Heap and Priority Queues

College of Computer Science, CQU

Priority Queues

- There are many situations, where we wish to choose the next "most important" from a collection of people, tasks, or objects.
- When a collection of objects is organized by importance or priority, we call this a **priority queue.**
- A normal queue data structure will not implement a priority queue efficiently because search for the element with highest priority will take $\Theta(n)$ time.

Heaps

- A heap is a data structure that defined by two properties:

1. it is a complete binary tree
   - its height is guaranteed to be the minimum possible. In particular, a heap containing n nodes will have a height of $\log_2 n$ +1

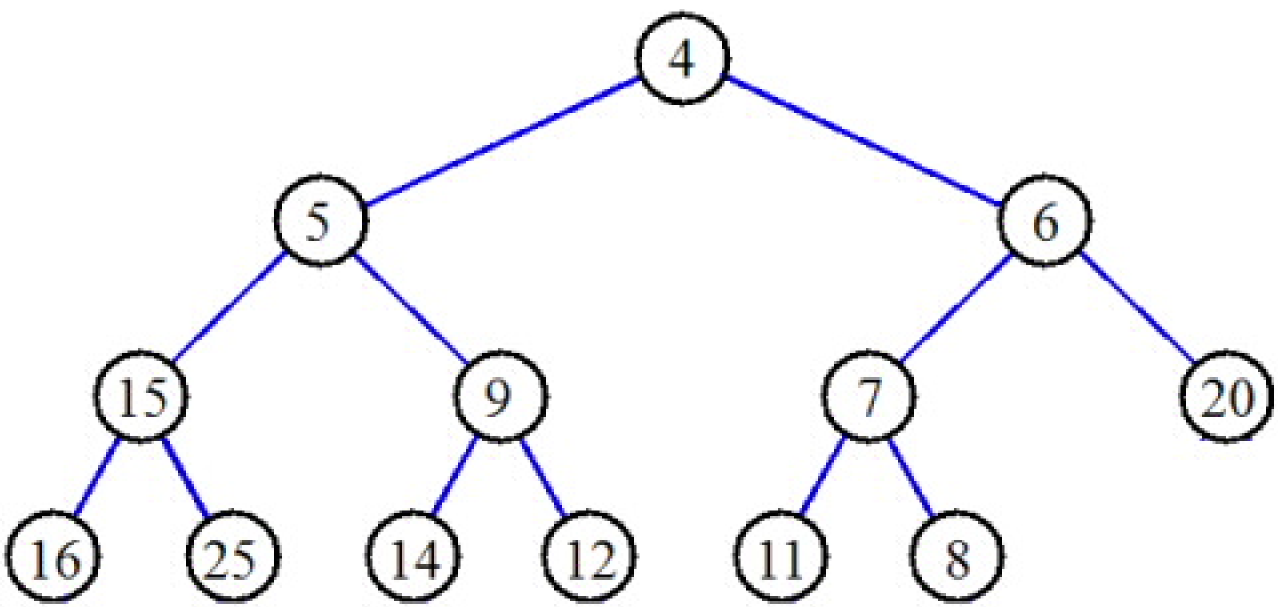
p the values stored in a heap are partially ordered. This means that there is a relationship between the value stored at any node and the values of its children.

- There are two variants of the heap, depending on the definition of this relationship:
  1. MinHeap: key(parent)   key(child)
  2. MaxHeap: key(parent) >= key(child)]

- Note : there is no necessary relationship between the value of a node and that of its sibling in either the min-heap or the max-heap.

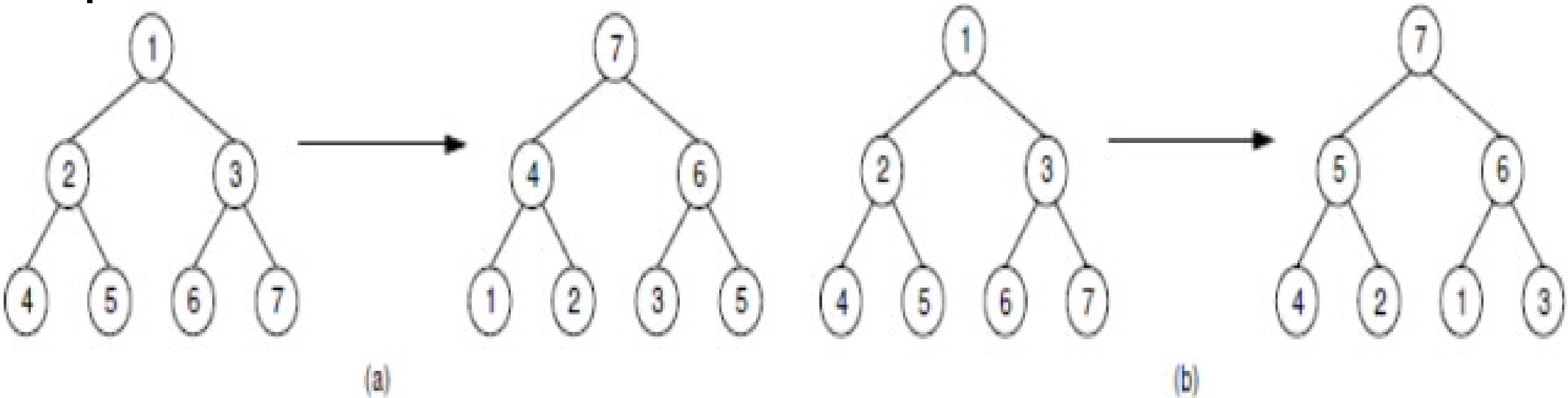
Heap : Example

- **Minheap**



Building a heap (a faster way)

- all n values are available at the beginning of the building process.



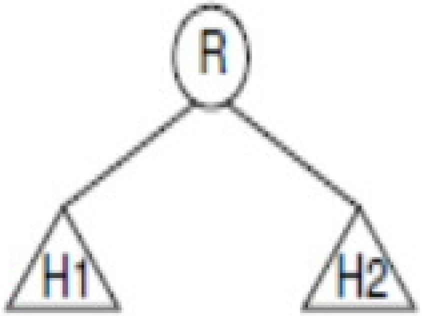
(a) This heap is built by a series of nine exchanges in the order (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6).

(b) This heap is built by a series of four exchanges in the order(5-2), (7-3), (7-1), (6-1).

Same input $\xrightarrow{\text{different arrangement}}$ different heaps

How do we pick the best rearrangement?

A good arrangement algorithm(call siftdown())



Suppose that the left and right subtrees of the root are already heaps, and R is the name of the element at the root.

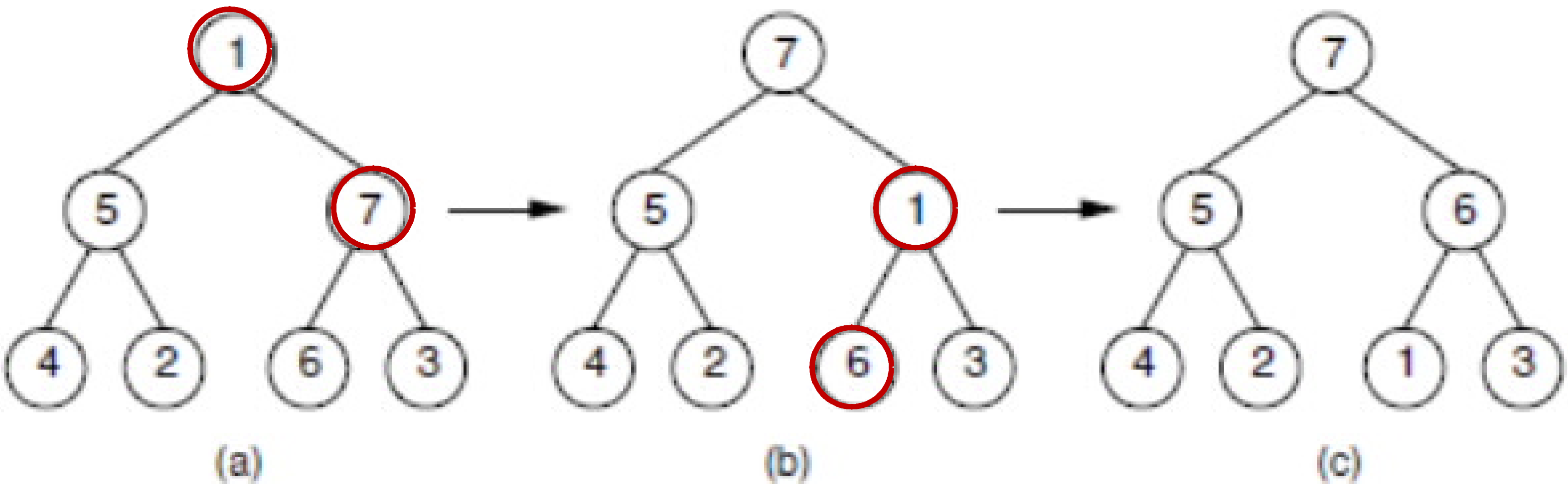
In this case there are two possibilities.

(1) Value(R) $\geqslant$Value(children) :construction is complete.

(2) Value(R) <one or both of Value(children): R should be exchanged with the child that has greater value.

- The result will be a heap, except that R might still be less than one or both of its (new) children.
- In this case, we simply continue the process of "pushing down" R until it reaches a level where it is greater than its children, or is a leaf node. This process is implemented by the private method siftdown(next slide).

Siftdown operation



The subtrees of the root are assumed to be heaps.
(a)The partially completed heap.
(b) Values 1 and 7 are swapped.
(c) Values 1 and 6 are swapped to form the final heap.

siftdown ()

```
// Helper function to put element in its correct place
void siftdown(int pos) {
  while (!isLeaf(pos)) { // Stop if pos is a leaf
    int j = leftchild(pos);  int rc = rightchild(pos);
    if ((rc < n) && Comp::prior(Heap[rc], Heap[j]))
      j = rc;              // Set j to greater child's value
    if (Comp::prior(Heap[pos], Heap[j])) return; // Done
    swap(Heap, pos, j);
    pos = j;               // Move down
  }
}
```

The cost of buildHeap

- **Cost(buildheap)=is the sum of all cost(siftdown)**
- **Each siftdown operation can cost at most the number of levels it takes for the node being sifted to reach the bottom of the tree.**
- **So, this algorithm takes $\Theta(n)$ time in the worst case.**

Heap : Class

```
// Heap class
template <typename E, typename Comp> class heap {
private:
  E* Heap;            // Pointer to the heap array
  int maxsize;           // Maximum size of the heap
  int n;                 // Number of elements now in the heap

  // Helper function to put element in its correct place
  void siftdown(int pos) {
    while (!isLeaf(pos)) { // Stop if pos is a leaf
      int j = leftchild(pos);  int rc = rightchild(pos);
      if ((rc < n) && Comp::prior(Heap[rc], Heap[j]))
        j = rc;              // Set j to greater child's value
      if (Comp::prior(Heap[pos], Heap[j])) return; // Done
      swap(Heap, pos, j);
      pos = j;               // Move down
    }
  }
```

Heap : Class

```
public:
  heap(E* h, int num, int max)       // Constructor
    { Heap = h;  n = num;  maxsize = max;  buildHeap(); }
  int size() const          // Return current heap size
    { return n; }
  bool isLeaf(int pos) const // True if pos is a leaf
    { return (pos >= n/2) && (pos < n); }
  int leftchild(int pos) const
    { return 2*pos + 1; }     // Return leftchild position
  int rightchild(int pos) const
    { return 2*pos + 2; }     // Return rightchild position
  int parent(int pos) const  // Return parent position
    { return (pos-1)/2; }
  void buildHeap()               // Heapify contents of Heap
    { for (int i=n/2-1; i>=0; i--) siftdown(i); }
```

演示

Building a heap (call insert())

- **insert the elements one at a time.**

```
// Insert "it" into the heap
void insert(const E& it) {
  Assert(n < maxsize, "Heap is full");
  int curr = n++;
  Heap[curr] = it;              // Start at end of heap
  // Now sift up until curr's parent > curr
  while ((curr!=0) &&
         (Comp::prior(Heap[curr], Heap[parent(curr)]))) {
    swap(Heap, curr, parent(curr));
    curr = parent(curr);
  }
}
```

- Each call to insert takes $\Theta(\log n)$ time in the worst case, because the value being inserted can move at most the distance from the bottom of the tree to the top of the tree.
- Thus, to insert n values into the heap, if we insert them one at a time, will take $\Theta(n \log n)$ time in the worst case.

Heap removal

- Removing the maximum (root) value from a heap containing n elements requires
  - maintain the complete binary tree shape,
    - by moving the element in the last position in the heap (the current last element in the array) to the root position.
  - the remaining n-1 node values conform to the heap property.
    - If the new root value is not the maximum value in the new heap, use siftdown to reorder the heap.
- the cost of deleting the maximum element is $\Theta(\log n)$ in the average and worst cases, since the heap is log n levels deep,

removefirst()& remove()

```
// Remove first value
E removefirst() {
  Assert (n > 0, "Heap is empty");
  swap(Heap, 0, --n);          // Swap first with last value
  if (n != 0) siftdown(0);  // Siftdown new root val
  return Heap[n];               // Return deleted value
}

// Remove and return element at specified position
E remove(int pos) {
  Assert((pos >= 0) && (pos < n), "Bad position");
  if (pos == (n-1)) n--; // Last element, no work to do
  else
  {
    swap(Heap, pos, --n);          // Swap with last value
    while ((pos != 0) &&
           (Comp::prior(Heap[pos], Heap[parent(pos)]))) {
      swap(Heap, pos, parent(pos)); // Push up large key
      pos = parent(pos);
    }
    if (n != 0) siftdown(pos);     // Push down small key
  }
  return Heap[n];
}
};
```