CS 315 April 1

Goals:

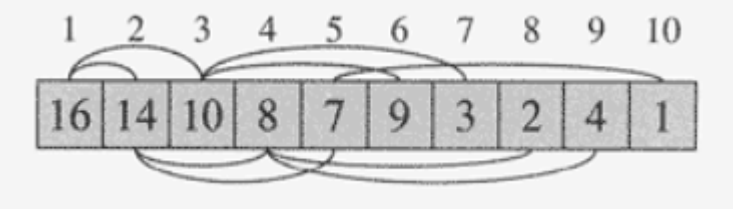
Heap (Chapter 6) continued

- review of Algorithms for
  - *Insert*
  - *DeleteMin*
  - *percolate-down*

- algoritms for
  - *decraseKey*
  - *increaseKey*
  - *Build-heap*

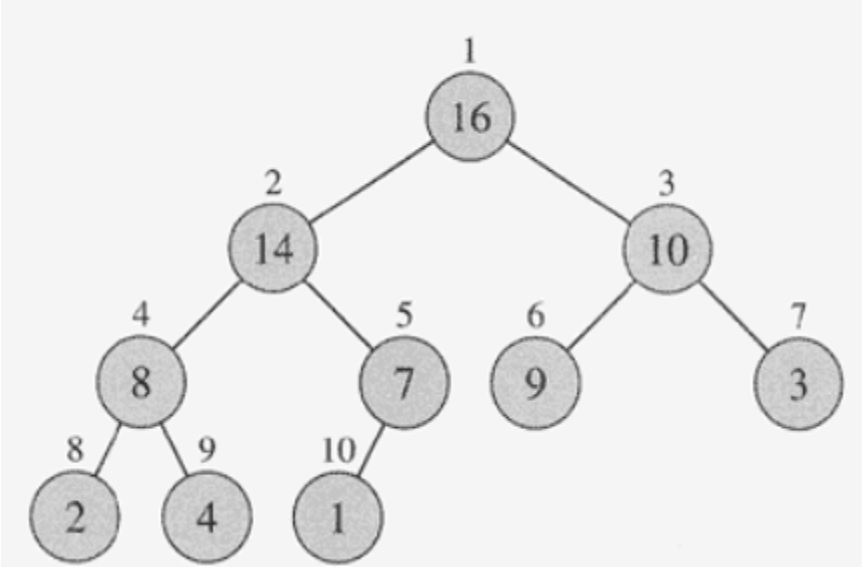
- heap-sorting, machine scheduling application

Binary Heap as an array and as a tree



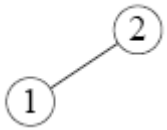
Array of 10 keys stored in A[1] through A[10]

Can be viewed as a tree. Index j has index 2j and 2j+1 as children.

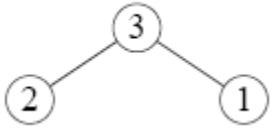


Connection between size and height of a heap

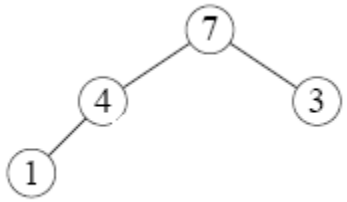
h = 1



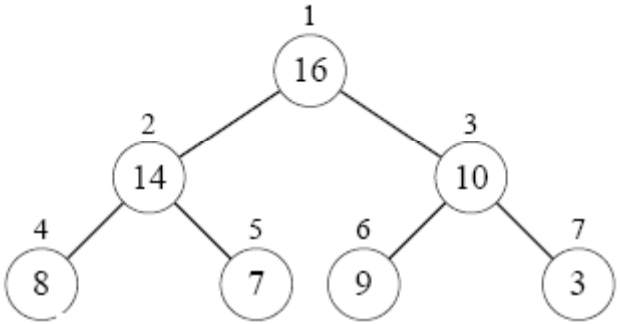
h = 2



h = 2



h = 3

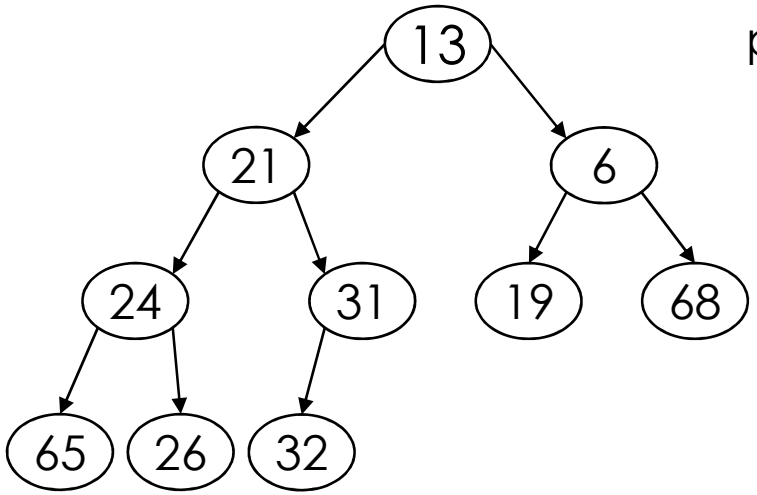


h = 3

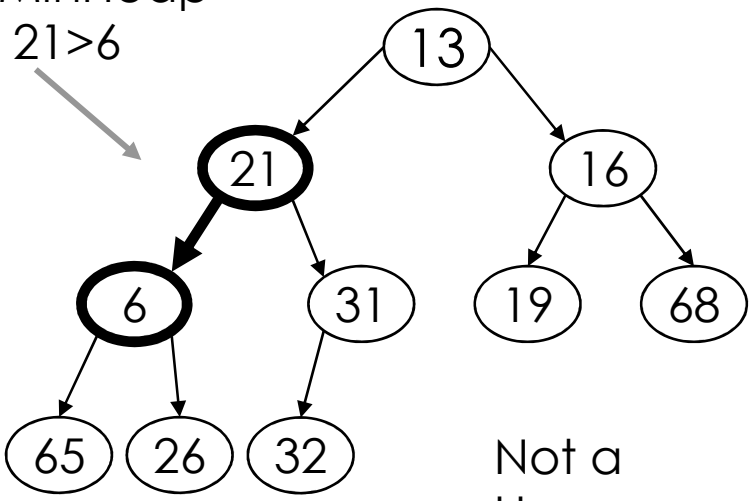
In general, if the number of nodes is N, height is at most

$1 + [\log_2 N] = O(\log N)$

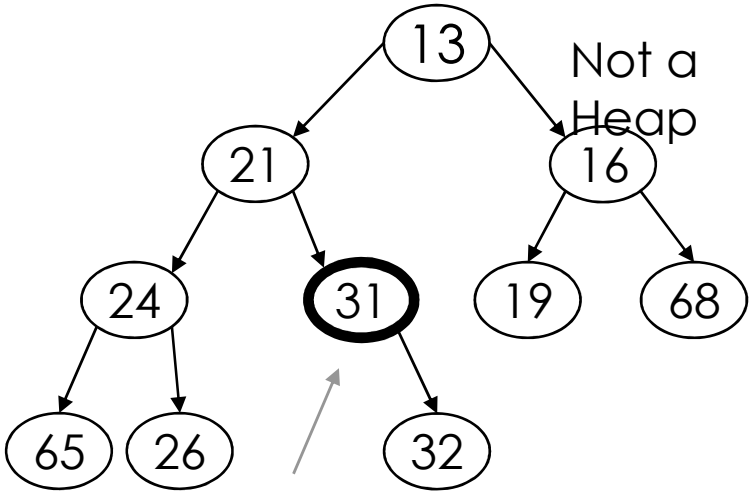
MinHeap and non-Min Heap examples



Violates MinHeap property 21>6

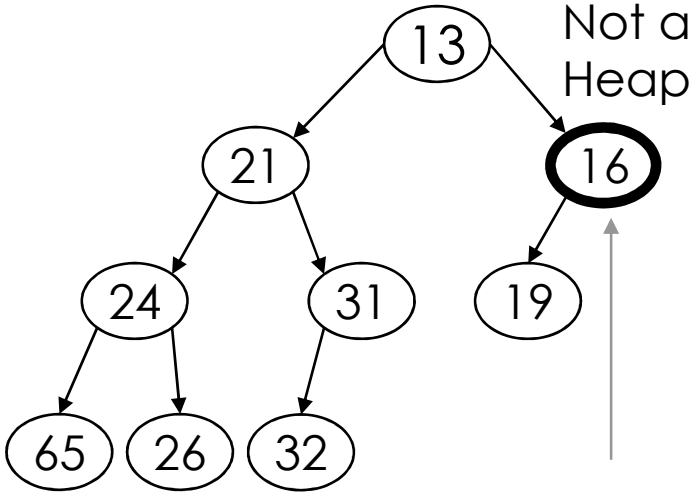


Not a Heap



Not a Heap

Violates heap structural property

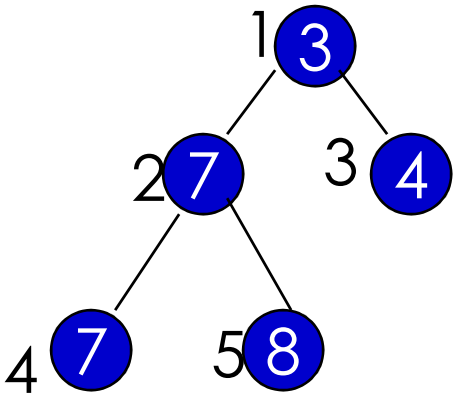


Not a Heap

Violates heap structural property

Heap class definition

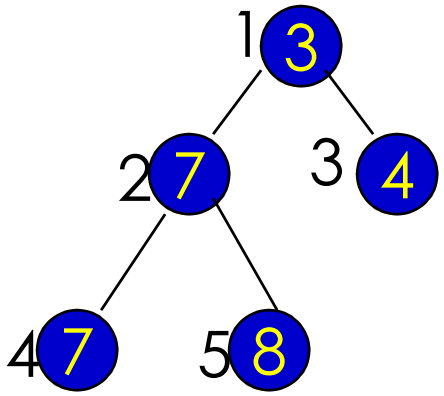
```
class BinaryHeap
   {private:
    vector<Comparable> array;
    int currentSize;
    public:  // member functions
      }
```



A	1	2	3	4	5
	3	7	4	7	8

Note: currentSize is different from the size of the vector (which is the number of allocated memory cells).

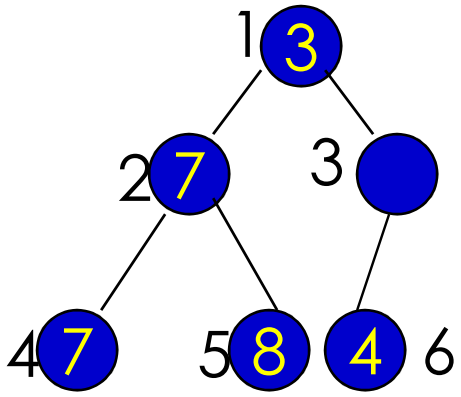
Heap Insertion Example



Insert 2 into the heap

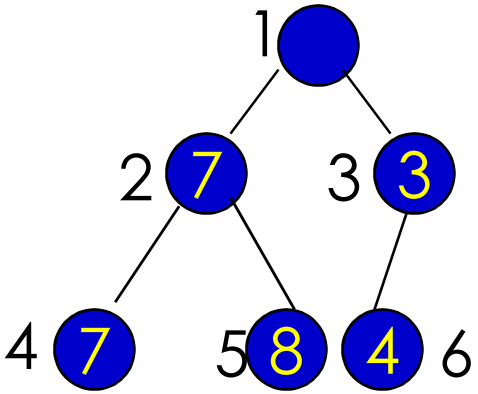
Create a hole and **percolate it up**.

Size = 6, so initially, j= 6; Since H[j/2]= 4 > 2, H[3] is copied to H[6]. Thus the heap now looks as follows:



The new value of j=3. Since H[j/2] =H[1] = 3 is also greater than 2, H[1] copied to H[3]. Now the heap looks as in the next slide.

Heap Insertion Example continued

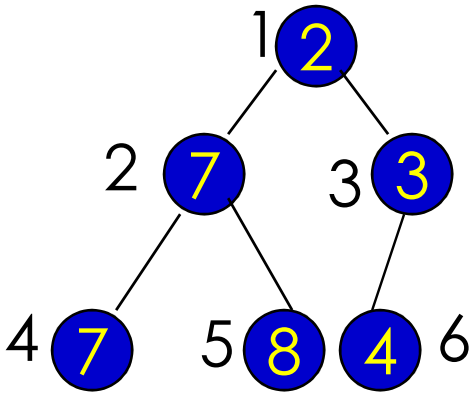


The new value of j = [3/2] = 1.

Now, j/2 = 0 so the iteration stops.

The last line sets H[j] = H[1] = 2.

The final heap looks as follows:



Code for insert

```
void insert(const Comparble & x) {
 if (currentSize == array.size() – 1)
   array.reSize(array.size() * 2);
  //percolate up
 int hole = ++currentSize;

 for (; hole > 1 && x < array[ hole / 2] >; hole/= 2)
   array[ hole ] = array[ hole / 2 ];
 array[ hole ] = x;

}
```

Try some more examples and make sure that the code works correctly.

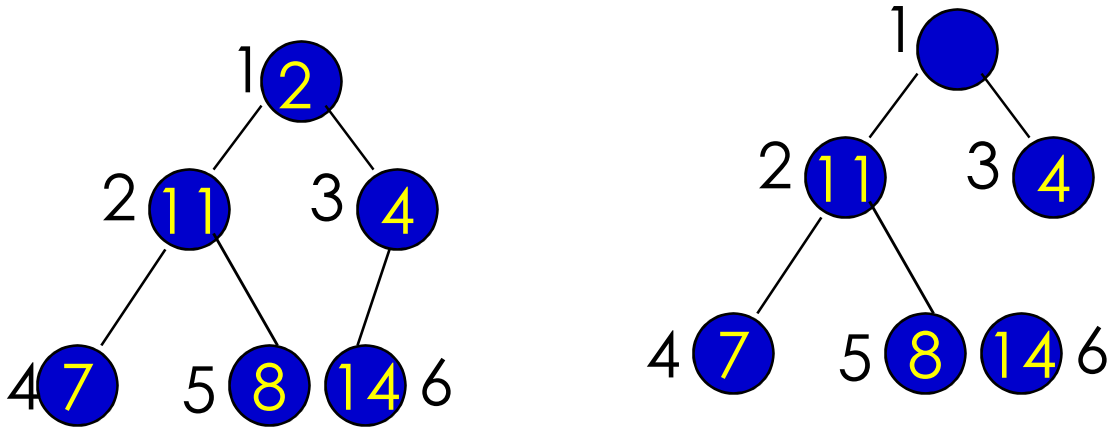
DeleteMin Operation

- Observation: The minimum key is at the root.
- FindMin() can be performed in O(1) time:

```
Comparable findMin() {
  if (currentSize == 0)
    cout << “heap is empty.” << endl;
  else return array[1];
 }
```

- Deleting this key creates a hole at the root and we perform **percolate down** to fill the hole.

Example of DeleteMin

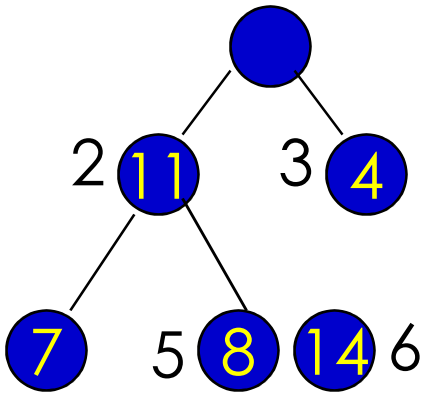


After deleting the min key, the hole is at the root. How should we fill the hole?

We also need to vacate the index 6 since the size of the heap now reduces to 5.

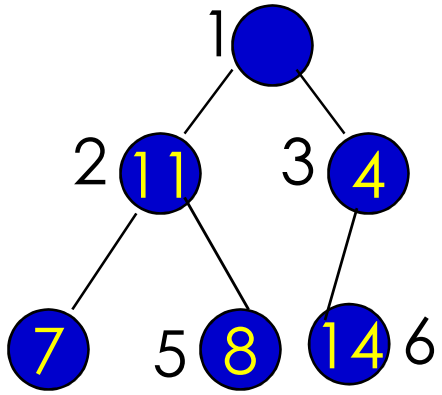
We need to find the right place for key 14.

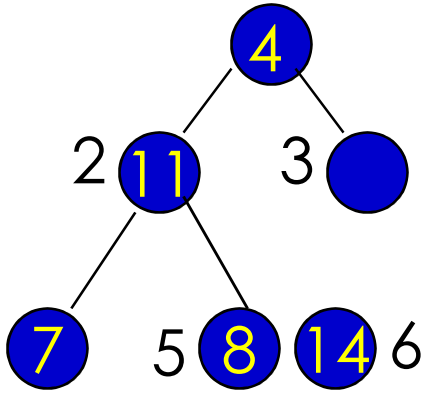
Example of DeleteMin

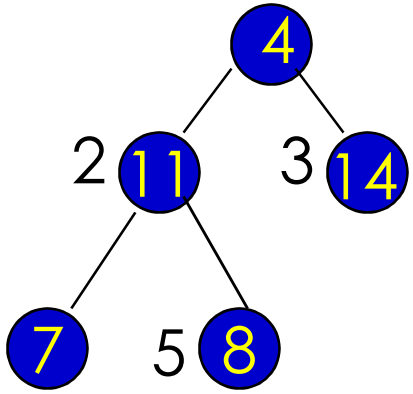


Hole should be filled with smallest key, and it is in array[2] or array[3].

Example of DeleteMin



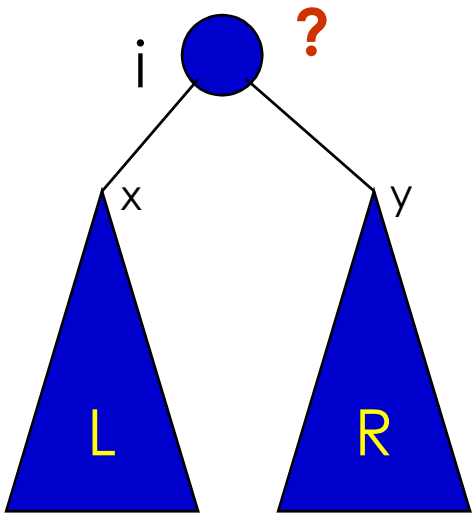




Try some more examples

DeleteMin – Outline of general case

Vacated
key = k



Hole is currently in index i.
its children are 2i and 2i+1.

Suppose array[2i] = x,
array[2i+1]= y.

- Case 1: (k <= x) && (k <= y) In this case, put k in array[I] and we are done.

- Case 2: (k > x) && (y > x). Thus, x is the minimum of the three keys k, x and y. We move the key x up so the hole percolates to 2i. Now the hole percolates down to 2i.

- Case 3: (k >x) && (x >y). Move y up and the hole percolates down to 2i+1.

- Case 4: the hole is at a leaf. put k there and this terminates

Code for delete-min

```
/**
     * Remove the minimum item and place it in minItem.
     * Throws Underflow if empty.
     */
    void deleteMin( Comparable & minItem )
    {
        if( isEmpty( ) )
            throw UnderflowException( );
        minItem = array[ 1 ];
        array[ 1 ] = array[ currentSize-- ];
        percolateDown( 1 );
    }
```

Code for percolateDown

```
void percolateDown( int hole )
    {
        int child;
        Comparable tmp = array[ hole ];

        for( ; hole * 2 <= currentSize; hole = child )
        {
            child = hole * 2;
            if( child != currentSize && array[ child + 1 ] < array[ child ] )
                child++;
            if( array[ child ] < tmp )
                array[ hole ] = array[ child ];
            else
                break;
        }
        array[ hole ] = tmp;
    }
```

Code for percolateDown

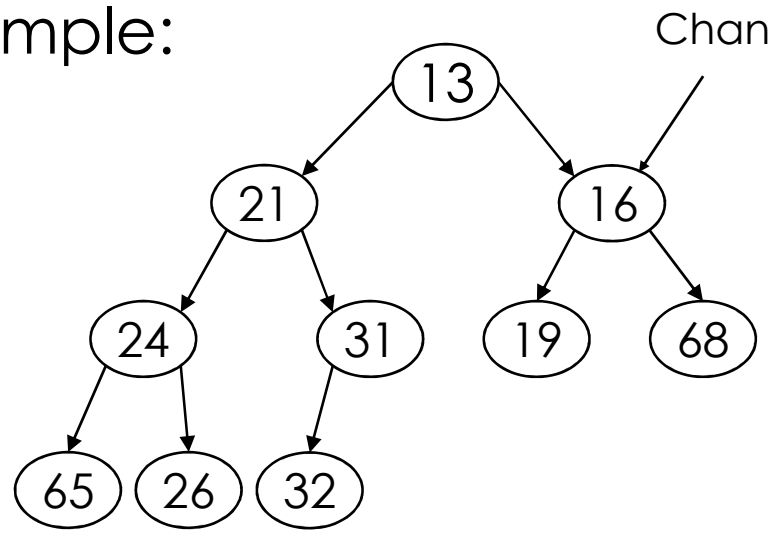
```
void percolateDown( int hole )
    {
        int child;
        Comparable tmp = array[ hole ];

        for( ; hole * 2 <= currentSize;
hole = child )
        {
            child = hole * 2;
            if( child != currentSize &&
array[ child + 1 ] < array[ child ] )
                child++;
            if( array[ child ] < tmp )
                array[ hole ] = array[
child ];
            else
                break;
        }
        array[ hole ] = tmp;
    }
```

Summary: **`percolateDown(j)`** can be called when the heap property is violated at node j in that array[j] could be > array[2j] or array[2j+1]. (All the nodes in the subtree rooted at j satisfy the heap property.) After the call, the heap property is satisfied at all the nodes in the subtree rooted at j.

increaseKey operation

Suppose the priority of a key stored in position p changes. Specifically, we want to increase the key in A[p] by quantity d.

Example:

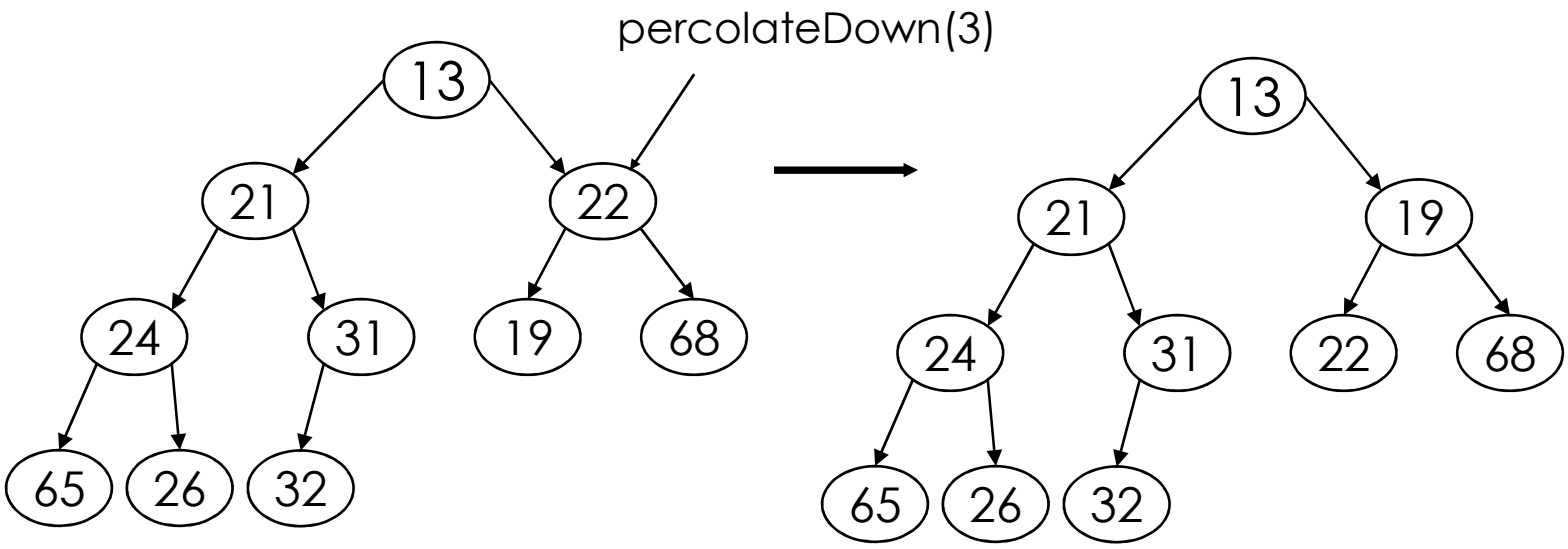


p = 3, d = 6
New key is 22

After changing the key to 22, call percolateDown(3)

increaseKey operation

Example:



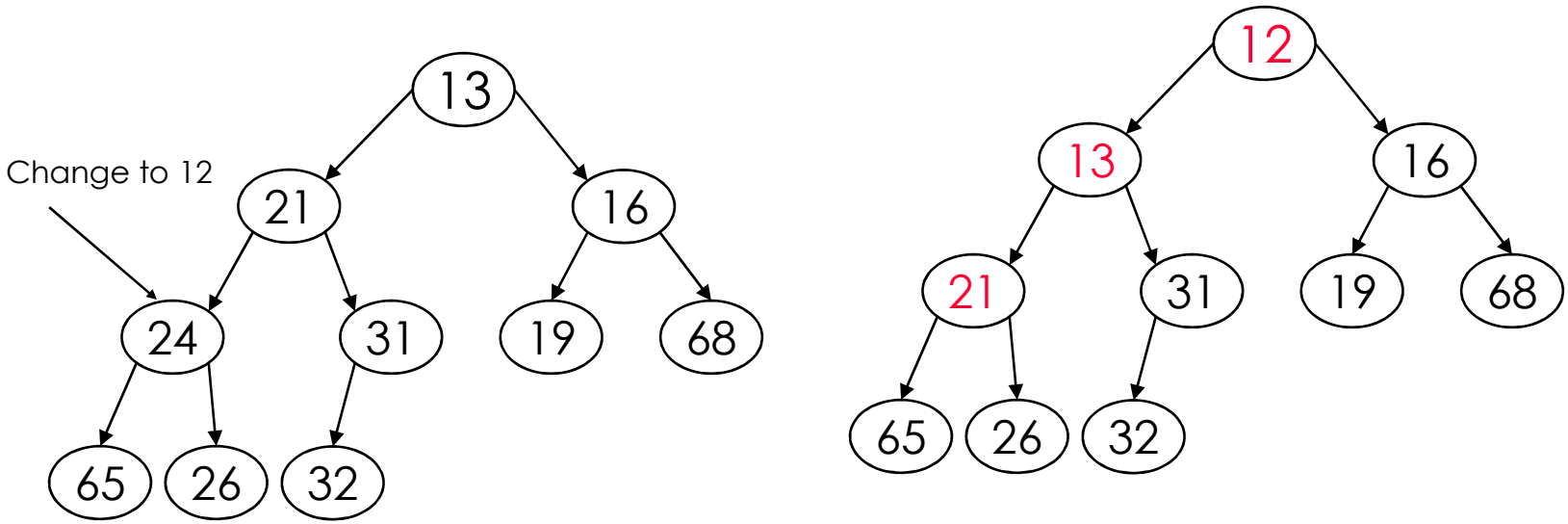
increaseKey operation

```
void increaseKey(int j, int d) {
// change array[j] to array[j]+d and restore heap
// d >= 0 is a pre-condition

 array[j] += d;
 percolateDown(j);
}
```

decreaseKey operation

In this case, we need to percolate the changed key up the heap.

Example: j = 4 and d = 12. Now, it is possible that array[j] is < array[j / 2] so we need to move the key up until the correct place is found.



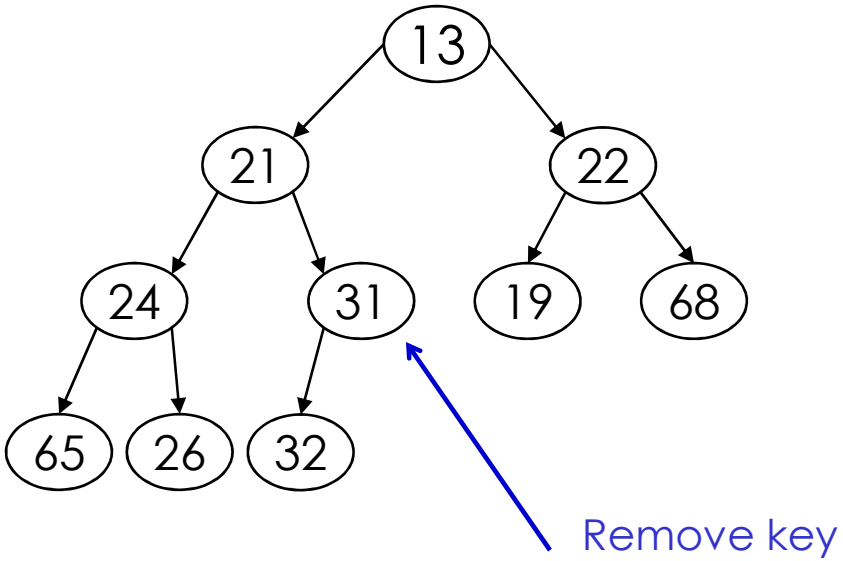
decreaseKey operation

```
void decreaseKey(int j, int d) {
// change array[j] to array[j] – d and restore heap
// d >= 0 is a pre-condition
 int temp = array[j]- d;
 while (j > 1 && array[j/2] > temp){
   array[j] = array[j/2];
   j/= 2;
 };
 array[j] = temp;
 }
```

Note that this procedure is very similar to insert.

Removing a key at index p in a heap

Example: Remove key at index 5



One way to do this:

- apply decreaseKey to make the key the smallest. (Ex: d = 32)
- perform deleteMin

What is the resulting heap?

Heap class constructor

We pass a vector **`items`** containing n (arbitrary) keys to the heap class constructor whose task it is to store these keys as a heap.

- Read these n keys into the heap array.

- call a procedure **`buildHeap`** to convert array into a heap.

```
binaryHeap(const vector<int> & items) {
  for (int i = 0; i < items.size(); ++i)
   array[i+1] = items[i];
  buildHeap();
}
```

Converting an array into a heap

Starting with an arbitrary array, how to convert it to a heap?

Solution 1:
We can view the array as a heap of size 1. Successively insert keys array[2], array[3], etc. into the heap until all the keys are inserted.

```
void buildHeap() {
 for (int j= 2; j <= currentSize; ++j) {
   insert(array[j]);
 }
};
```

This takes O(n log n) time. (n operations each taking O(log n) time.)

Alternate (faster) way to buildHeap

Idea: perform `percolateDown(j)` starting from `j = currentSize/2` to `j = 1`.

Why does it work?

- Before percolateDown(j) is called, we have already performed percolateDown at 2j and 2j+1, and so in the subtree rooted at j, the only node where the heap property may not hold is j.
- Thus, after percolateDown(j) is called, the subtree rooted at j is a heap.
- The last call is percolateDown(1) so the tree rooted at 1 is a heap at the end.

Code for solution 2:

```
void buildHeap() {
  for (int j = currentSize/2; j > 0; --j)
    percolateDown(j);
}
```

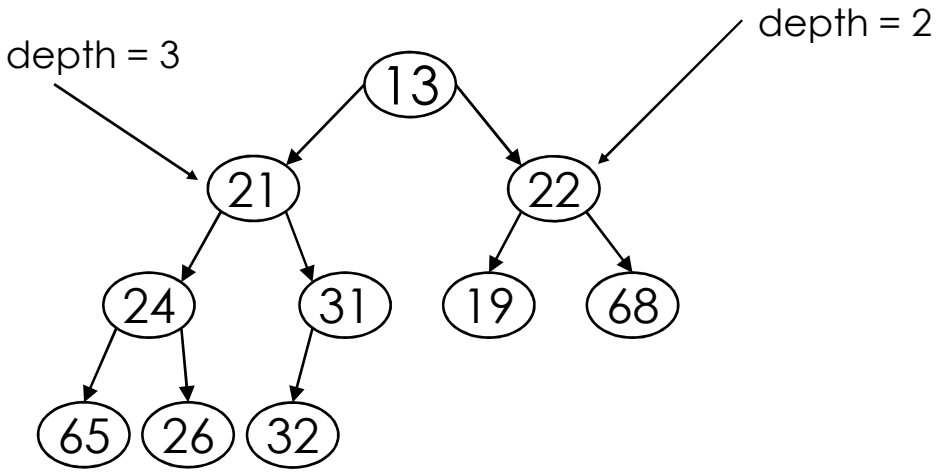
BuildHeap using percolateDown is faster

Solution 2 builds a heap in O(n) time. (Recall solution 1 takes O(n log n) time.)

Let n be such that $2^h < n <= 2^{h+1} - 1$.

To build heap using the second method, percolateDown is called n/2 times.

Depth of a node in a heap: The max number of nodes in a path from that node to a leaf.



Let j be a node at depth = k. Number of operations performed by `percolateDown(j)` is = 3k since the procedure moves down one level after two comparisons and one assignment.

Number of nodes at depth k is 2^{h-k}.

Total work done at depth k is $3k * 2^{h-k}$

For a heap with n nodes, depth varies from 1 (for root) to $\lfloor \log_2 n \rfloor$ for the lowest level non-leaf nodes.

Thus, the total work done =

$$\sum_{k=1}^{h} 3k \times 2^{h-k} = 3\sum_{k=1}^{h} k \times 2^{h-k} = O(n) \text{ if } h = \lfloor \log_2 n \rfloor$$

A summation formula

Series of the kind $\sum_{k=1}^{m} k \times 2^k$ is called AGM (arithmetic-geometric series)

Geometric series is of the form $\sum_{k=1}^{m} c^k$

Formula for the sum $\sum_{k=1}^{m} c^k = \frac{c^{k+1} - 1}{c - 1}$

Formula for the sum of AGM is

$$\sum_{k=1}^{m} k \times 2^k = (m-1)2^{m+1} + 6$$

Using this formula, you can show that the total number of operations performed by buildHeap is O(n).

Heap-sorting

To sort n numbers:

- read the numbers into an array A.
- call heap constructor to build a heap from A.
- perform n deleteMin operations. Store the successive outputs of deleteMin in the array A.
- A is now sorted.

Total number of operations = O(n) (for builing heap)
+ O(n log n) (for n deleteMin operations)

= O(n log n)

Heap-sorting

```
void sort(vector<Comparable> out) {
 // the sorted array is output in array out
  out.resize(currentSize); int j=0;
  while (!isEmpty())
   out[j++] = deleteMin();
}
```

It is possible to collect the output in the heap array itself. As the heap size is shrinking, we can use the unused slots of the array to collect the delete min keys. The resulting heap will be sorted in descending order.

Machine Scheduling

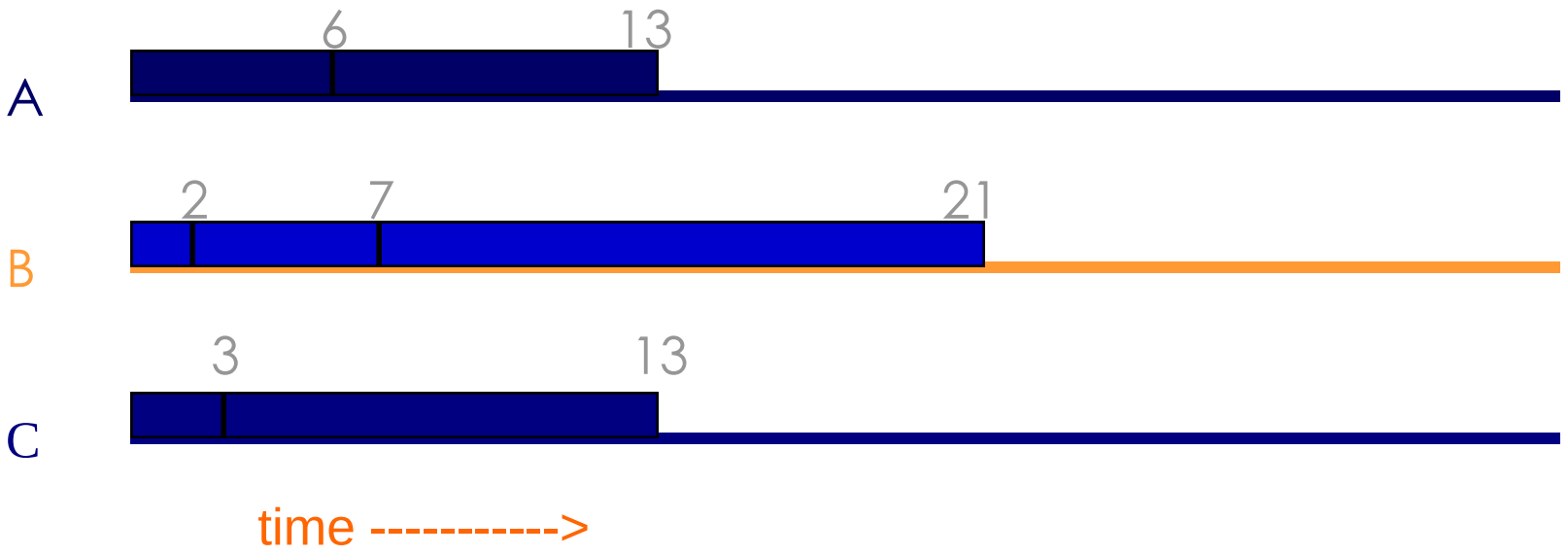
- m identical machines (drill press, cutter, sander, etc.)
- n jobs/tasks to be performed
- assign jobs to machines so that the time at which the last job completes is minimum

Machine Scheduling Example

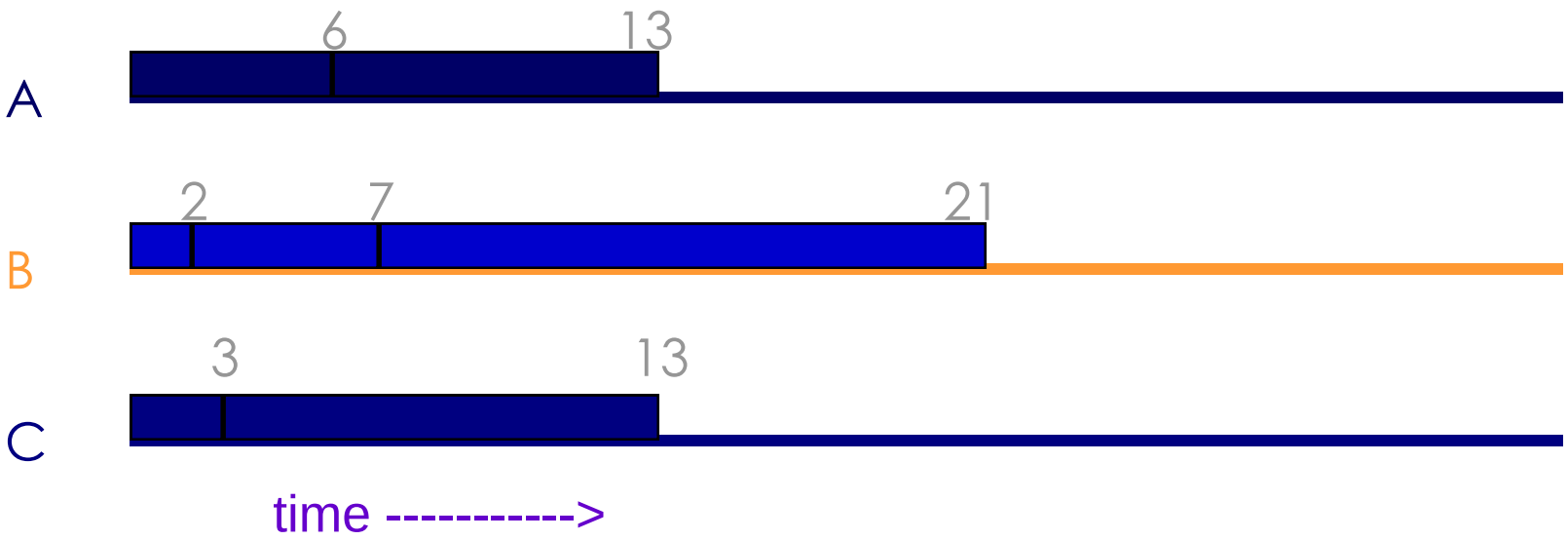
3 machines and 7 jobs

job times are [6, 2, 3, 5, 10, 7, 14]

possible schedule



Machine Scheduling Example



Finish time = 21

Objective: Find schedules with minimum finish time.

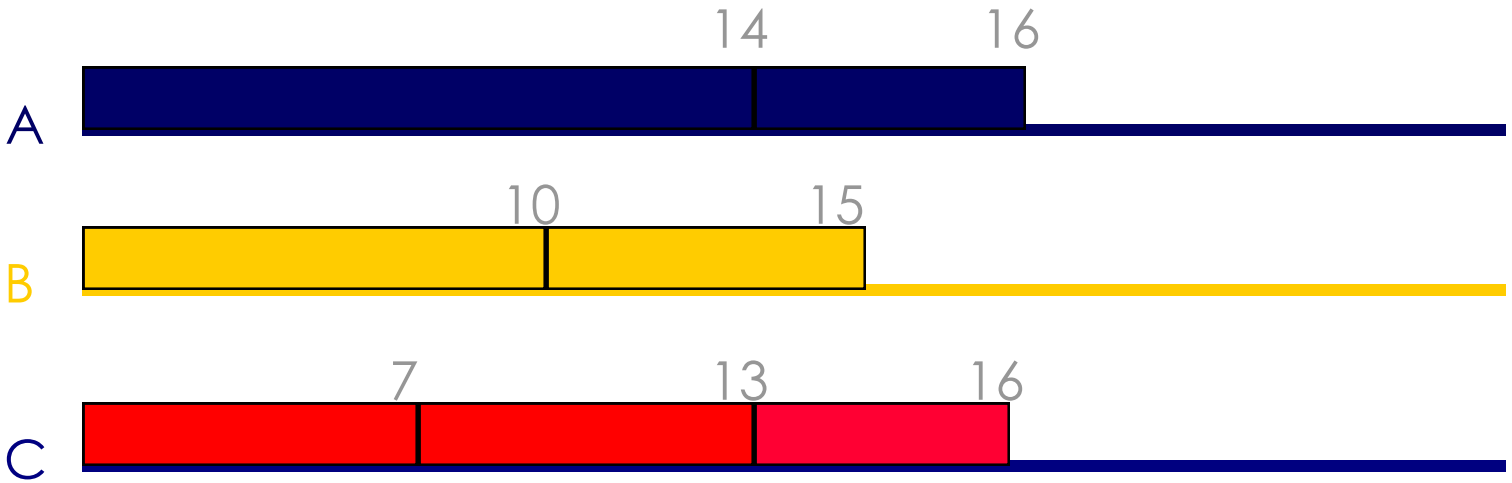
LPT Schedules

- 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.
- For each machine, maintain information about when it is ready.
- Choice of data structure: priority queue

LPT Schedule

[14, 10, 7, 6, 5, 3, 2]



Finish time is 16!

LPT Schedule

- LPT rule does not guarantee minimum finish time schedules.

- Usually LPT finish time is very close to minimum finish time.

Complexity Of LPT Scheduling

- Sort jobs into decreasing order of task time.
  - O(n log n) time (n is number of jobs)
- Schedule jobs in this order.
  - assign job to machine that becomes available first
  - must find minimum of m (m is number of machines) finish times
  - takes O(m) time using simple strategy
  - so need O(mn) time to schedule all n jobs.

LPT implementation using a minheap

- Minheap has the finish times of the m machines.
- Initial finish times are all 0.
- To schedule a job remove the machine with minimum finish time from the priority queue.
- Update the finish time of the selected machine and put the machine back into the priority queue.

LPT implementation using a minheap

```
ListPtrs LPTschedule(vector<int> jobs, int m) {
 // jobs is a sorted array of job durations, m is the
 // number of machines
  ListPtrs L(m); // array of m null pointers
  BinaryHeap<pair> H;
  for (int j=0; j < jobs.size(); ++j) {
     pair p = new pair(j,0);
     H.insert(p);
   }
  for (int j=0; j < jobs.size(); ++j) {
    pair m = H.deleteMin();
    int id = m.getId(); int ft = m.getFinishTime();
    L[id].insert(j); ft+=jobs[j];
    m = new pair(id, ft); H.insert(m);
  }
}
```

LPT implementation using a minheap

- Sorting n jobs takes O(n log n) steps.
- Initialization of priority queue with m 0's
  - O(m) time
- 1 remove min and 1 put to schedule each job
  - each put and remove min operation takes O(log m) time
- time to schedule is O(m + n log m)
- overall time is

$$O(n \log n + n \log m + m)$$