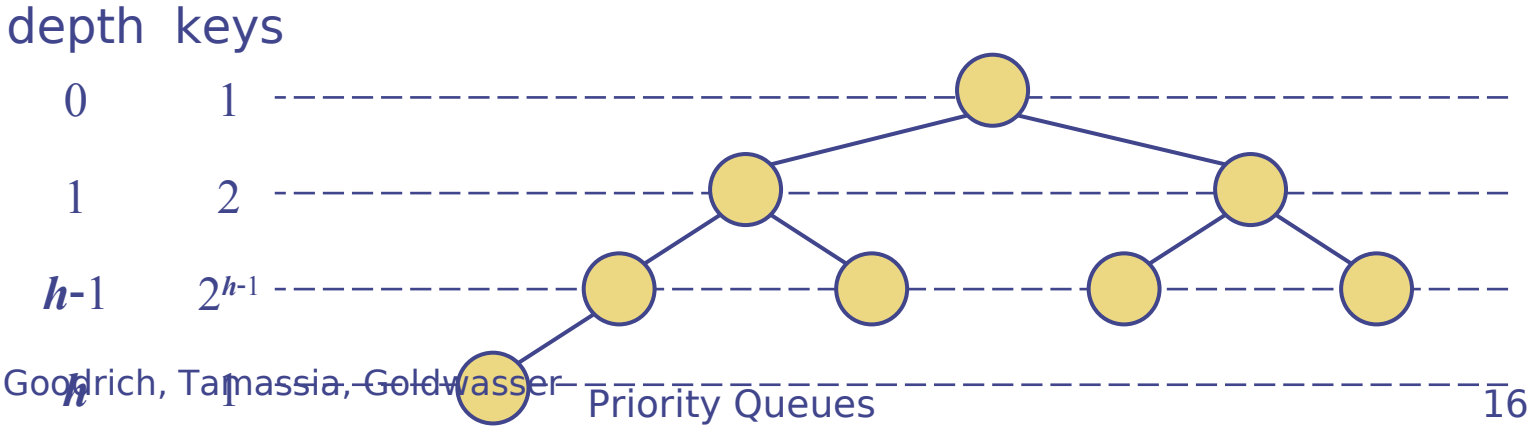
Proof: (we apply the complete binary tree property)

Let h be the height of a heap storing n keys

Total nodes upto h-1 is $1+ 2 + 4 + \cdots + 2^{h-1} = 2^h - 1$

Thus, $n \geq 2^h - 1 + 1$ (lower bound) and $n \leq 2^h - 1 + 2^h = 2^{h+1} - 1$ (upper bound)

Apply log on these inequalities and rearranging : $\log(n+1) - 1 \leq h \leq \log n$

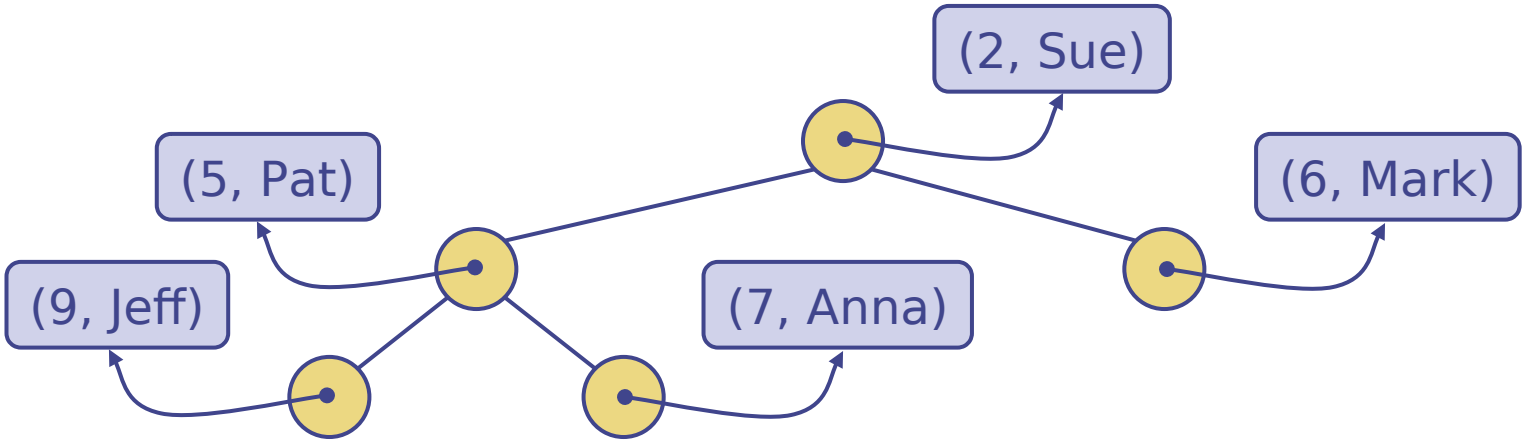


Heaps and Priority Queues

We can use a heap to implement a priority queue

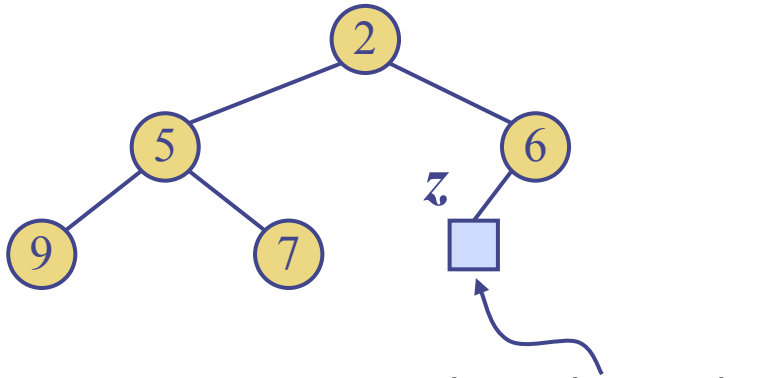
We store a (key, element) item at each internal node

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

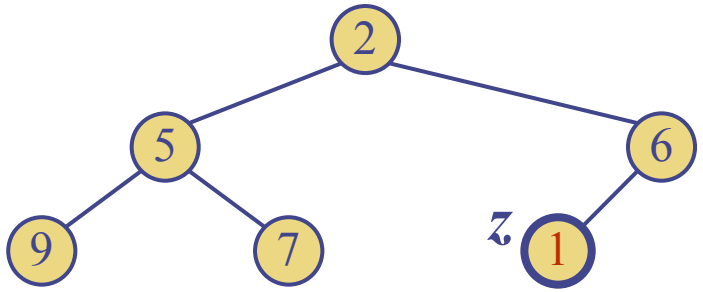


The insertion algorithm consists of three steps

a)Find the insertion node z (the new last node)

b)Store k at z

c)Restore the heap-order property



Up-heap bubbling

- 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

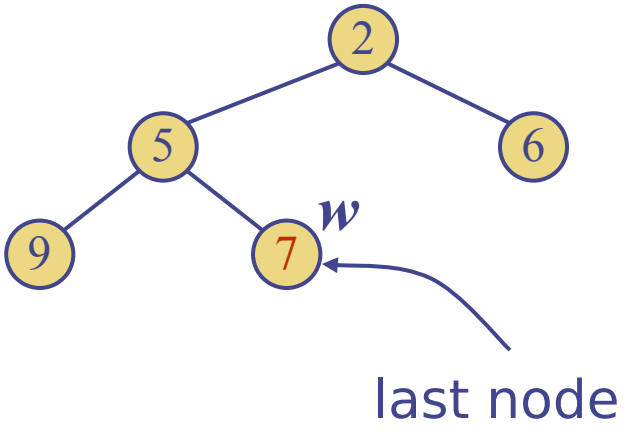
Method remove_min of the priority queue ADT corresponds to the removal of the root key from the heap

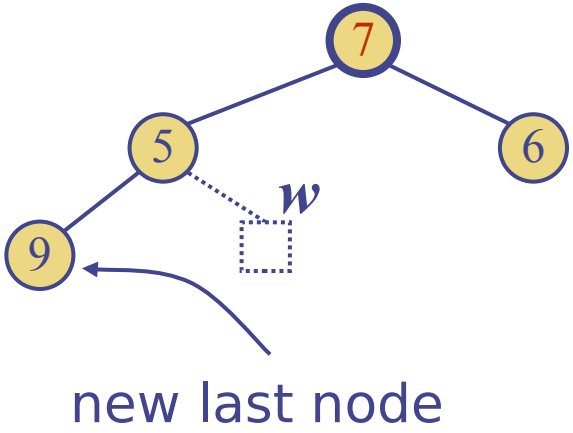
The removal algorithm consists of three steps

a)Replace the root key with the key of the last node *w*

b)Remove *w* (old root node)

c)Restore the heap-order property





Down-heap Bubbling

a)After replacing the root key with the key k of the last node, the heap-order property may be violated

b)Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root

c)Upheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k

d)Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time

Array-based Heap Implementation

We can represent a heap with n keys by means of an array of length n

For the node at rank i

the left child is at rank $2i + 1$

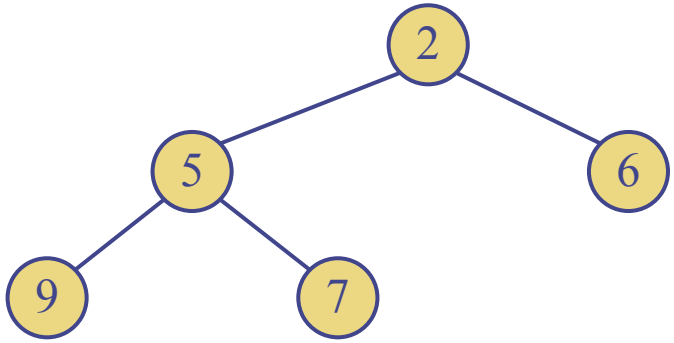
the right child is at rank $2i + 2$

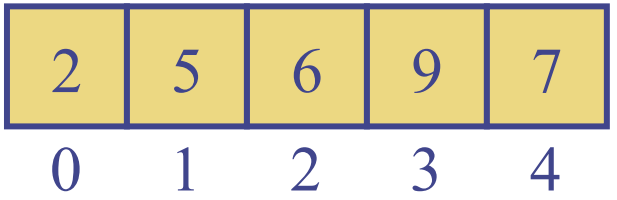
Links between nodes are not explicitly stored

Operation add corresponds to inserting at rank $n + 1$

Operation remove_min corresponds to removing at rank n

Yields in-place heap-sort





Python Heap Implementation

```
1  class HeapPriorityQueue(PriorityQueueBase): # base class defines _Item
2    """A min-oriented priority queue implemented with a binary heap."""
3    #------------------------- nonpublic behaviors -------------------------
4    def _parent(self, j):
5      return (j-1) // 2
6
7    def _left(self, j):
8      return 2*j + 1
9
10   def _right(self, j):
11     return 2*j + 2
12
13   def _has_left(self, j):
14     return self._left(j) < len(self._data)      # index beyond end of list?
15
16   def _has_right(self, j):
17     return self._right(j) < len(self._data)     # index beyond end of list?
18
19   def _swap(self, i, j):
20     """Swap the elements at indices i and j of array."""
21     self._data[i], self._data[j] = self._data[j], self._data[i]
22
23   def _upheap(self, j):
24     parent = self._parent(j)
25     if j > 0 and self._data[j] < self._data[parent]:
26       self._swap(j, parent)
27       self._upheap(parent)                      # recur at position of parent
```

Python Heap Implementation

```
29    def _downheap(self, j):
30      if self._has_left(j):
31        left = self._left(j)
32        small_child = left                    # although right may be smaller
33        if self._has_right(j):
34          right = self._right(j)
35          if self._data[right] < self._data[left]:
36            small_child = right
37        if self._data[small_child] < self._data[j]:
38          self._swap(j, small_child)
39          self._downheap(small_child)         # recur at position of small child
```

Python Heap Implementation

```
40    #------------------------------ public behaviors ------------------------------
41    def __init__(self):
42      """Create a new empty Priority Queue."""
43      self._data = [ ]
44
45    def __len__(self):
46      """Return the number of items in the priority queue."""
47      return len(self._data)
48
49    def add(self, key, value):
50      """Add a key-value pair to the priority queue."""
51      self._data.append(self._Item(key, value))
52      self._upheap(len(self._data) - 1)          # upheap newly added position
53
54    def min(self):
55      """Return but do not remove (k,v) tuple with minimum key.
56
57      Raise Empty exception if empty.
58      """
59      if self.is_empty( ):
60        raise Empty('Priority queue is empty.')
61      item = self._data[0]
62      return (item._key, item._value)
63
64    def remove_min(self):
65      """Remove and return (k,v) tuple with minimum key.
66
67      Raise Empty exception if empty.
68      """
69      if self.is_empty( ):
70        raise Empty('Priority queue is empty.')
71      self._swap(0, len(self._data) - 1)         # put minimum item at the end
72      item = self._data.pop( )                   # and remove it from the list;
73      self._downheap(0)                          # then fix new root
74      return (item._key, item._value)
```

Adaptable Priority Queues

Discussed in the class

Exercises

R-9.1 How long would it take to remove the $\lceil \log n \rceil$ smallest elements from a heap that contains n entries, using the remove_min operation?

R-9.2 Suppose you label each position p of a binary tree T with a key equal to its preorder rank. Under what circumstances is T a heap?

R-9.3 What does each remove_min call return within the following sequence of priority queue ADT methods: add(5,A), add(4,B), add(7,F), add(1,D), remove_min( ), add(3,J), add(6,L), remove_min( ), remove_min( ), add(8,G), remove_min( ), add(2,H), remove_min( ), remove_min( )?

Exercises

R-9.10 At which positions of a heap might the third smallest key be stored?

R-9.11 At which positions of a heap might the largest key be stored?

R-9.12 Consider a situation in which a user has numeric keys and wishes to have a priority queue that is *maximum-oriented*. How could a standard (min-oriented) priority queue be used for such a purpose?

Exercises

R-9.16 Is there a heap H storing seven entries with distinct keys such that a preorder traversal of H yields the entries of H in increasing or decreasing order by key? How about an inorder traversal? How about a postorder traversal? If so, give an example; if not, say why.

R-9.17 Let H be a heap storing 15 entries using the array-based representation of a complete binary tree. What is the sequence of indices of the array that are visited in a preorder traversal of H? What about an inorder traversal of H? What about a postorder traversal of H?

Exercises

R-9.19 Bill claims that a preorder traversal of a heap will list its keys in nondecreasing order. Draw an example of a heap that proves him wrong.

R-9.20 Hillary claims that a postorder traversal of a heap will list its keys in nonincreasing order. Draw an example of a heap that proves her wrong.

References

Data Structures and Algorithms in Python
Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

Introduction to Algorithms
Leiserson,Stein,Rivest,Cormen

Algorithms, 4th Edition
Robert Sedgewick and Kevin Wayne

Few Images from the internet