CSCI 204: Data Structures & Algorithms

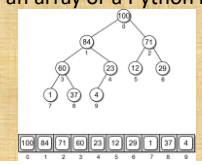
Revised by Xiannong Meng based on textbook author's notes

Binary Tree Application Operations in Heaps

Revised based on textbook author's notes.

Heap Implementation

- While a heap is a binary tree, it's seldom implemented as a dynamically linked structure.
- Use a sequence to physically store the nodes.
- We could use an array or a Python list



Heap – Node Access

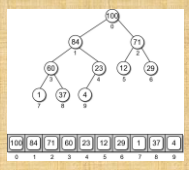
- The complete tree will never contain "holes".
- The root will always be at position 0.
- Its two children will always occupy positions 1 and 2.
- The children of any node will always occupy the positions in the same relation to their parent.

Heap – Node Access

- Given the array index *i*

```
parent = (i-1) // 2
left = 2 * i + 1
right = 2 * i + 2
```

- A child link is null if the index is out of range.



Heap – Class Definition

listheap.py

```
class MaxHeap :
  def __init__( self, max_size = 16 ):
    self._elements = [None for i in range(max_size)]
    self._count = 0
    self._max_size = max_size

  def __len__( self ):
    return self._count

  def capacity( self ):
    return len( self._elements )
# ...
```

Heap – Class Definition

listheap.py

```
class MaxHeap :
# ...
  def add( self, value ):
    if self._count >= self.capacity():
        self._expand()   # double the capacity and copy the content

    # Add the new value to the end of the list.
    self._elements[ self._count ] = value
    self._count += 1
    # Sift the new value up the tree.
    self._sift_up( self._count - 1 )

  def _sift_up( self, ndx ):
    if ndx > 0 :
      parent = ndx // 2
      if self._elements[ndx] > self._elements[parent] :
        tmp = self._elements[ndx]
        self._elements[ndx] = self._elements[parent]
        self._elements[parent] = tmp
        self._sift_up( parent )
```


Heap – Class Definition

listheap.py

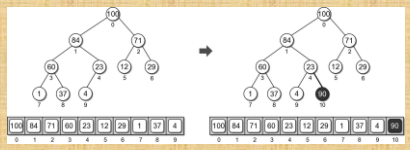
```
class MaxHeap :
# ...
  def extract( self ):
    assert self._count > 0,
           "Cannot extract from an empty heap."
    value = self._elements[0]
    self._count -= 1
    self._elements[0] = self._elements[ self._count ]
    self._sift_down( 0 )
    return value
```


Your Exercise

- Following the pattern of function sift-up(), write the function sift-down() when the top (root) item is removed.

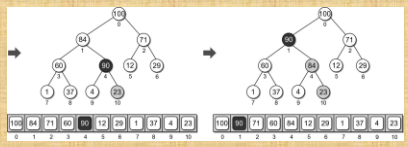
Heap Example

- Physical view of adding value 90.



Heap Example

- Physical view of adding value 90.



Heap Analysis

- Assume a heap containing *n* elements:
  - Insertion: *O(log n)*
  - Extraction: *O(log n)*
- Why?
- Height of the heap is *O(log n)*

The Heapsort

- The simplicity and efficiency of the heap structure can be applied to the sorting problem.
  - Build a heap from a sequence of unsorted keys.
  - Extract the keys from the heap to create a sorted sequence.
- Very efficient: *O(n log n)*

14

Heapsort Implementation

- A simple implementation is provided below.

```
def simple_heap_sort( the_seq ):
  # Create an array-based max-heap.
  n = len(the_seq)
  heap = MaxHeap( n )

  # Build a max-heap from the list of values.
  for item in the_seq :
    heap.add( item )

  # Extract each value from the heap and store
  # them back into the list.
  for i in range( n-1, -1, -1 ) : # small to large
  # for i in range( n ) :  # large to small
    theSeq[i] = heap.extract()
```

15