PRIORITY QUEUES

School of Artificial Intelligence

PREVIOUSLY ON DS&A

- Priority Queues
- Implementation of Priority Queues
- Heaps
- Implementation of Heaps

TREES

- Proper binary tree （真二叉树）：二叉树的每一个结点都有0个或2个子节点
- Full binary tree（满二叉树）：除叶子结点外，所有内部结点都有2个子结点；所有叶子结点的深度都相同
- Complete binary tree（完整二叉树）：对于高为h的真二叉树，其从0层到h-1都有2^i个结点（i为层数）；对于第h层，如果没有2^h个结点的话，则所有结点都集中在第h层的最左面

PRIORITY QUEUES （优先队列）

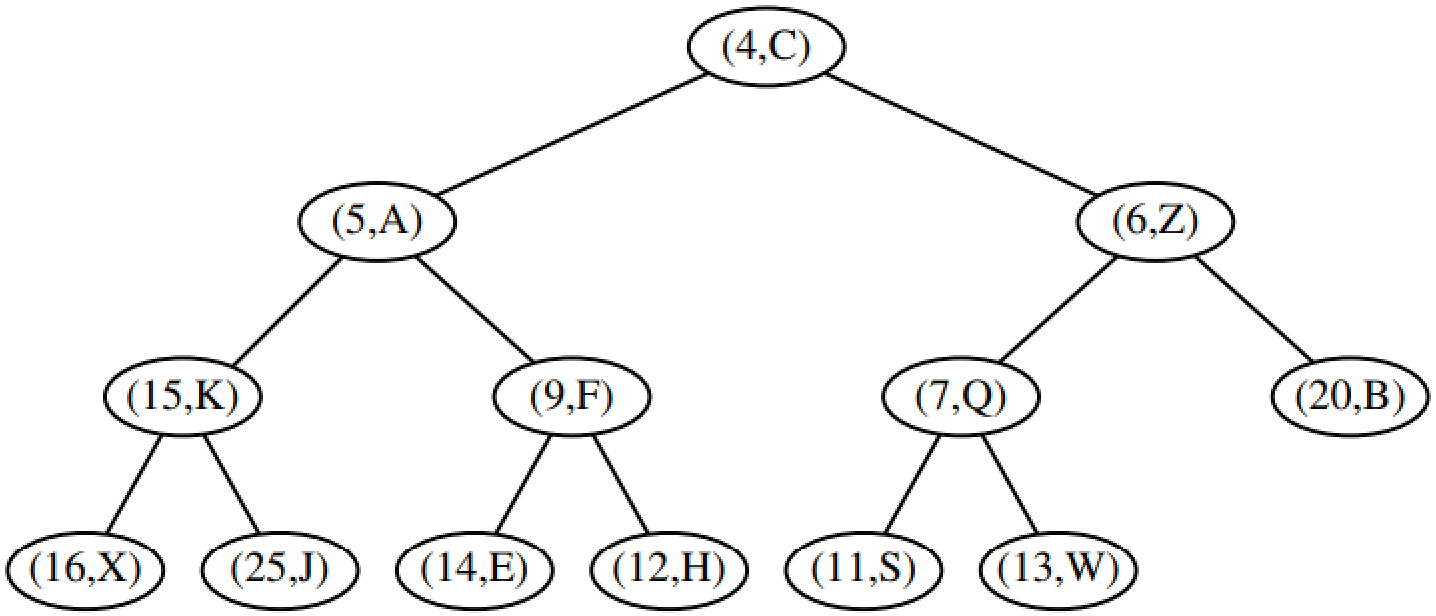
- Collection of prioritized elements
  - Arbitrary element insertion
  - Removal of the element that has first priority
  - When an element is added, a priority can be assigned to it with a **key**
  - Element with the minimum key will be removed next from the queue
  - **Key**s can be other data types, as long as there is a way to compare them
  - E.g. a < b for instances a and b
- Implementation of Priority Queues
  - Based on Positional list
    - Can you do with an array? What problems can arise?
  - Unsorted list
    - Add – O(1), remove_min - O(n)
  - Sorted list
    - Add – O(n), remove_min – O(1)

HEAP（堆）

- Heap: a binary tree that stores a collection of items at its positions
  - A relational property defined in terms of the way keys are stored in T
  - A structural property defined in terms of the shape of T itself
- Relational property (**heap order property**): In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p's parent
- 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 (level i has 2^i nodes, for 0<= i <= h-1) and the remaining nodes at level h reside in the lfeftmost possible positions at that level

HEAP（堆）

- Complete
  - Levels 0, 1, and 2 are full
  - 6 nodes in level 3 are in the six leftmost possible positions at that level
- An alternative definition
  - If we are to store a complete binary tree T with n elements in an array A , then its 13 entries would be stored from A[0] to A[n-1]

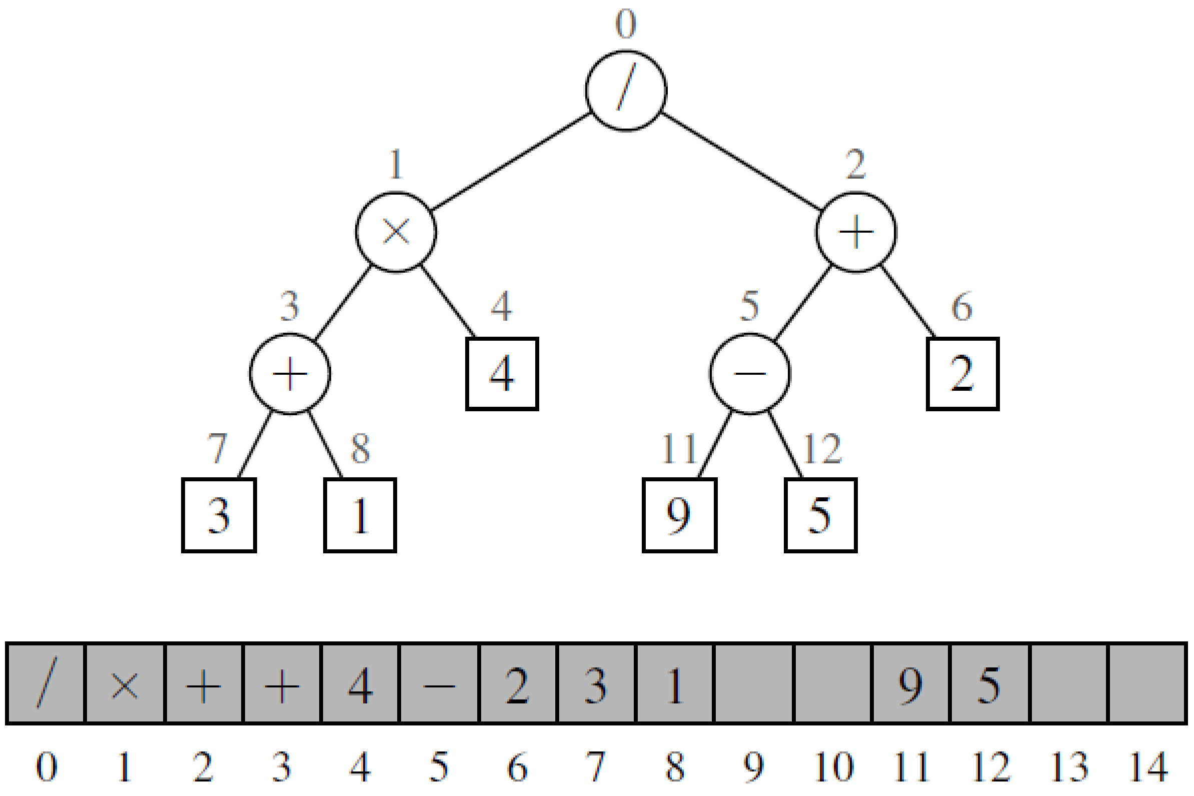


THIS LECTURE

- Heap implementation
- Heap Construction
- Sorting with a priority queue
- Adaptable priority queues

ARRAY-BASED REPRESENTATION OF A COMPLETE BINARY TREE

- Array based representation of a binary tree
- For every position p of T, let f(p) be the integer defined as follows
  - If p is the root of T, then f(p) = 0
  - If p is the left child of position q, then f(p) = 2f(q) + 1
  - If p is the right child of position q, then f(p) = 2f(q) + 2



ARRAY-BASED REPRESENTATION OF A COMPLETE BINARY TREE

- Array based representation of the heap structure

(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(8,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

- Array-based vs. node-based (linked structure)
- add() and remove_min()
  - Node-based: iterating through the positional list
  - Array-based:
- However, for array-based, complexities becomes **amortised**
  - Why?

HEAP IMPLEMENTATION

- Array-based representation
- A python list for items
- _parent(), _left(), _right achieved by maths
- _has_left(): check if the current node has a left child node
- swap(): exchange elements stored in i and j

```
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]
```

HEAP IMPLEMENTATION

- _upheap(): recursive function to restore the heap order property
- _downheap(): recursive function to restore the heap order property

```
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
```

HEAP IMPLEMENTATION

- Constructor: initialise array
- add(): append() to keep the binary complete tree property, then upheap() to restore the heap order property
- min(): returns the item with the minimal key
- remove_min(): swap the root of the heap with the end of the heap, delete the end, downheap() to restore the heap order property

```
#----------------------------- 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)
```

HEAP-BASED PRIORITY QUEUE

- Heap T has n nodes, each node stores a key-value pair
- The height of heap T is O(log n), since T is complete
- The min() operation runs in O(1)
- Locating the last position of a heap is O(1) for array based representation, or O(logn) for linked structure representation
- The worst case upheap() and downheap() performs a number of swaps equal to the height of T

Operation	Running Time
len(P), P.is_empty()	$O(1)$
P.min()	$O(1)$
P.add()	$O(\log n)^*$
P.remove_min()	$O(\log n)^*$

*amortized, if array-based

BOTTOM UP HEAP CONSTRUCTION

- Start with an empty heap, n successive calls to the add() operation will run in O(n log n) time.
- However, if all n key-value pairs are given at first, a bottom-up construction method can run in O(n) time.
- Example: we assume a heap with n, such that $n = 2^{h+1} -1$, where h is the height of the heap $h = \log(n+1) -1$
- 1. construct (n+1)/2 elementary heap storing one entry each
- 2. construct (n+1)/4 heaps, each storing three entries, by joining pairs of elementary heaps and adding a new entry, perform downheap() if necessary
- 3. in the generic i^{th} step, 2<=i<=h, form $(n+1)/2^i$ heaps, each storing 2^i - 1entries, perform downheap() to restore the heap order property

BOTTOM UP HEAP CONSTRUCTION

- Add 16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14 (15 elements) in order to construct a heap
- h = floor(log n) = 3
- construct (n+1)/2 elementary heap storing one entry each
- In the generic i^{th} step, 2<=i<=h, form $(n+1)/2^i$ heaps, each storing 2^i -1entries, perform downheap() to restore the heap order property

BOTTOM UP HEAP CONSTRUCTION IN PYTHON

- Store all n items in arbitrary order within the array
- _heapify() to turn _data into a heap

```
def _downheap(self, j):
  if self._has_left(j):
    left = self._left(j)
    small_child = left                                  #
    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)                       #
```

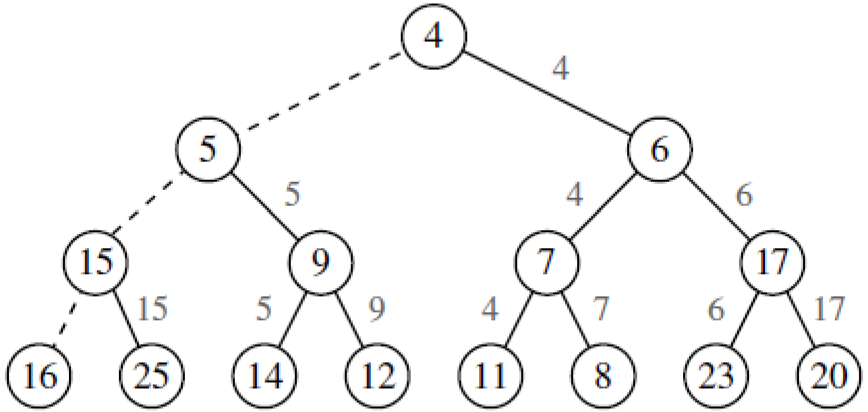
```
def __init__(self, contents=()):
  """Create a new priority queue.

  By default, queue will be empty. If contents is given, it should be as an
  iterable sequence of (k,v) tuples specifying the initial contents.
"""
  self._data = [ self._Item(k,v) for k,v in contents ]     # empty by default
  if len(self._data) > 1:
    self._heapify()

def _heapify(self):
  start = self._parent(len(self) - 1)        # start at PARENT of last leaf
  for j in range(start, -1, -1):             # going to and including the root
    self._downheap(j)
```

ASYMPTOTIC ANALYSIS OF BOTTOM UP HEAP CONSTRUCTION

- **Bottom up construction of a heap with n entries takes O(n) time, assuming two keys can be compared in O(1) time**
- Primary cost: downheap() at each nonleaf position.
- Let P_v denote the path of T from nonleaf node v to its inorder successor leaf, P_v is proportional to the height of the subtree roted at v.
- Total running time therefore the sum of the sizes of paths
- Paths are edge-disjoint, and therefore bounded by the number of total edges, hence O(n)



SORTING WITH A PRIORITY QUEUE

- Priority queue ADT: any type of object can be used as a key, as long as they can be compared with the comparison operator <
- Comparison operators need to be irreflexive and transitive

```
def pq_sort(C):
  """Sort a collection of elements stored in a positional list."""
  n = len(C)
  P = PriorityQueue()
  for j in range(n):
    element = C.delete(C.first())
    P.add(element, element)          # use element as key and value
  for j in range(n):
    (k,v) = P.remove_min()
    C.add_last(v)                    # store smallest remaining element in C
```

k3.

- Use priori nents.
- Insert all e nove_min to get an increc

SORTING WITH A PRIORITY QUEUE

- pq_sort(): works OK, but its complexity?
- Depends on add() and remove_min()
- Selection-Sort: implement P with an unsorted list
  - add() takes $O(n)$ time in total since it is $O(1)$ for add()
  - remove_min(): selecting element to dequeue()
  - Total running time: $O(n + (n-1) + (n-2) + \ldots + 1) = O(n^2)$
- Insertion-Sort: implement P with a sorted list
  - remove_min() takes $O(n)$ time in total since it is $O(1)$ for each remove_min()
  - add(): finding the proper place to add takes $O(n^2)$ time in total

HEAP SORT

- Priority queue implemented with a heap
  - Logarithmic time for operations
- pq_sort() with heap-based implementation
  - add(): i^{th} add operation takes O(log i) time, in total it takes O(n log n) time
  - remove_min(): O(log (n-j+1)) for the j^{th} remove_min() operation, in total it takes O(n log n) time
  - This is known as heap-sort
- The heap-sort algorithm sorts a collection C of n elements in O(n log n) time, assuming two elements of C can be compared in O(1) time

HEAP SORT

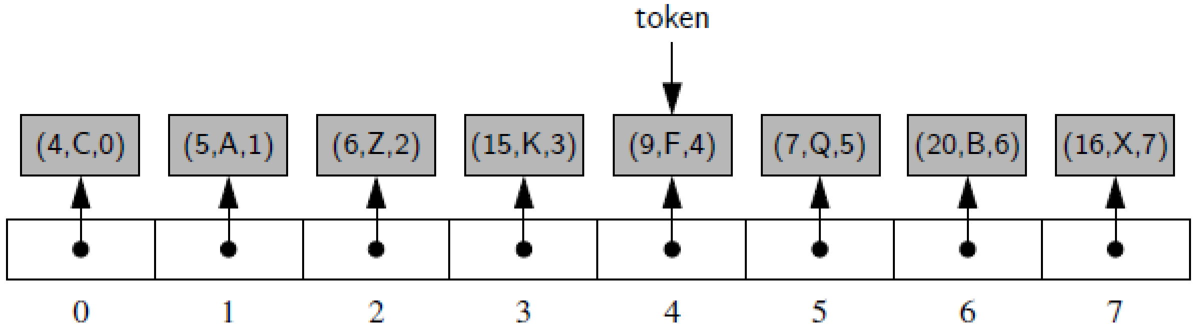
- Implementing in-place heap-sort （原地堆排序）
- Need to modify the algorithm
- Maximum-oriented heap: each position’s key being at least as large as its children. At any time during the execution, use the left portion of C, up to a certain index i-1, to store the entries of the heap, and the right portion of C, from i to n-1, to store the elements of the sequence
- In the first phase of the algorithm, start with an empty heap and move the boundary between the heap and the sequence from left to right, one step at a time.
- In the second phase of the algorithm, we start with an empty sequence and move the boundary between the heap and the sequence from right to left, one step at a time.

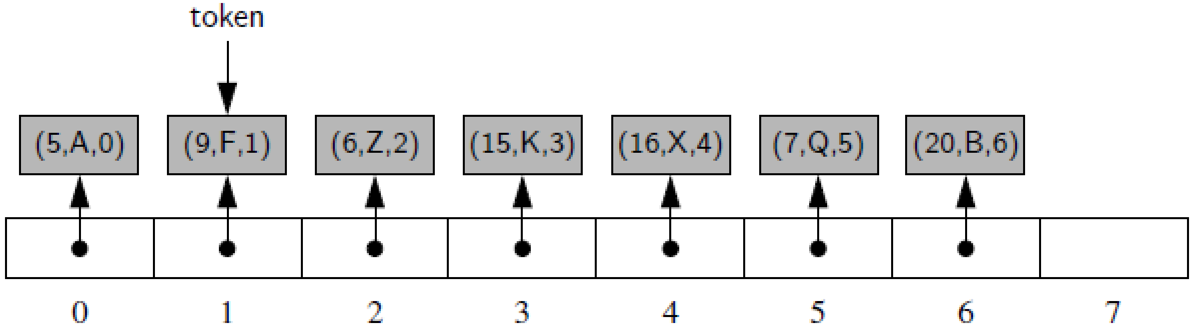
ADAPTABLE PRIORITY QUEUES

- Priority queue ADT is sufficient for most basic applications. However, the following situations are not accounted for:
- A person waiting in queue may want to drop out, requesting to be removed from the waiting list.
  - Need a remove() operation
- An element may suddenly have a higher priority and needs to be placed in its rightful place.
  - Need a update() operation
- The above behaviours make the priority queue adaptable to any situations we can think of

ADAPTABLE PRIORITY QUEUES

- Token points to (9, F) with index 4
- remove_min() to remove (4, C)





ADAPTABLE PRIORITY QUEUES

- Locators class to record index
- P.update(loc, k, v): replace key and value for the item identified by the locator loc
- P.remove(loc): remove the item identified by locator loc from the priority queue and return its (key,value) pair

```
class AdaptableHeapPriorityQueue(HeapPriorityQueue):
  """A locator-based priority queue implemented with a binary heap."""

  #------------------------------ nested Locator class ------------------------------
  class Locator(HeapPriorityQueue._Item):
    """Token for locating an entry of the priority queue."""
    __slots__ = '_index'                  # add index as additional field

    def __init__(self, k, v, j):
      super().__init__(k,v)
      self._index = j

  #------------------------------ nonpublic behaviors ------------------------------
  # override swap to record new indices
  def _swap(self, i, j):
    super()._swap(i,j)                    # perform the swap
    self._data[i]._index = i              # reset locator index (post-swap)
    self._data[j]._index = j              # reset locator index (post-swap)

  def _bubble(self, j):
    if j > 0 and self._data[j] < self._data[self._parent(j)]:
      self._upheap(j)
    else:
      self._downheap(j)
```

ADAPTABLE PRIORITY QUEUES

- add(): to add key value pairs to the priority queue
- update(): update the location of the key-value pair

```
def add(self, key, value):
  """Add a key-value pair."""
  token = self.Locator(key, value, len(self._data)) # initiaize locator index
  self._data.append(token)
  self._upheap(len(self._data) - 1)
  return token

def update(self, loc, newkey, newval):
  """Update the key and value for the entry identified by Locator loc."""
  j = loc._index
  if not (0 <= j < len(self) and self._data[j] is loc):
    raise ValueError('Invalid locator')
  loc._key = newkey
  loc._value = newval
  self._bubble(j)
```

ADAPTABLE PRIORITY QUEUES

- remove(): remove the element pointed by the location from the priority queue

```
def remove(self, loc):
  """Remove and return the (k,v) pair identified by Locator loc."""
  j = loc._index
  if not (0 <= j < len(self) and self._data[j] is loc):
    raise ValueError('Invalid locator')
  if j == len(self) − 1:                   # item at last position
    self._data.pop( )                      # just remove it
  else:
    self._swap(j, len(self)−1)             # swap item to the last position
    self._data.pop( )                      # remove it from the list
    self._bubble(j)                        # fix item displaced by the swap
  return (loc._key, loc._value)
```

Operation	Running Time
len(P), P.is_empty(), P.min()	$O(1)$
P.add(k,v)	$O(\log n)^*$
P.update(loc, k, v)	$O(\log n)$
P.remove(loc)	$O(\log n)^*$
P.remove_min()	$O(\log n)^*$

*amortized with dynamic array

QUIZ OF THE WEEK

- 3 lottery tickets
- 1 of them wins the jackpot
- You are asked to pick one of them
- I now discard one of the remaining 2 tickets and tell you the discarded ticket does not win the lottery
- You are given a chance to pick again from the 2 tickets
- What would you do?

THANKS

See you in the next session!