CS F211: DATA STRUCTURES & ALGORITHMS (2ND SEMESTER 2024-25) PRIORITY QUEUES AND HEAPS

Chittaranjan Hota, PhD
Senior Professor, Computer Sc.
BITS-Pilani Hyderabad Campus
hota[AT]hyderabad.bits-pilani.ac.in

PRIORITY QUEUES: WHAT ARE THESE?

- Arrays, Linked lists: Explicit Linear ordering
- Priority queues: Implicit Linear ordering
- How did you decide BITS campus to choose from several institutions?

Priority may also depend on multiple variables:

- Two values specify a priority: (a, b)
- A pair (a, b) has higher priority than (c, d) if:
  - $a < c$, or
  - $a = c$ and $b < d$

Lexicographic order

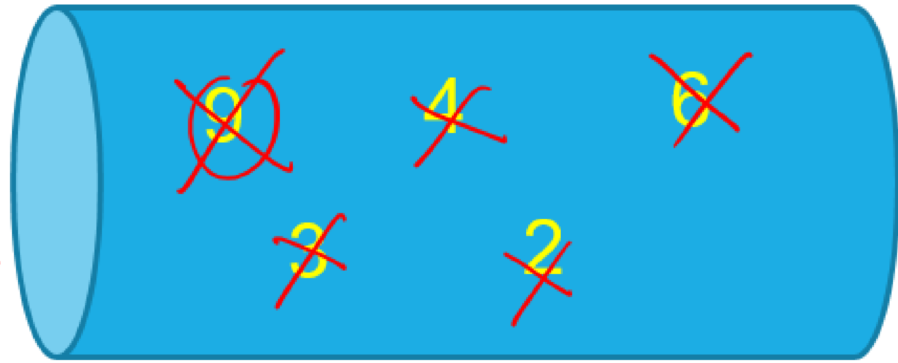
- For example, (5, 19), (13, 1), (13, 24), and (15, 0) all have *higher* priority than (15, 7)

(Prioritizing Medical Attention)

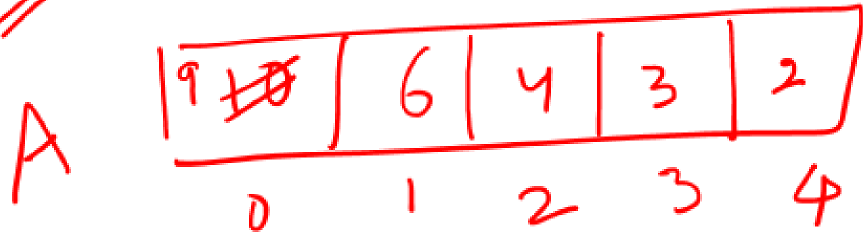
PRIORITY QUEUE ADT: EX. OPERATIONS

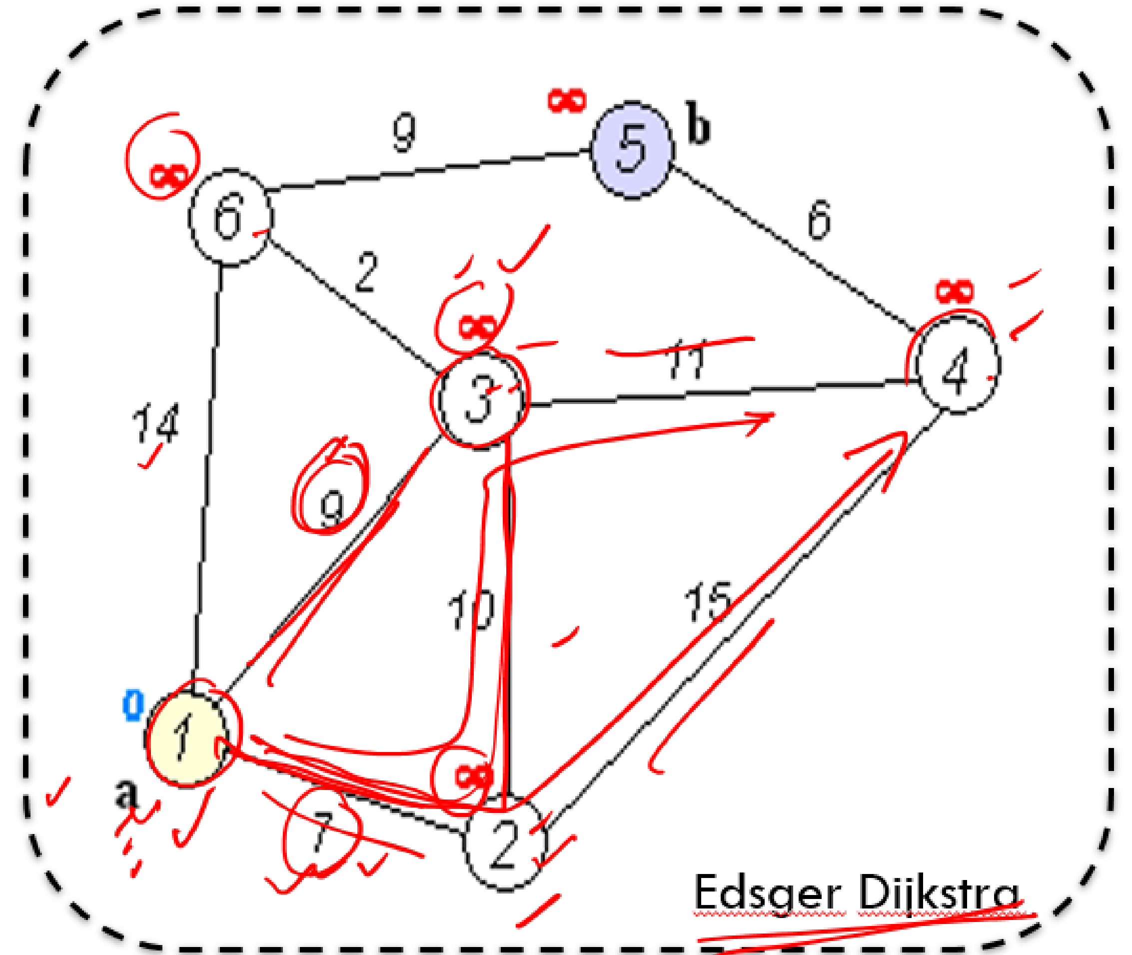
- A priority queue stores a collection of entries.
- Typically, an entry is a pair (key, value), where the key indicates the priority.
- Main methods of the Priority Queue ADT
  - insert(e): inserts an entry e.
  - removeMin(): removes the entry with smallest key.
- Additional methods
  - min(): returns, but does not remove, an entry with smallest key.
  - size(), empty()

APPLICATIONS OF PRIORITY QUEUES



(A max priority queue)





PQ-SORT(S, C) ALGORITHM

Input: S, C for the elements of S

Output: S sorted in increasing order

P ← priority queue with comparator C

while ¬S.empty () {

    e ← S.front();

    S.eraseFront();

    P.insert (e);

}

while ¬P.empty() {

    e ← P.min();

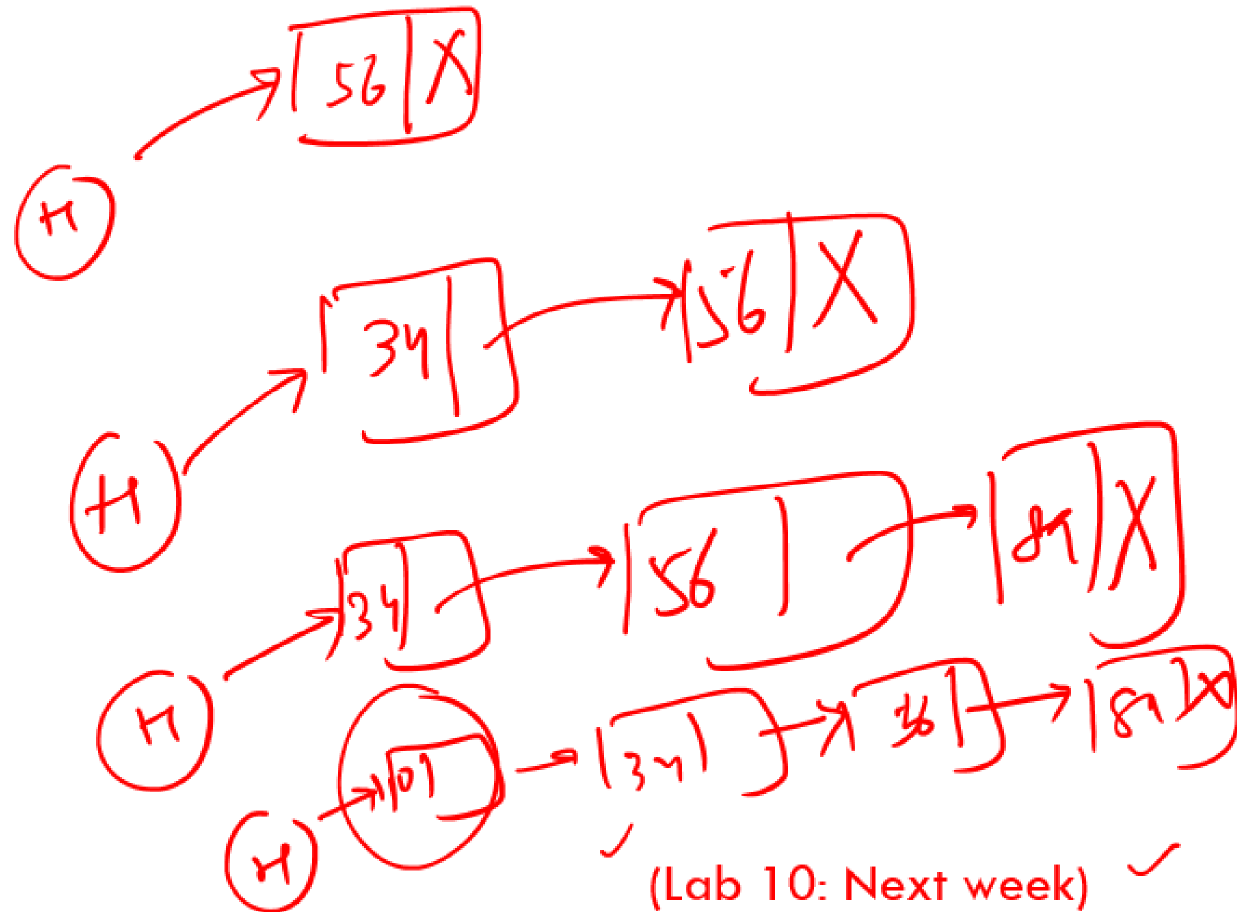
    P.removeMin();

    S.insertBack(e);

}

```
26  template <typename E, typename C>
27  void ListPriorityQueue<E,C>::insert(const E& e) {
28      typename list<E>::iterator p;
29      p = L.begin();
30      while (p != L.end() && !(e <*p)) ++p;
31      L.insert(p, e);
32  }
33
34  template <typename E, typename C>
35  const E& ListPriorityQueue<E,C>::min() const
36      { return L.front(); }
37
38  template <typename E, typename C>
39  void ListPriorityQueue<E,C>::removeMin()
40      { L.pop_front(); }
```

CONTINUED...




```
1 : Insert
2 : Get size
3 : Check if empty
4 : Get minimum element
5 : Remove minimum element
6 : Exit
1
Enter element to be inserted : 56

1
Enter element to be inserted : 34

1
Enter element to be inserted : 89

1
Enter element to be inserted : 10

2
Size is : 4
3
The list is not empty
4
Minimum element : 10
5
Removing minimum element
4
Minimum element : 34
```

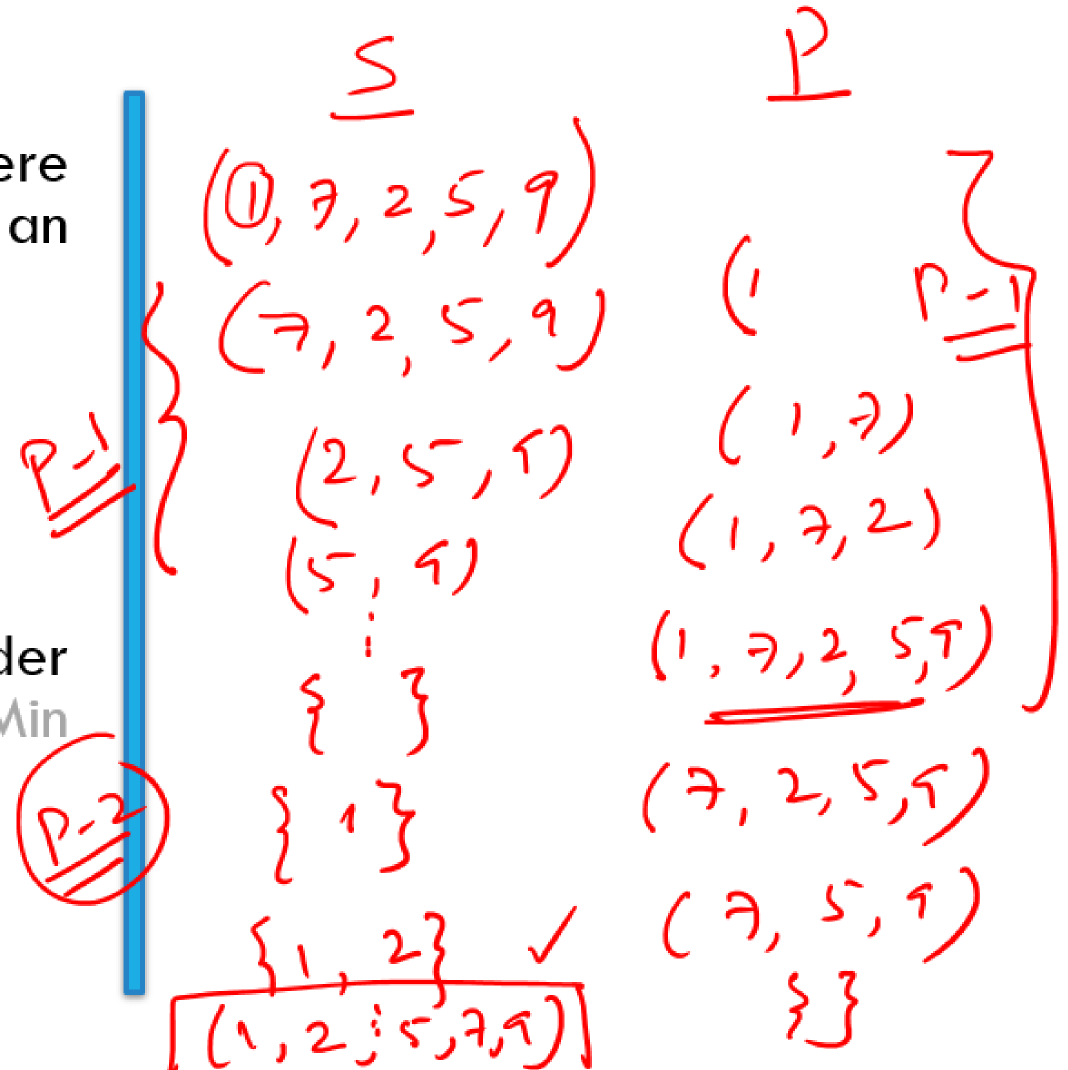
SELECTION SORT EXAMPLE

Selection-sort is the variation of PQ-Sort where the priority queue is implemented with an unsorted sequence.

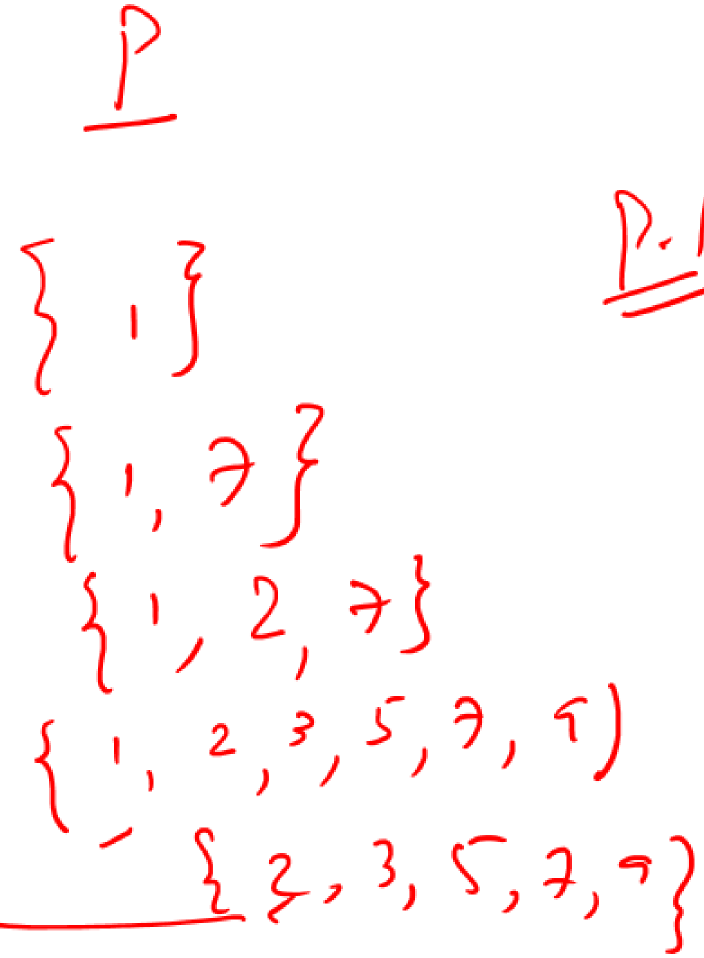
Let us see an example!

Running time of Selection-sort:

1. Phase 1 takes how much time?
2. Removing the elements in sorted order from the priority queue with *n* removeMin operations takes how much time?
3. What is the final time complexity?



INSERTION SORT EXAMPLE



STL PRIORITY QUEUE

```cpp
#include<iostream>
#include <queue>
using namespace std;
void display_priority_queue(priority_queue<int> pq);
int main() {
  priority_queue<int> numbers;

  numbers.push(25);
  numbers.push(50);
  numbers.push(10);
  cout << "Initial Priority Queue: ";
  display_priority_queue(numbers);

  numbers.pop();
  cout << "Final Priority Queue: ";
  display_priority_queue(numbers);

  return 0;
}

void display_priority_queue(priority_queue<int> pq) {
  while(!pq.empty()) {
    cout << pq.top() << ", ";
    pq.pop();
  }

  cout << endl;
}
```

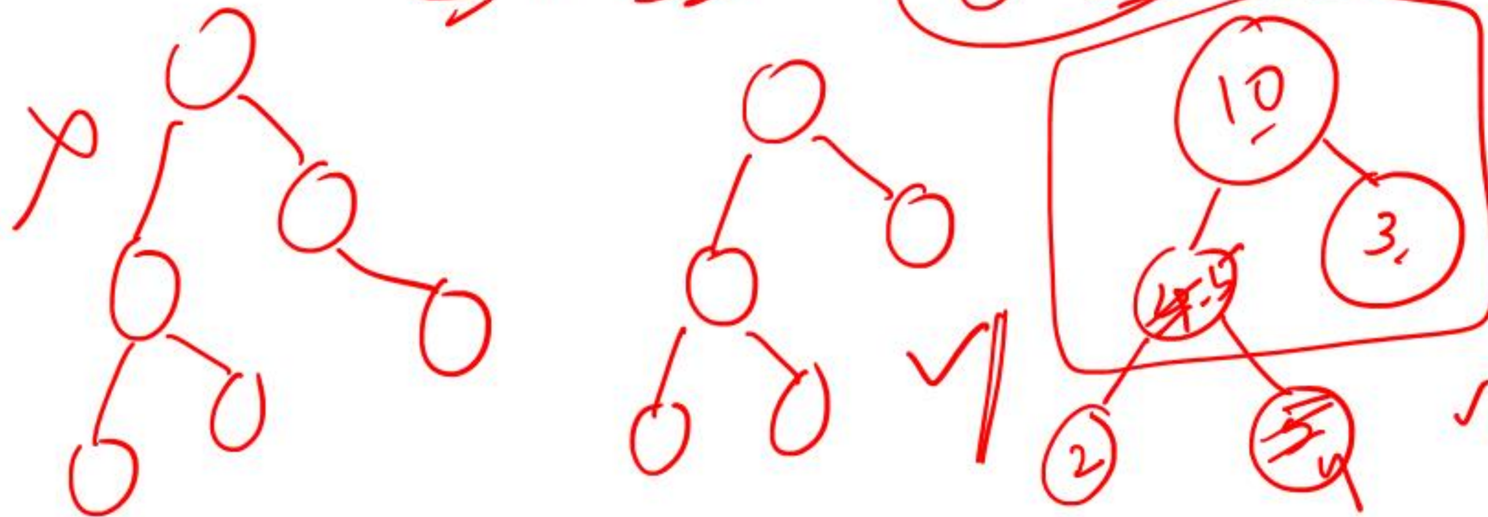
```
Initial Priority Queue: 50, 25, 10,
Final Priority Queue: 25, 10,
```

```cpp
#include <iostream>
#include <queue>
using namespace std;
#define ROW 6
#define COL 3
struct student { //defining the student structure
   int roll,marks;
   student(int roll, int marks)
      : roll(roll), marks(marks)
   {
   }
};
struct comparemarks{ // defining the comparemarks structure
   bool operator()(student const& s1, student const& s2)
   //overloading the operators of the student structure
   {
      return s1.marks < s2.marks;
   }
};
int main()
{
   priority_queue<student, vector<student>, comparemarks> M;
   // using the priority queue.
   int a[ROW][COL] = {{15, 50}, {16, 60},
   {18,70}, {14, 80}, {12, 90}, {20, 100}};
   for (int i = 0; i < ROW; ++i) {
      M.push(student(a[i][0], a[i][1])); //inserting variables in the queue
   }
   cout<<"priority queue for structure ::"<<endl;
   while (!M.empty()) {
      student s = M.top();
      M.pop();
      cout << s.roll << " " << s.marks << "\n"; //printing the values
   }
   return 0;
}
```

input

```
priority queue for structure ::
20 100
12 90
14 80
18 70
16 60
15 50
```

HEAP DATA STRUCTURE



(Heap of Clothes)

(Heap of Books)

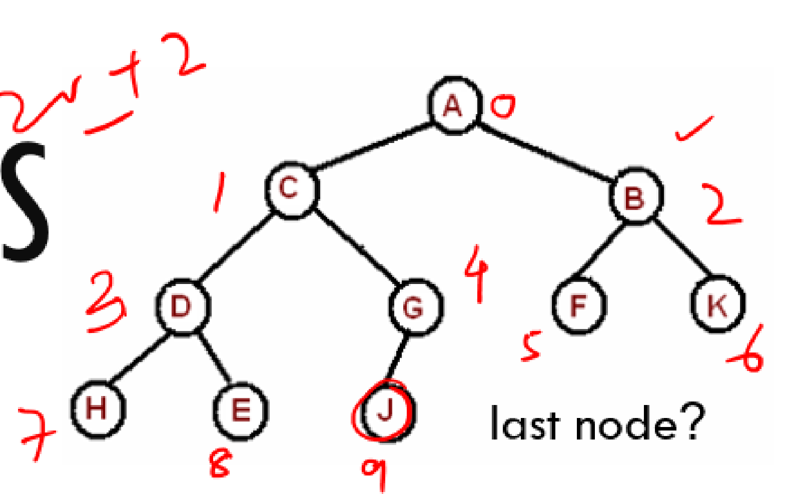
HEIGHT OF A BINARY HEAP

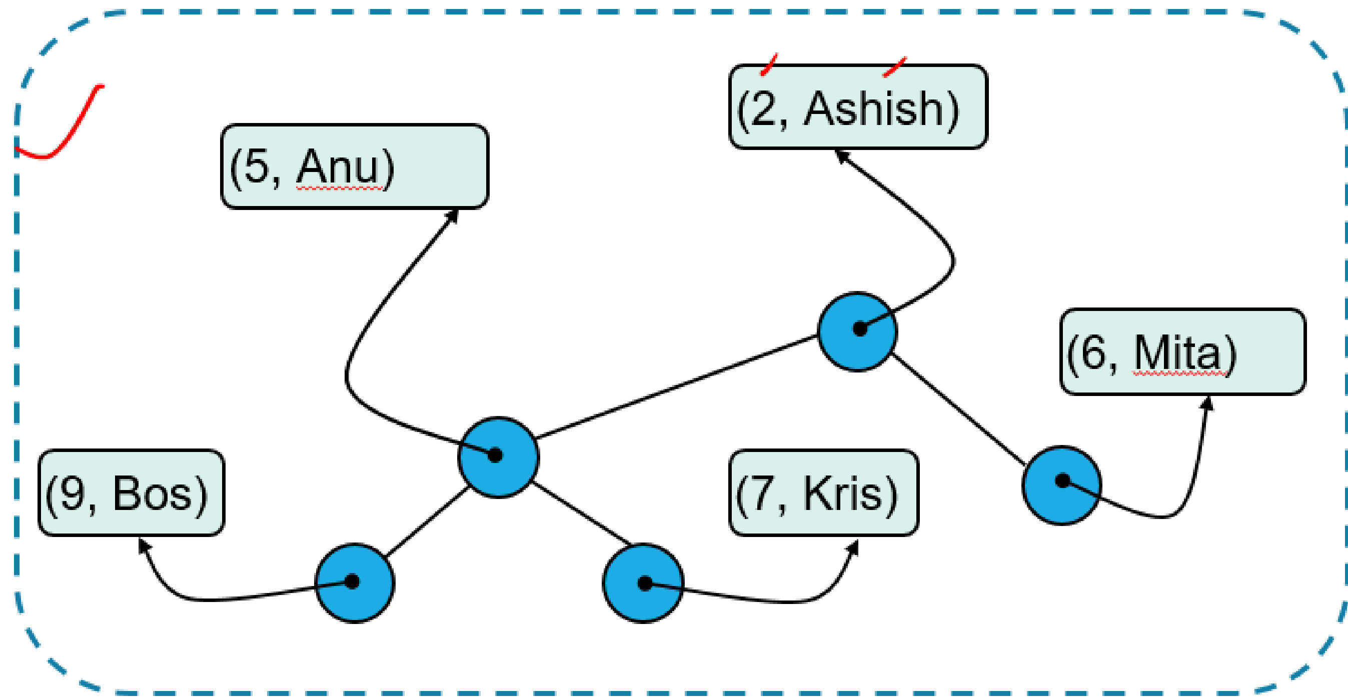
Theorem: A heap storing 'n' keys has height ? $h = \lfloor \log_2 n \rfloor$.

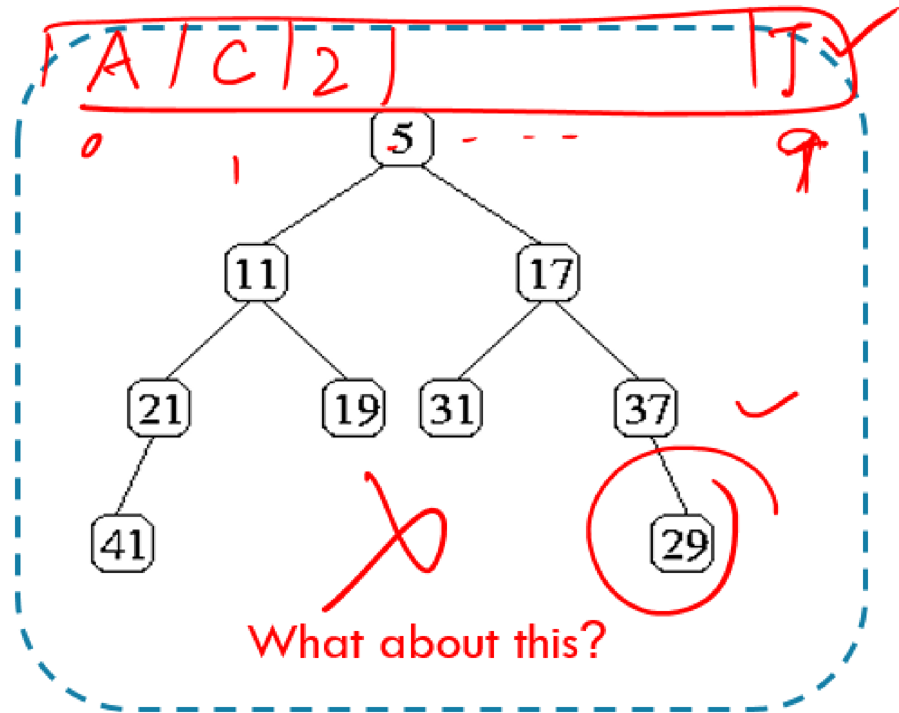
Proof:

HEAPS AS PRIORITY QUEUES

- Why do you need to keep track of the position of the last node?





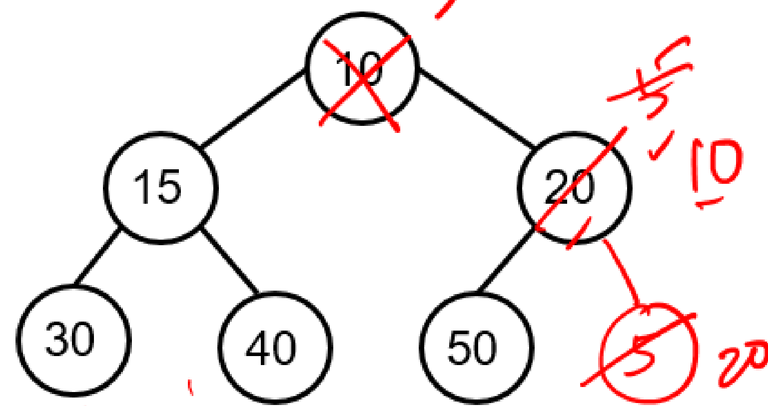


UPHEAP (HEAPIFY-UP OR BUBBLE-UP OR CASCADE-UP)

```
62  template <typename E, typename C>        // insert element
63  void HeapPriorityQueue<E,C>::insert(const E& e) {
64      T.addLast(e);                       // add e to heap
65      Position v = T.last();              // e's position
66      while (!T.isRoot(v)) {              // up-heap bubbling
67          Position u = T.parent(v);
68          if (!(*v < *u)) break;          // if v in order, we're done
69          T.swap(v, u);                   // ...else swap with parent
70          v = u;
71      }
72  }
```

Time Complexity: Ω (?), Θ (?), O(?)

Lab 10 Next week



Insert 5 into this heap!

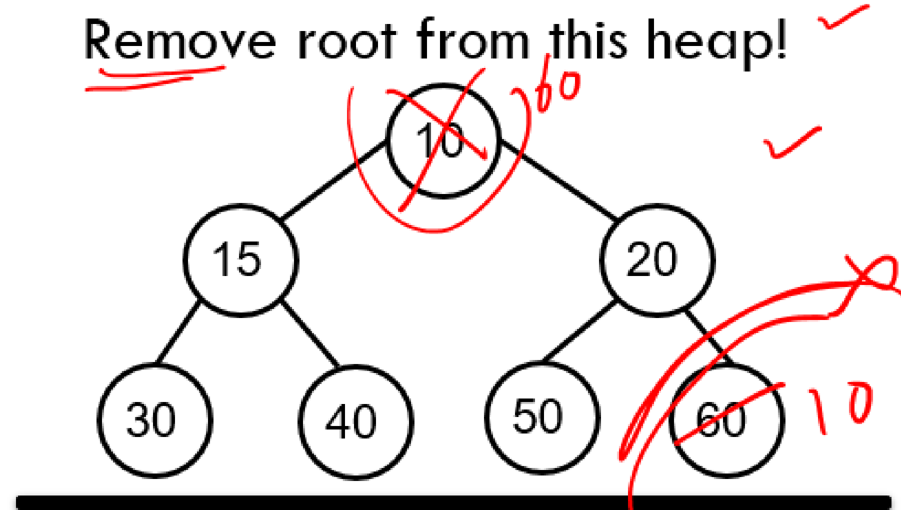
DOWNHEAP (CASCADE-DOWN OR BUBBLE DOWN)

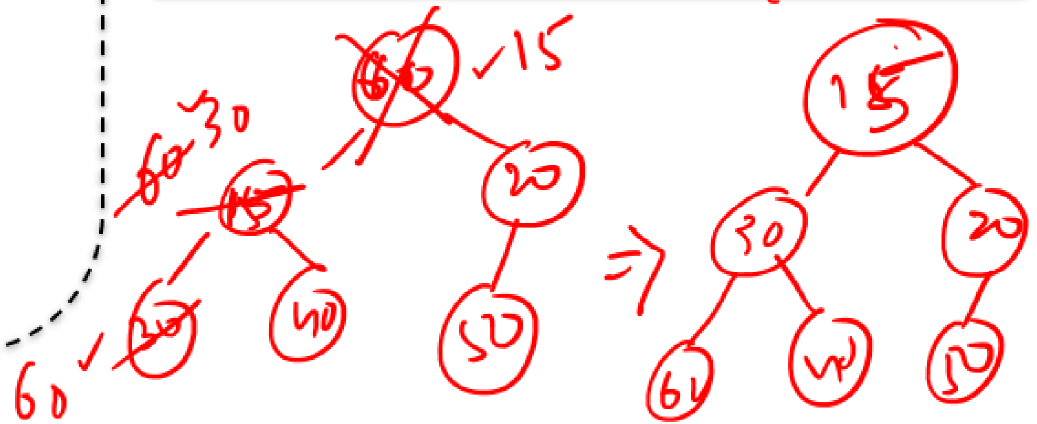
```
74  template <typename E, typename C>        // remove minimum
75  void HeapPriorityQueue<E,C>::removeMin() {
76      if (size() == 1)                      // only one node?
77          T.removeLast();                   // ...remove it
78      else {
79          Position u = T.root();            // root position
80          T.swap(u, T.last());              // swap last with root
81          T.removeLast();                   // ...and remove last
82          while (T.hasLeft(u)) {            // down-heap bubbling
83              Position v = T.left(u);
84              if (T.hasRight(u) && (*(T.right(u)) < *v))
85                  v = T.right(u);           // v is u's smaller child
86              if ((*v < *u)) {              // is u out of order?
87                  T.swap(u, v);             // ...then swap
88                  u = v;
89              }
90              else break;                   // else we're done
91          }
92      }
93  }
```

Lab 10 Next week

Worst case Time Complexity: O(?)

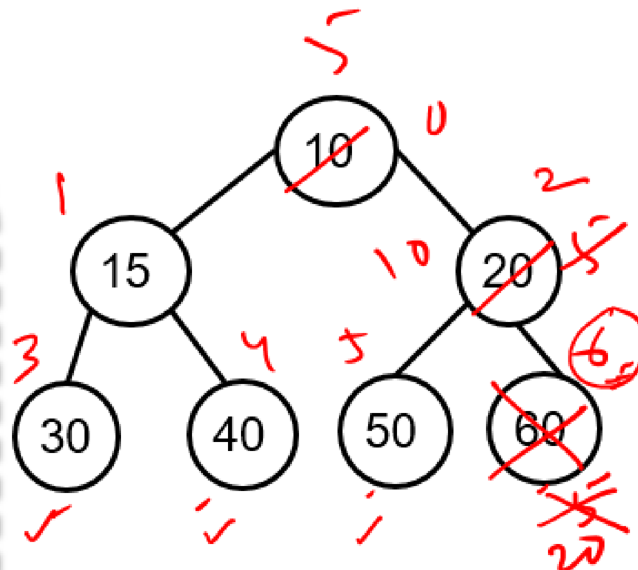
Remove root from this heap!



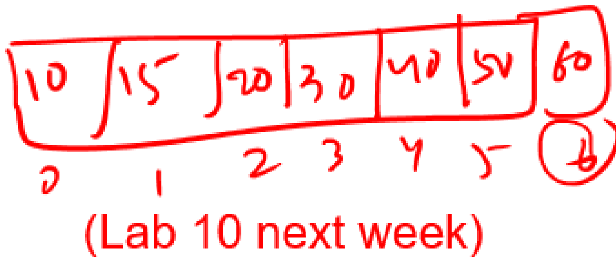


UPDATING LAST NODE IN A HEAP

```
function updateLastNode(heap, newValue){
    n = heap.size;
    lastIndex = n – 1;
    heap[lastIndex] = newValue;
    parentIndex = (lastIndex - 1) / 2;
    if heap[lastIndex] < heap[parentIndex]
        heapifyUp(heap, lastIndex);
    else
        heapifyDown(heap, lastIndex);
}
```



(updating the last node with 5)

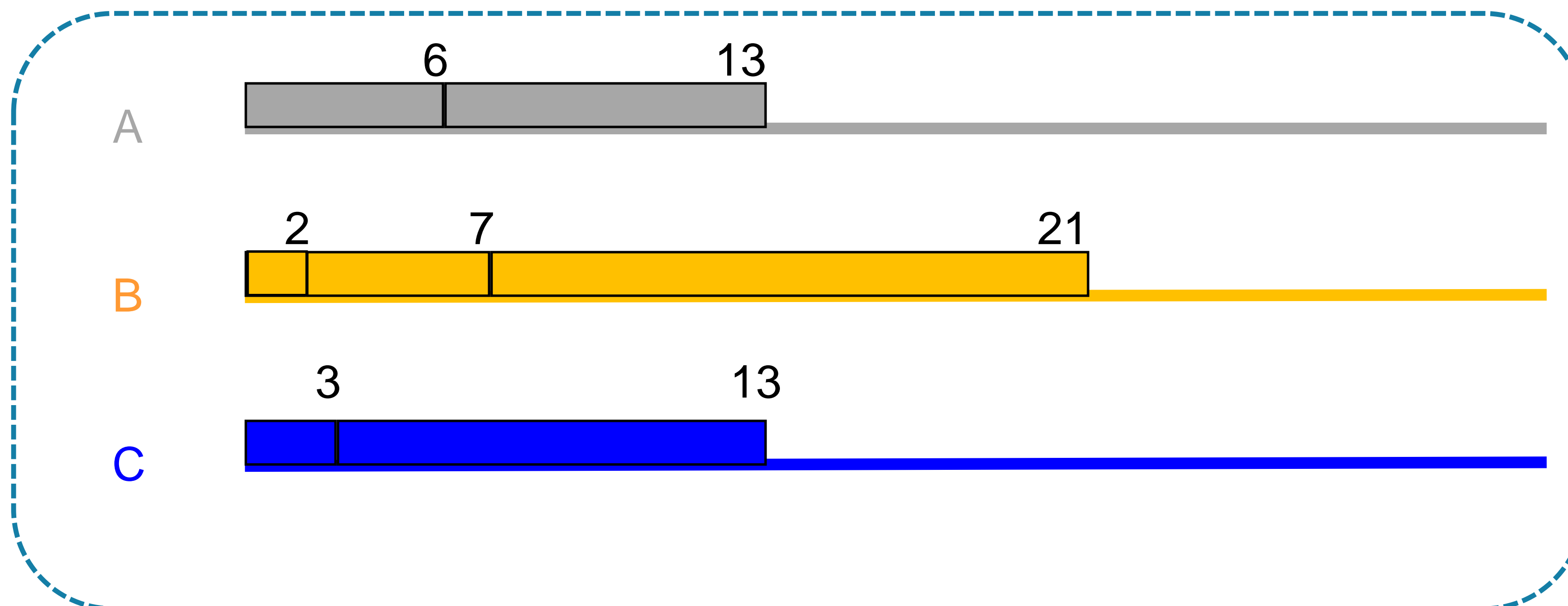


(Lab 10 next week)

```
1 : Get size
2 : Check if empty
3 : Insert
4 : Get minimum element
5 : Remove minimum element
6 : Exit
3
Enter element to be inserted : 10

3
Enter element to be inserted : 30

3
Enter element to be inserted : 60

3
Enter element to be inserted : 5

3
Enter element to be inserted : 20

1
Size : 5
2
Heap is not empty
4
Minimum element : 5
5
Removing minimum element
4
Minimum element : 10
```

MACHINE SCHEDULING: APPLICATION OF HEAPS

- 3 machines and 7 jobs
- Task/ job times are [6, 2, 3, 5, 10, 7, 14]
- Possible schedule



What is the finish time?

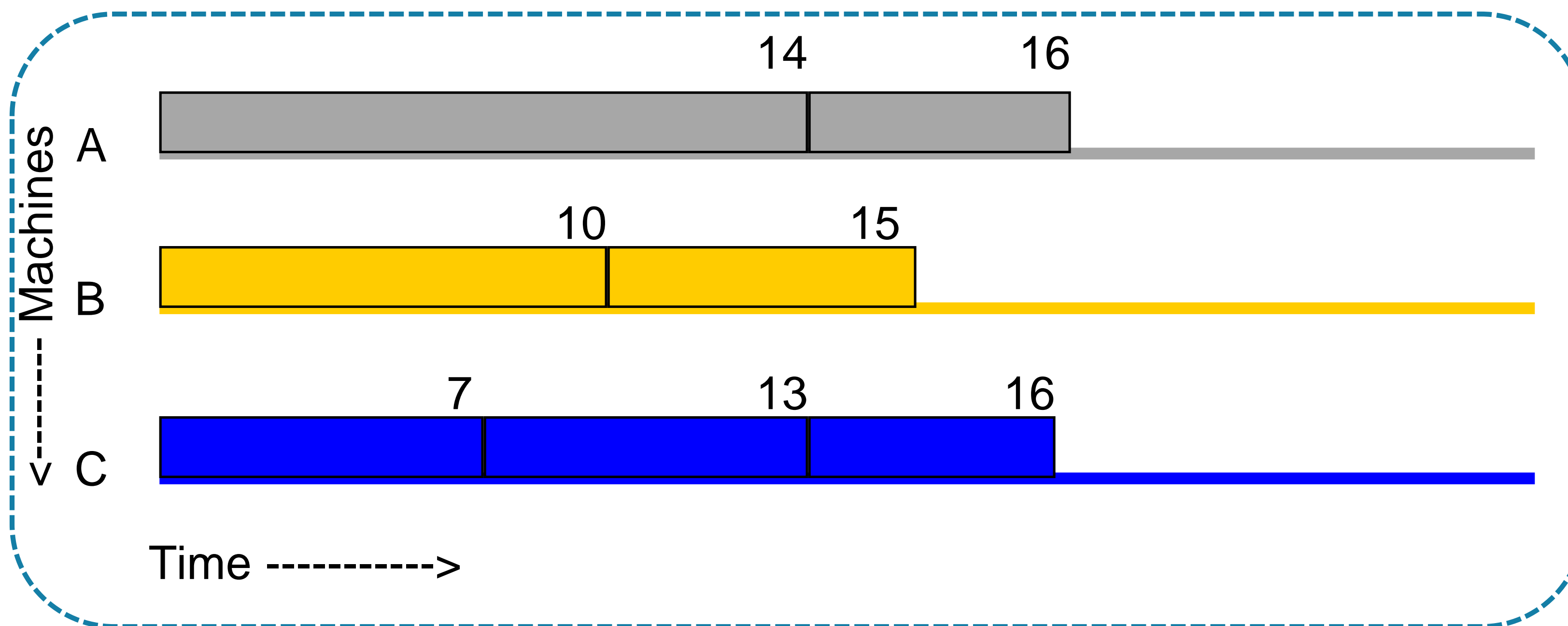
Are there any better ways of doing this?

LONGEST PROCESSING TIME FIRST

Jobs are scheduled in the order

- 14, 10, 7, 6, 5, 3, 2

Each job is scheduled on the machine on which it finishes earliest.



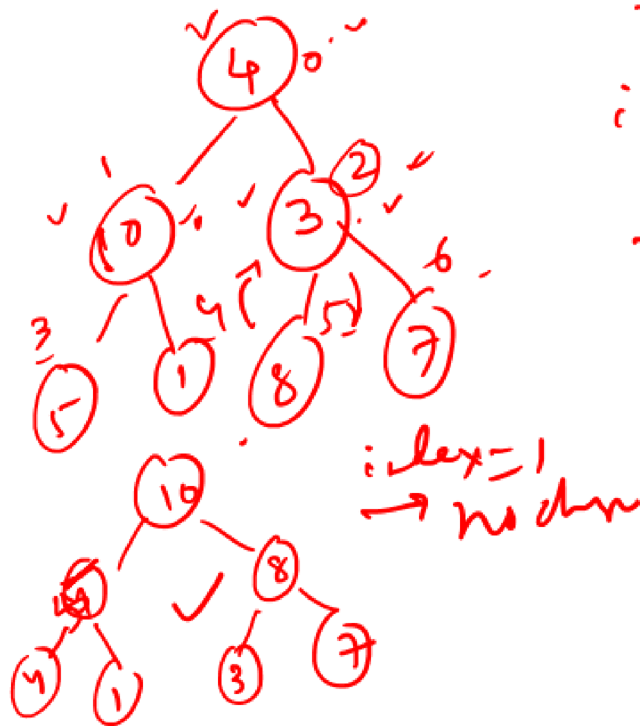
What is the finish time?

Lab 10 next week

BUILDING A MAX-HEAP

Example: Build a Max-Heap from the array:

4	10	3	5	1	8	7



```
void buildMaxHeap(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
}
```

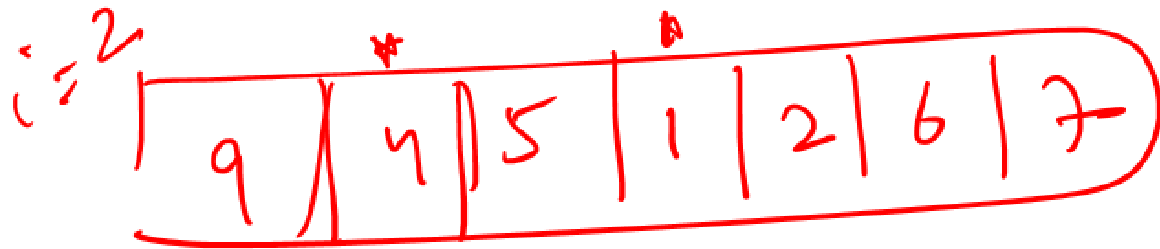
Lab 10 next week

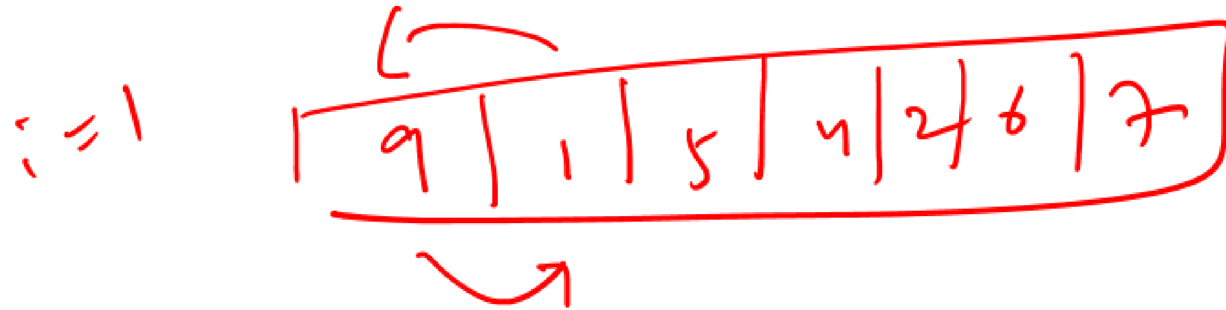
```
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    // Check if left child exists and is > root
    if (left < n && arr[left] > arr[largest])
        largest = left;
    // Check if right child exists and is > largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;
    // If largest is not root, swap and heapify the affected
    subtrees recursively
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
```

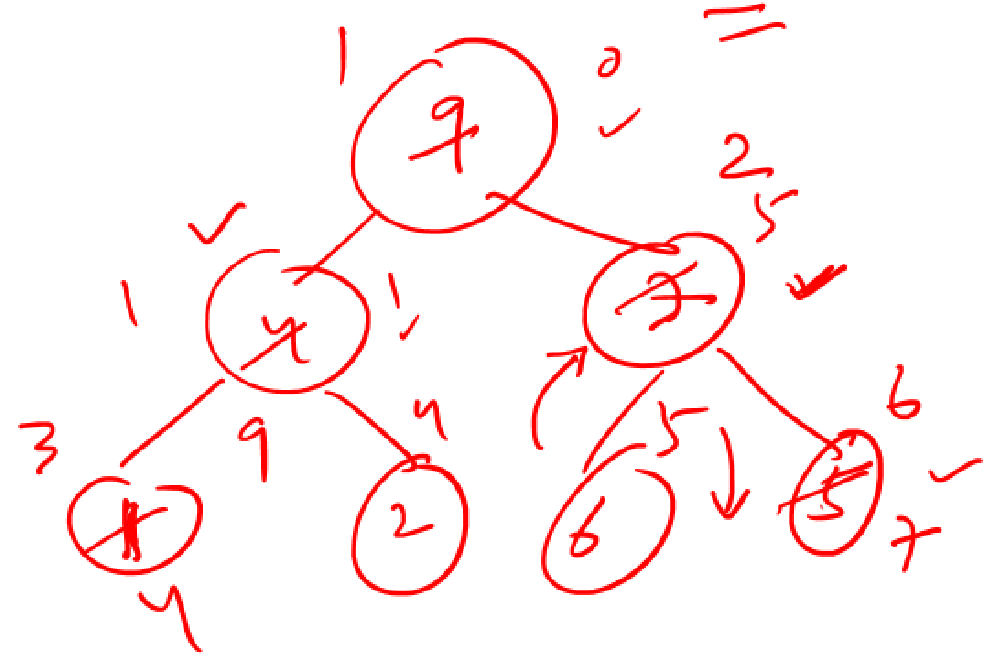
ANOTHER EXAMPLE: BUILDING MIN-HEAP

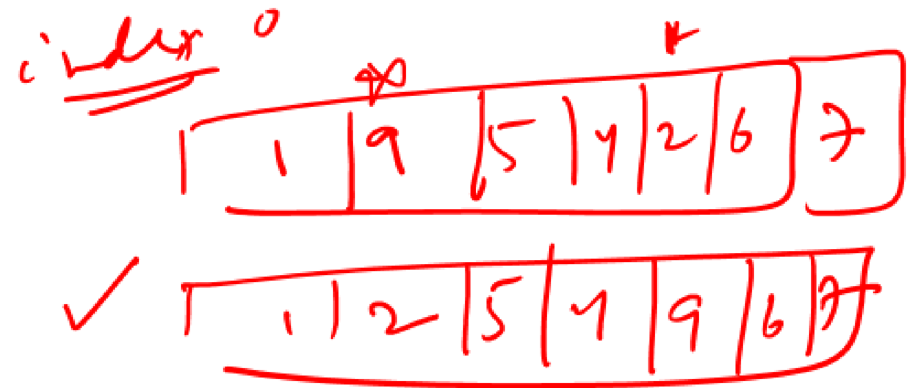
Example: Build a Min-Heap from the array:

9	4	7	1	2	6	5









BUILDING HEAPS USING C++ STL

```
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
    std::vector<int> vec = {3, 1, 4, 1, 5, 9, 2};
    std::make_heap(vec.begin(), vec.end());
    vec.push_back(8);
    std::push_heap(vec.begin(), vec.end());
    std::cout << "Max Heap after pushing 8:";
    for (int i : vec) {
        std::cout << i << " ";
    }
std::pop_heap(vec.begin(), vec.end());
    vec.pop_back();
    std::cout << "Max Heap after
            popping the largest element: ";
    for (int i : vec) {
        std::cout << i << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

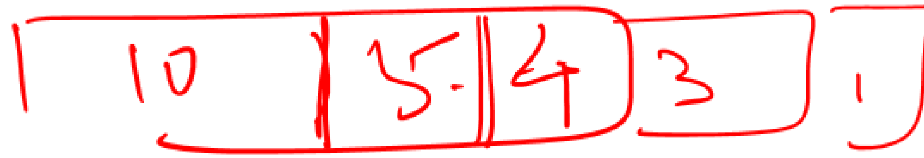
Min Heap: `make_heap(minHeap.begin(), minHeap.end(), greater<int>());`

OTHER STL IN C++

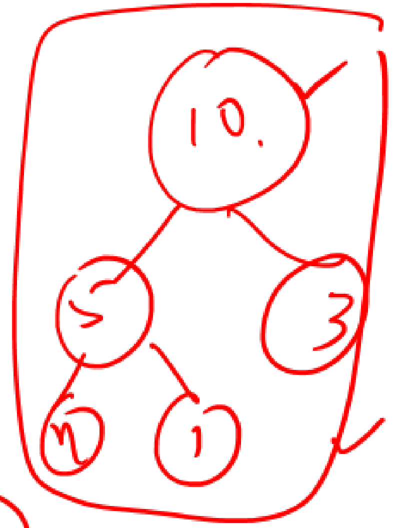
Feature	priority_queue(STL)	make_heap(STL)
Implementation	Uses a binary heap internally	Converts a vector into heap structure
Insertion	push() adds an element efficiently	No direct insertion: use push_back() followed by push_heap()
Deletion	pop()	pop_heap() followed by pop_heap()
Default type	Max Heap	Max Heap
When to use?	Frequent insertion and removal	One time heap structure

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;
int main() {
    // Example using priority_queue (Min Heap)
    priority_queue<int, vector<int>, greater<int>> pq;
    pq.push(10);
    pq.push(4);
    pq.push(15);
    pq.push(20);
    pq.push(1);
    cout << "Priority Queue (Min Heap) Output: ";
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
    cout << endl;
    // Example using make_heap (Min Heap)
    vector<int> minHeap = {10, 4, 15, 20, 1};
    make_heap(minHeap.begin(), minHeap.end(), greater<int>());

    cout << "make_heap (Min Heap) Output: ";
    while (!minHeap.empty()) {
        pop_heap(minHeap.begin(), minHeap.end(), greater<int>());
        cout << minHeap.back() << " ";
        minHeap.pop_back();
    }
    cout << endl;

    return 0;
}
```

```
Priority Queue (Min Heap) Output: 1 4 10 15 20
make_heap (Min Heap) Output: 1 4 10 15 20
```

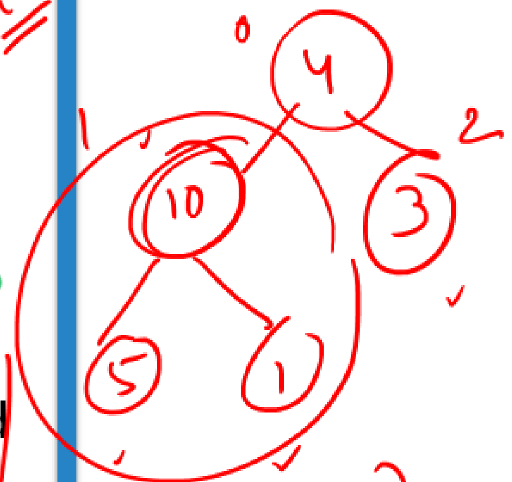


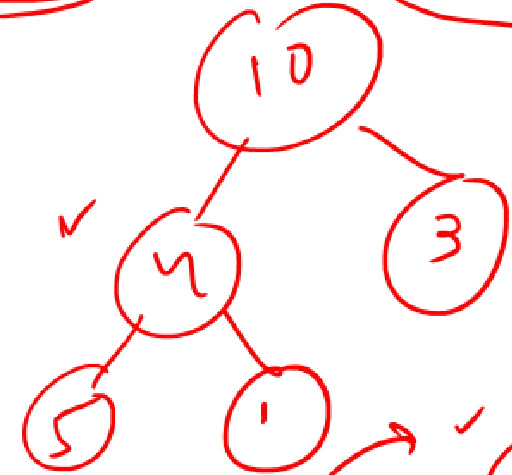
HEAP SORT: APPLICATION OF HEAP

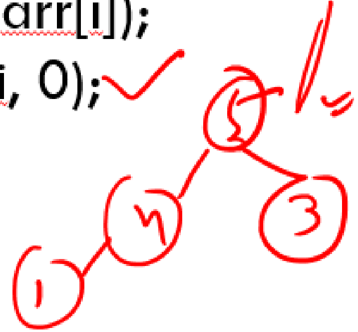


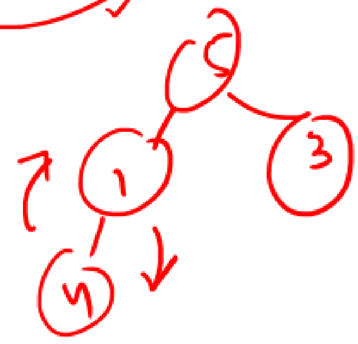

```
void heapSort( vector <int>& Arr) {
  int n = arr.size();
  // Build a max-heap
  for (int i = n/2 - 1; i >= 0; i--)
    heapify(arr, n, i);
  // Extract elements from the heap
  for (int i = n - 1; i > 0; i--) {
    // Move current root to the end
    swap(arr[0], arr[i]);
    heapify(arr, i, 0);
  }
}
```

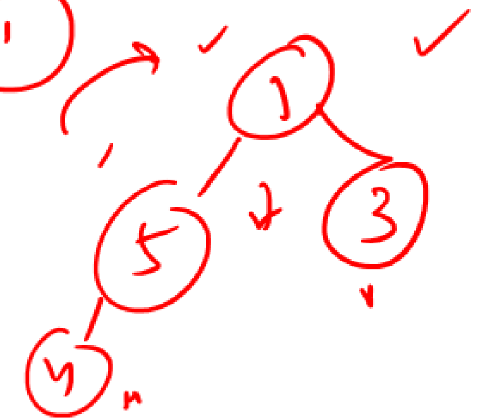
Input Array: [4, 10, 3, 5, 1]











COMPLEXITY OF HEAP SORT

- Building the Heap: O(n)

Extracting elements: O(nlogn)

Extracting root: O(1), Heapify the remaining heap: logn

Repeated n times: nlogn

max (n, nlogn) = O(nlogn)

Work done at each level:

- The heapify operation at a node at level i : O(h-i), where h is the height of heap.

(because the node might need to swap its way down to the bottom of the heap)

- Total work done: $\sum_{i=0}^{h} (\text{Number of Nodes at level i}) \times \mathrm{O}(\mathrm{h}-\mathrm{i})$

$$= \sum_{i=0}^{h} 2^i \times \mathrm{O}(\mathrm{h}-\mathrm{i}) = O(\sum_{i=0}^{h} 2^i(\mathrm{h}-\mathrm{i})) = O(\sum_{\mathrm{k}=0}^{h} 2^{\mathrm{h}-\mathrm{k}}.\mathrm{k})$$

As, $2^{h-k} = 2^h / 2^k$ and $2^h <= n$, the sum can be rewritten as: $O(n.\sum_{\mathrm{k}=0}^{h} \frac{k}{2^k})$

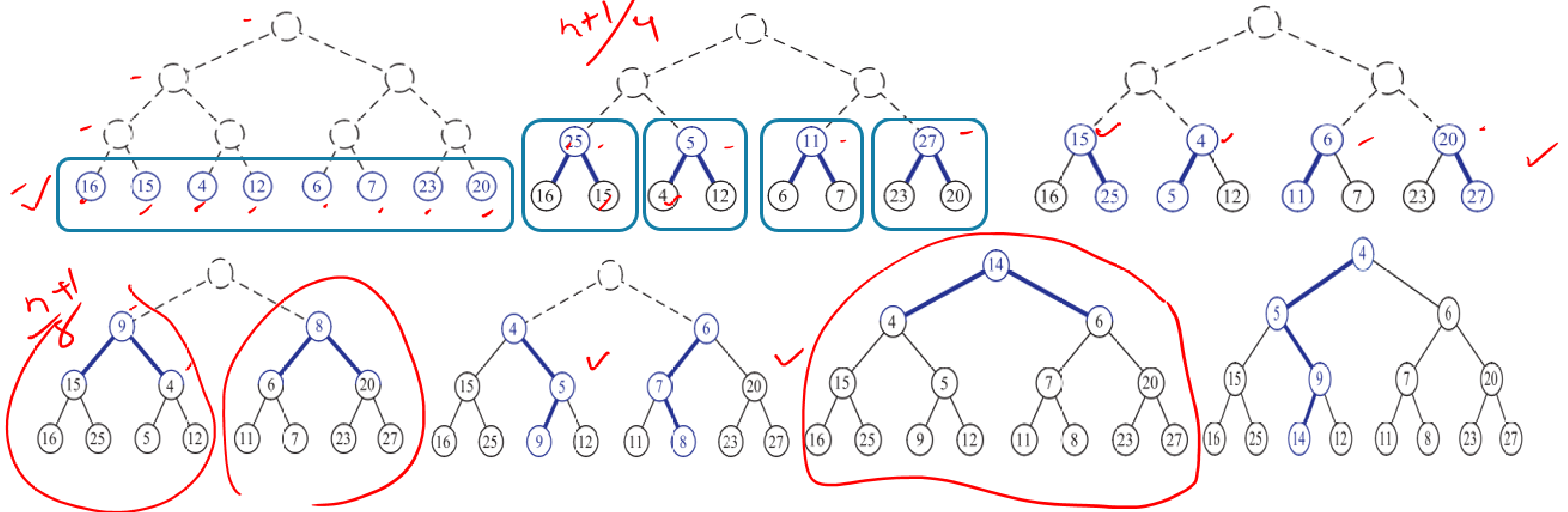
Constant 2

O(n)

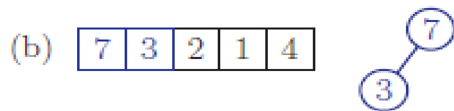
BOTTOM-UP HEAP CONSTRUCTION

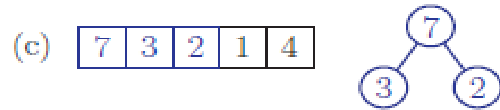
If all the elements in the heap are given in advance, in place of O(nlogn), one can improve on the time by building the heap in a bottom-up fashion rather than inserting and then heapifying. It will be O(n).

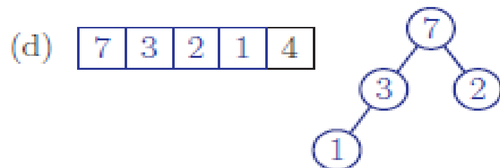
In the first step, we construct (n+1)/2 elementary heaps storing one entry each.

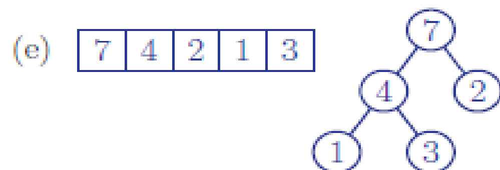


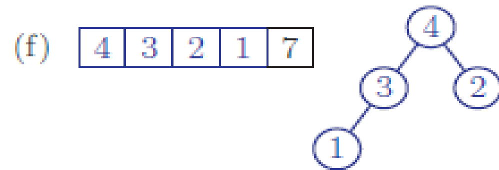
IN-PLACE HEAP SORT

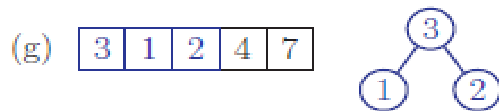


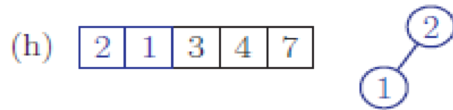






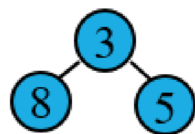


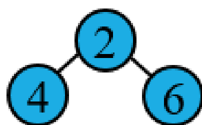




```
function heapSort(arr) {
    n = length(arr)
    for (i = n/2 – 1, i>=0; i--)           } Build a Max-heap
        heapify(arr, n, i);
    for (i = n – 1, i>=0, i--)             } Extract elements
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
}

function heapify(arr, n, i) {
    largest = i ;
    left = 2 * i + 1;
    right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;
    if largest != i:
        swap(arr[i], arr[largest]);
        # Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
}
```

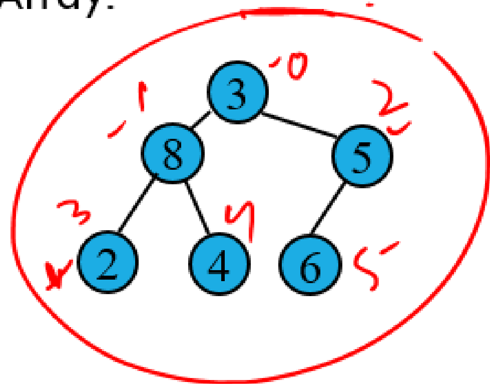
MERGING TWO HEAPS

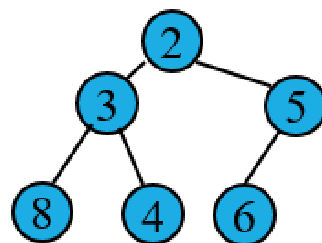




```
vector<int> mergeHeaps(vector<int>& heap1, vector<int>& heap2) {
    vector<int> mergedHeap = heap1;
    mergedHeap.insert(mergedHeap.end(), heap2.begin(), heap2.end());

    // Build a max heap from the merged array
    int n = mergedHeap.size();
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(mergedHeap, n, i);

    return mergedHeap;
}
```

Merged Array: [3, 8, 5, 2, 4, 6]





Next Class: Maps, Dictionaries, Hash tables…