CS 2412 Data Structures

Chapter 7

Heaps

A binary tree is

- a complete tree if it has maximum number of entries for its height.
- a nearly complete if it has the minimum height for its nodes and all nodes in the last level are found on the left.

Definition

A heap is a binary tree structure with the following properties:

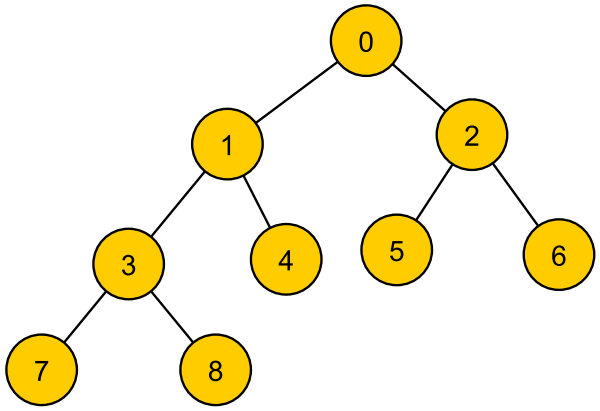
1. The tree is complete or nearly complete.
2. The key value of each node is greater than or equal to the key values in each of its descendants.

From the definition we see that

- The key value in the root of a heap is the maximum value of keys.
- Any subtree of a heap is a heap.

Usually a heap is implemented in an array. So we can define a heap in another way.

A heap is a list in which each entry contains a key, and for all positions k in the list, the key at position k is at least as large as the keys in positions $2k + 1$ and $2k + 2$, provided these positions exist in the list.

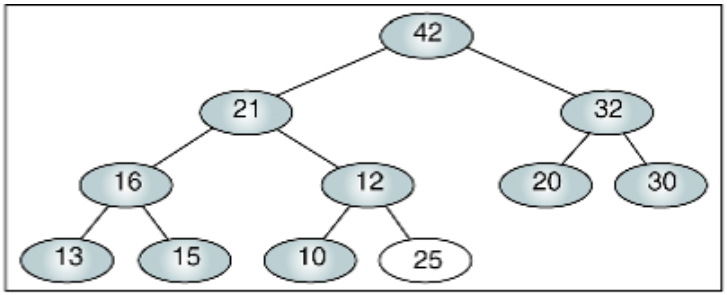


0	1	2	3	4	5	6	7	8

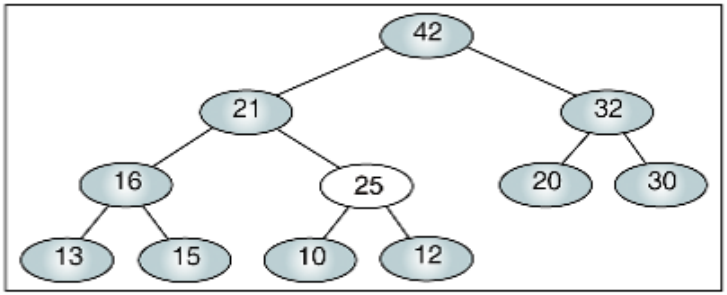
The main operations for heaps are: insert a node and delete a node.

To implement these two operations, we define two algorithms:

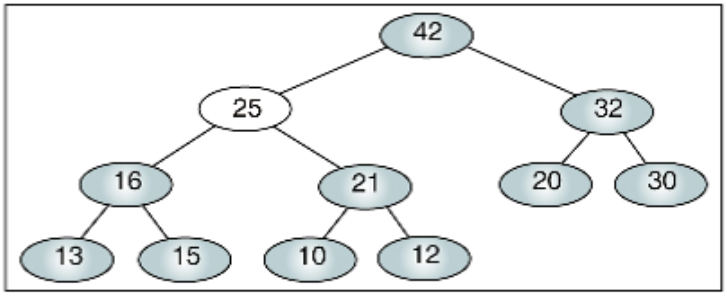
- Reheap up: reorders a "broken" heap (the last element is not smaller than its parent) by floating the last element up the tree until it is in its correct location in the heap.
- Reheap down: reorders a "broken" heap (the root is not greater or equal to its children) by pushing the root down the tree until it is in its correct position in the heap.



(a) Original tree: not a heap

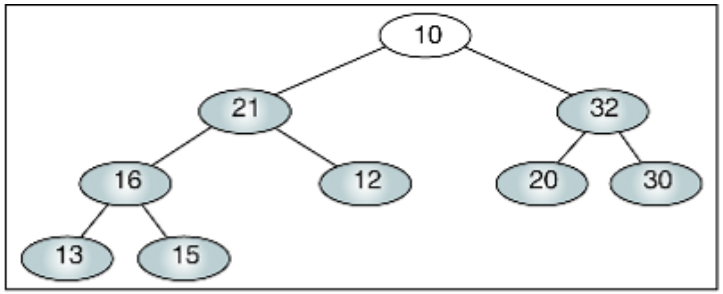


(b) Last element (25) moved up

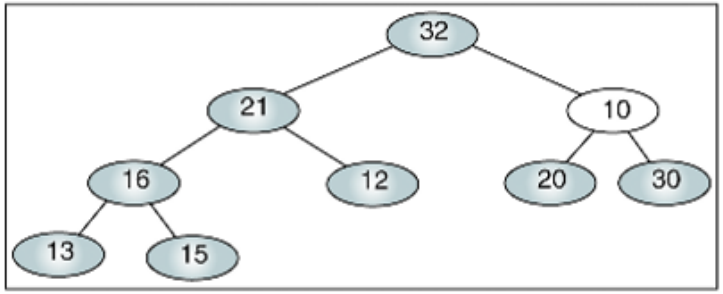


(c) Moved up again: tree is a heap

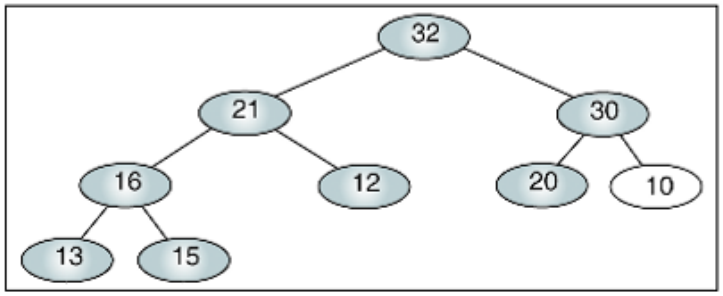
FIGURE 9-5 Reheap Up Example



(a) Original tree: not a heap



(b) Root moved down (right)



(c) Moved down again: tree is a heap

FIGURE 9-7 Reheap Down Example

Main idea for insert and delete:

- To insert a node, first add the node as the last node. Then reheap up.
- To delete a node, first delete the node and put the last node of the heap there. Then reheap down.

```
Algorithm reheapUp (heap, newNode)

  if (newNode not the root)
    set parent to parent of newNode
    if (newNode key > parent key)
      exchange (newNode and parent)
      reheapUp(Heap, parent)
      end if
  end if
```

```
Algorithm reheapDown(heap, root, last)
    if (there is a left subtree)
    set leftKey to left subtree key
    if (there is a right subtree)
       set rightKey to right subtree key
    else
       set rightKey to null
    end if
    if (leftKey > rightKey)
      set largeSubtree to left subtree
    else
      set largeSubtree to right subtree
    end if
    if (root key < largeSubtree key)
      exchange (root and largeSubtree)
      reheadDown (heap, largeSubtree, last)
    end if
  end if
```

Considering a filled array of random elements is given. To build a heap, we need to rearrange the data so that each node in the heap is greater than its children (note that in the array, the children of kth element are $(2k+1)$th and $(2k+2)$th elements). We begin by dividing the array into two parts, the left being a heap and the right being data to be inserted into the heap.

At the beginning, the root (first node) is the only node in the heap and the rest of the array are data to be inserted.

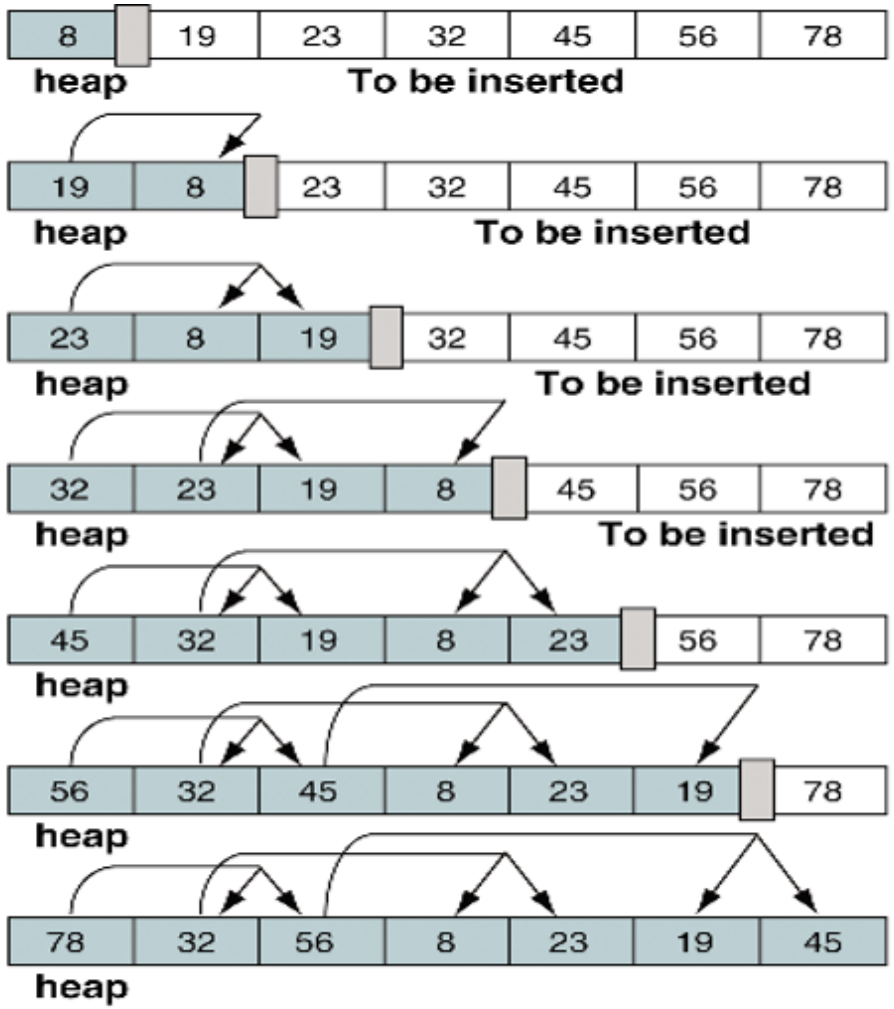


FIGURE 9-9 Building a Heap

```
Algorithm buildHeap (heap, size)
  set walker to 1
  loop (walker < size)
    reheapUp(heap, walker)
    increment walker
  end loop
```

```
Algorithm insertHeap (heap, last, data)
  if (heap full)
    return false
  end if
  increment last
  move data to last node
  reheapUp (heap, last)
  return true
```

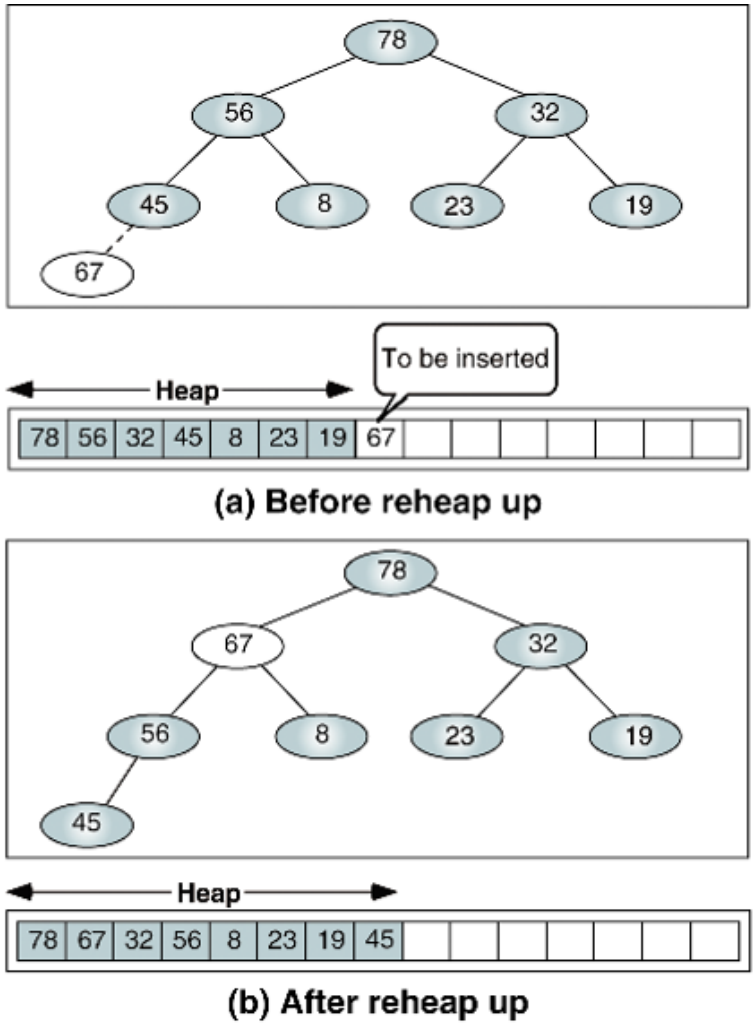


FIGURE 9-10 Insert Node

```
Algorithm deleteHeap (heap, last, dataOut)
  if (heap empty)
     return false
  end if
  set dataOut to root data
  move last data to root
  decrement last
  reheapDown (heap, 0, last )
  return true
```

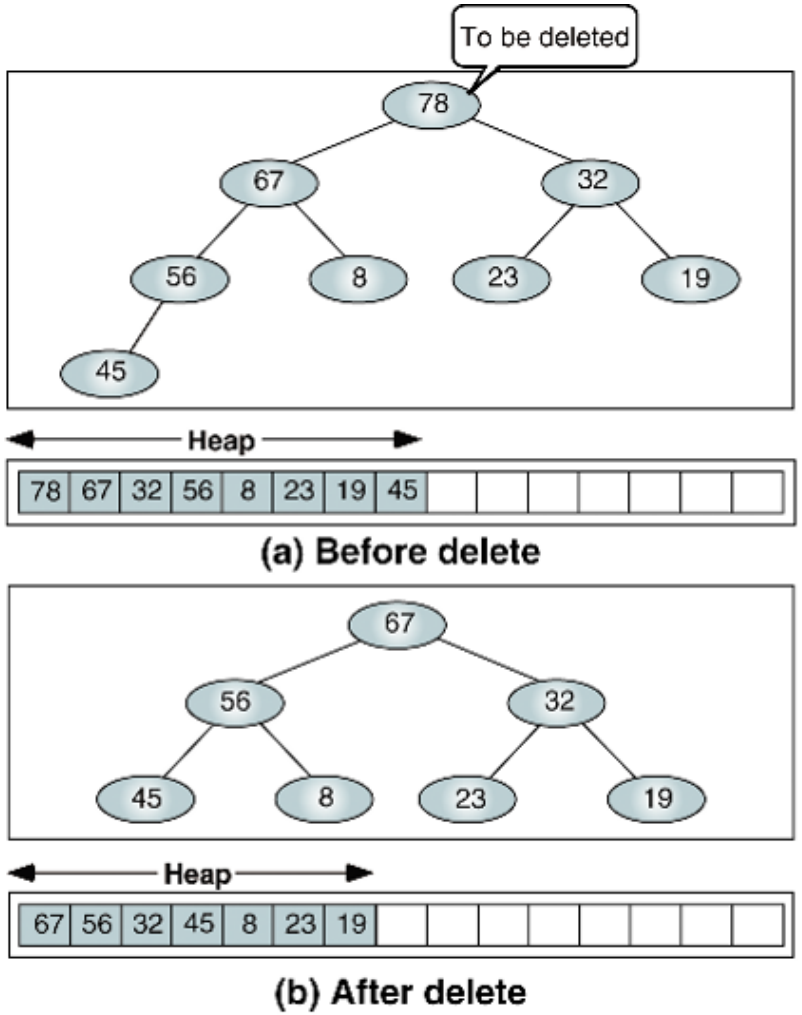


FIGURE 9-11 `deleteHeap` Node

Heap ADT

Use an array to implement a heap in general. Use `void** heapAry` to point to the array.

```
typedef struct
{
   void** heapAry;
   int    last;
   int    size;
   int    (*compare) (void* argu1, void* argu2);
   int    maxSize;
} HEAP;
```

```
HEAP* heapCreate (int maxSize,
                  int  (*compare) (void* argu1, void* argu2))
{
HEAP* heap;
heap = (HEAP*)malloc(sizeof (HEAP));
if (!heap)
   return NULL;
heap->last    = -1;
heap->compare = compare;
// Force heap size to power of 2 -1
heap->maxSize = (int) pow(2, ceil(log(maxSize)/log(2)))-1;
heap->heapAry = (void*)
                  calloc(heap->maxSize, sizeof(void*));
return heap;
} // createHeap
```

```
bool heapInsert (HEAP* heap, void* dataPtr)
{
if (heap->size == 0)                    // Heap empty
   {
    heap->size                 = 1;
    heap->last                 = 0;
    heap->heapAry[heap->last] = dataPtr;
    return true;
   } // if
if (heap->last == heap->maxSize - 1)
  return false;
++(heap->last);
++(heap->size);
heap->heapAry[heap->last] = dataPtr;
_reheapUp (heap, heap->last);
return true;
} // heapInsert
```

```
void  _reheapUp  (HEAP* heap, int childLoc)
{ int    parent;
void** heapAry;
void*  hold;
if (childLoc) // if not at root of heap -- index 0
   {
    heapAry = heap->heapAry;
    parent = (childLoc - 1)/2;
    if (heap->compare(heapAry[childLoc], heapAry[parent]) > 0)
        {
         hold  = heapAry[parent];
         heapAry[parent]  = heapAry[childLoc];
         heapAry[childLoc] = hold;
        _reheapUp (heap, parent);
       } // swap
   }
return;
} // reheapUp
```

```
bool heapDelete (HEAP* heap, void** dataOutPtr)
{
if (heap->size == 0)
    return false;
*dataOutPtr = heap->heapAry[0];
heap->heapAry[0]  = heap->heapAry[heap->last];
(heap->last)--;
(heap->size)--;
_reheapDown (heap, 0);
return true;
} // heapDelete
```

```
void _reheapDown (HEAP* heap, int root)
{
void* hold;
void* leftData;
void* rightData;
int   largeLoc;
int   last;
last = heap->last;
if ((root * 2 + 1) <= last)          // left subtree
   {
    leftData   = heap->heapAry[root * 2 + 1];
    if ((root * 2 + 2) <= last)  // right subtree
       rightData = heap->heapAry[root * 2 + 2];
    else
       rightData = NULL;
    if ((!rightData)
         || heap->compare (leftData, rightData) > 0)
       {
```

```
        largeLoc = root * 2 + 1;
       } // if no right key or leftKey greater
    else
       {
        largeLoc = root * 2 + 2;
       } // else
    if (heap->compare (heap->heapAry[root],
        heap->heapAry[largeLoc]) < 0)
        {
         hold   = heap->heapAry[root];
         heap->heapAry[root] = heap->heapAry[largeLoc];
         heap->heapAry[largeLoc] = hold;
         _reheapDown (heap, largeLoc);
        } // if root <, swap
}
return;
} // reheapDown
```

Applications of heaps:

- Selection algorithm: select the kth element in an unsorted list.
- Priority queues: in a queue, the element with higher priority can rise quicker to advanced position.
- Sorting: sort a random list into an ordered list.

In selection algorithm using heaps:

- input the data of list into a heap.
- remove the root to other place for $k-1$ times.
- process the root, which is the kth element of the list.
- reinsert the removed element to the heap for further usage.

Since we use array implementation for heaps, we can put the removed element to the end of the array.

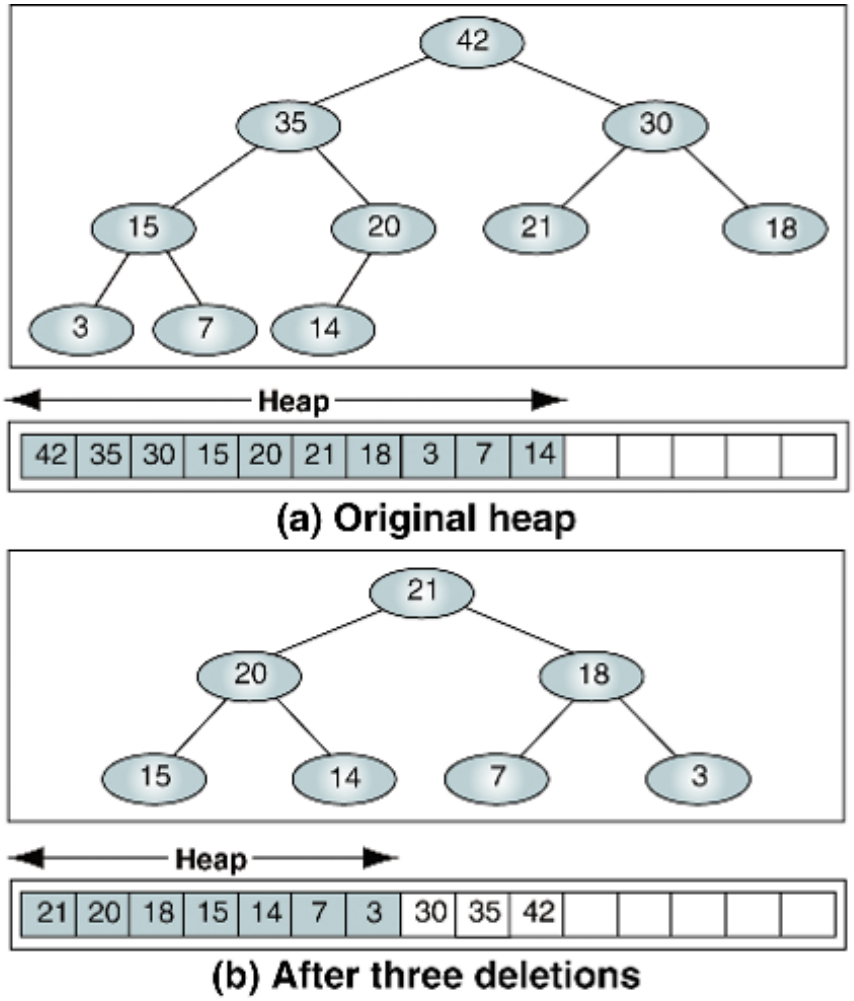


FIGURE 9-14 Heap Selection

```
Algorith selectK (heap, k, heapLast)
  if (k > heap size)
    return false
  end if
  set origHeapSize to heapLast + 1
  loop (k times)
    set tempData to root data
    deleteHeap(heap, heapLast, dataOut)
    move tempData to heapLast + 1
  end loop
  move root data to holdOut
  loop (while heapLast < origHeapSize)
     increment heapLast
     reheapUp (heap, heapLast)
  end loop
  return holdOut
```

To use a heap to construct a priority queue:

- Design key system that reflect both priority and serial of an element. For example, suppose there are 5 priority classes and the maximum number of events in each class is 1000. Then first digit of the key represent the priority and the rest 3 digits are used to represent the serial number. So the keys are $5999, 5998, \ldots, 5000, 4999, \ldots, \ldots, 1000$.
- For each event, a key is first assigned according to its priority and the serial number (as in the above example, the serial number in each class is ordered as $999, 998, \ldots$).
- Then the events are inserted to a heap.
- The events are processed as the deletion of a heap.

The events can be defined as

```
typedef struct
   {
    int  id;
    int  priority;
    int  serial;
   } CUST;

// Prototype Declarations
int   compareCust (void* cust1, void* cust2);
void  processPQ   (HEAP* heap);
char  menu        (void);
CUST* getCust     (void);
```