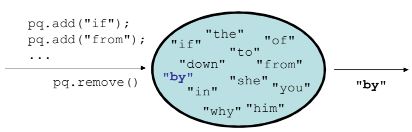
Priority Queues and Heaps

Prioritization problems

- **print jobs:** e.g.Lab printers constantly accept and complete jobs from all over the building. We want to print faculty jobs before staff before student jobs, and grad students before undergrad, etc.
- **ER scheduling:** Scheduling patients for treatment in the ER. A gunshot victim should be treated sooner than a guy with a cold, regardless of arrival time. How do we always choose the most urgent case when new patients continue to arrive?
- *key operations we want:*
  - ***add** an element (print job, patient, etc.)*
  - ***get/remove** the **most "important"** or "urgent" element*

Priority Queue ADT

- **priority queue**: A collection of ordered elements that provides fast access to the minimum (or maximum) element.
  - `add` adds in order
  - `peek` returns **minimum** or "highest priority" value
  - `remove` removes/returns **minimum** value
  - `isEmpty`, `clear`, `size` O(1)



priority queue

Question: PQ interface

What's the output? Assume lower value means higher priority.

```
PriorityQueue q;
q.add(35);
q.add(25);
cout << q.remove() << " " ;
q.add(15);
cout << q.peek() << " ";
q.add(5);
cout <<q.remove() << " ";
cout << q.remove() << " ";
```

Simple Implementations

- Array
- Sorted array
- A simple linked list
- A sorted linked list
- A binary search tree
- Binary Heap

5

Array?

- Consider using an unfilled array to implement a priority queue.
  - `add`: Store it in the next available index.
  - `peek`: Loop over elements to find minimum element.
  - `remove`: Loop over elements to find min. Shift to remove.

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);.
queue.remove();
```

index	0	1	2	3	4	5	6	7	8	9
value	9	23	8	-3	49	12	0	0	0	0
size	6									

- How efficient is `add`? `peek`? `remove`?
  - O(1), O(*N*), O(*N*)
  - (`peek` must loop over the array; `remove` must shift elements)

Sorted array?

- Consider using a *sorted* array to implement a priority queue.
  - `add`: Store it in the proper index to maintain sorted order.
  - `peek`: Minimum element is in index [0].
  - `remove`: Shift elements to remove min from index [0].

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```

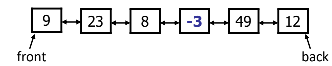
index	0	1	2	3	4	5	6	7	8	9
value	-3	8	9	12	23	49	0	0	0	0
size	6									

  - How efficient is `add`? `peek`? `remove`?
    - O(*N*), O(1), O(*N*)
    - (`add` and `remove` must shift elements)

Linked list?

- Consider using a doubly linked list to implement a priority queue.
  - `add`: Store it at the end of the linked list.
  - `peek`: Loop over elements to find minimum element.
  - `remove`: Loop over elements to find min. Unlink to remove.

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```

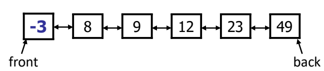


  - How efficient is `add`? `peek`? `remove`?
    - O(1), O(*N*), O(*N*)
    - (`peek` and `remove` must loop over the linked list)

Sorted linked list?

- Consider using a *sorted* linked list to implement a priority queue.
  - `add`: Store it in the proper place to maintain sorted order.
  - `peek`: Minimum element is at the front.
  - `remove`: Unlink front element to remove.

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```

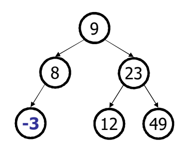


  - How efficient is `add`? `peek`? `remove`?
    - O(*N*), O(1), O(1)
    - (`add` must loop over the linked list to find the proper insertion point)

Binary search tree?

- Consider using a binary search tree to implement a PQ.
  - `add`: Store it in the proper BST L/R - ordered spot.
  - `peek`: Minimum element is at the far left edge of the tree.
  - `remove`: Unlink far left element to remove.

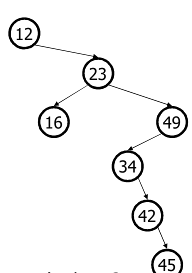
```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
```



  - How efficient is `add`? `peek`? `remove`?
    - O(log *N*), O(log *N*), O(log *N*)...?
    - (good in theory, but the tree tends to become unbalanced to the right)

Unbalanced binary tree

```
queue.add(9);
queue.add(23);
queue.add(8);
queue.add(-3);
queue.add(49);
queue.add(12);
queue.remove();
queue.add(16);
queue.add(34);
queue.remove();
queue.remove();
queue.add(42);
queue.add(45);
queue.remove();
```



  - Simulate these operations. What is the tree's shape?
  - A tree that is *unbalanced* has a height close to *N* rather than log *N*, which breaks the expected runtime of many operations.

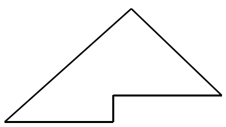
Binary Heap

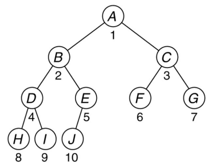
- The binary heap is the classic method used to implement priority queues.
- We usually use the term **heap** to refer to binary heap.
- Heap is different from the term *heap* used in dynamic memory allocation.
- Heap has two properties:
  1. Structure property
  2. Ordering property

12

Structure Property

- A **binary heap** is a *complete binary tree*
- A **complete binary tree** is a tree where every level is full except possibly the lowest level, which must be filled from left to right





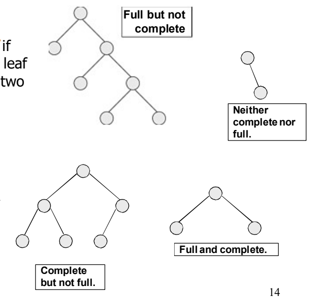
13

Full and Complete Binary Trees

Here are two important types of binary trees.

Definition: a binary tree T is *full* if each node is either a leaf or possesses exactly two child nodes.

Definition: a binary tree T with n levels is *complete* if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.



14

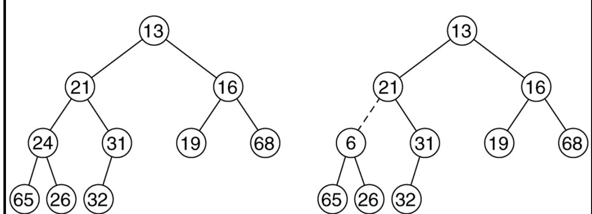
Heap-Order Property

- In a heap (a.k.a **min heap**), for every node X with parent P, the key in P is <u>smaller</u> than or equal to the key in X.
- Thus the minimum element is always at the root.
  - Thus we get the operation peek in constant time.

Q. Is a heap a BST? How are they related?

15

Two complete trees: (a) a heap; (b) not a heap



(a) (b)

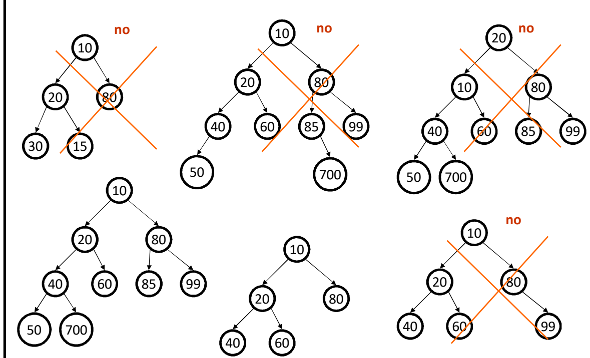
16

Max Heap

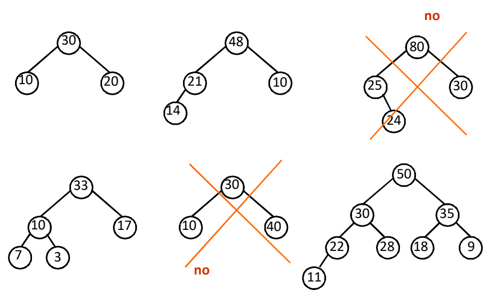
- Heap order property is slightly different :
- In a **max heap** for every node X with parent P, the key in P is <u>greater</u> than or equal to the key in X.
- <u>The default is **min heap**.</u>

17

Which are min-heaps?



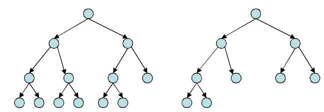
Which are max-heaps?



19

Heap height and runtime

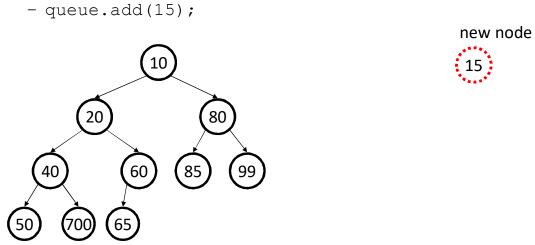
- The height of a complete tree is always $\log_2 N$.
- Because of this, if we implement a priority queue using a heap, we can provide the following runtime guarantees:
  - `add`: O(log *N*)
  - `peek`: O(1)
  - `remove`: O(log *N*)



n-node complete tree of height h:
$2^h \leq n \leq 2^{h+1} - 1$
$h = \lfloor \log n \rfloor$

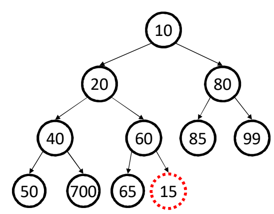
The add operation

- When an element is added to a heap, where should it go?
  - Must insert a new node while maintaining heap properties.
  - `queue.add(15);`



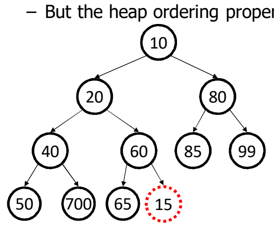
The add operation

- When an element is added to a heap, where should it go?
  - Must insert a new node while maintaining heap properties.
  - `queue.add(15);`



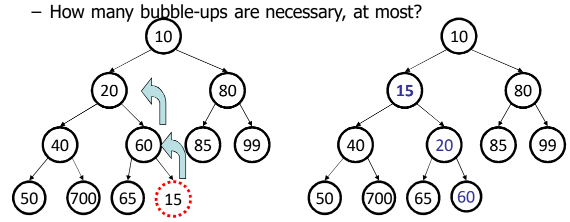
The add operation

- When an element is added to a heap, it should be initially placed as the *rightmost leaf* (to maintain the completeness property).
  - But the heap ordering property becomes broken!



"Bubbling up" a node

- **bubble up**: To restore heap ordering, the newly added element is shifted ("bubbled") up the tree until it reaches its proper place.
  - Weiss (textbook): *"percolate up"* by swapping with its parent
  - How many bubble-ups are necessary, at most?

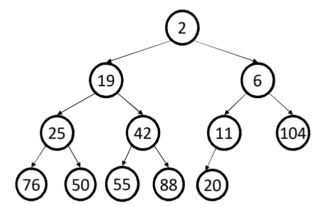


Bubble-up exercise

- Draw the tree state of a min-heap after adding these elements:
  - 6, 50, 11, 25, 42, 20, 104, 76, 19, 55, 88, 2

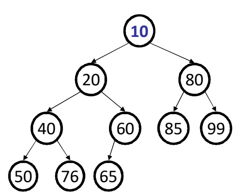
Bubble-up exercise

- Draw the tree state of a min-heap after adding these elements:
  - 6, 50, 11, 25, 42, 20, 104, 76, 19, 55, 88, 2



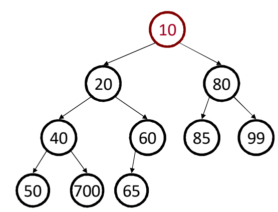
The peek operation

- A peek on a min-heap is trivial to perform.
  - because of heap properties, minimum element is always the root
  - O(1) runtime
- Peek on a max-heap would be O(1) as well (return max, not min)



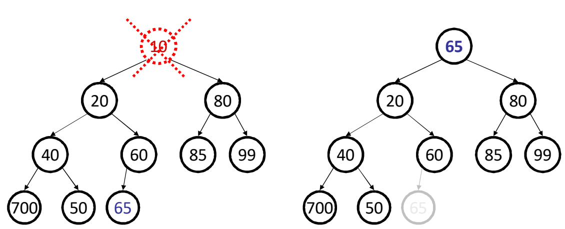
The remove operation

- When an element is removed from a heap, what should we do?
  - The root is the node to remove. How do we alter the tree?
  - `queue.remove();`



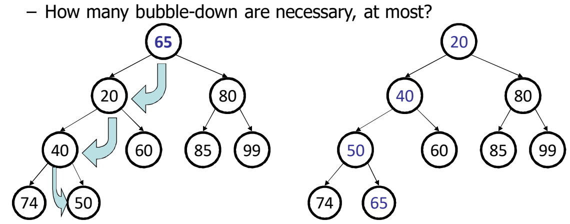
The remove operation

- When the root is removed from a heap, it should be initially replaced by the *rightmost leaf* (to maintain completeness).
  - But the heap ordering property becomes broken!



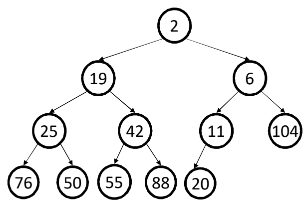
"Bubbling down" a node

- **bubble down**: To restore heap ordering, the new improper root is shifted ("bubbled") down the tree until it reaches its proper place.
  - Weiss: *"percolate down"* by swapping with its *__smaller__* child (why?)
  - How many bubble-down are necessary, at most?

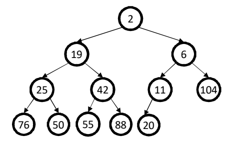


Bubble-down exercise

- Suppose we have the min-heap shown below.
- Show the state of the heap tree after remove has been called 3 times, and which elements are returned by the removal.

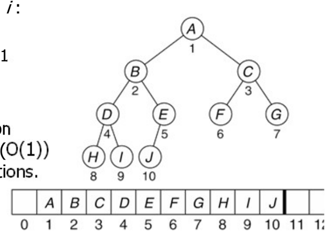


Bubble-down exercise



Array heap implementation

- Though a heap is conceptually a binary tree, since it is a *complete* tree, when implementing it we actually can "cheat" and just *use an array*!
  - index of root = 1 (leave 0 empty to simplify the math)
  - for any node *n* at index *i* :
    - index of *n*.left = $2i$
    - index of *n*.right = $2i + 1$
    - parent index of *n*?
  - This array representation is elegant and efficient (O(1)) for common tree operations.

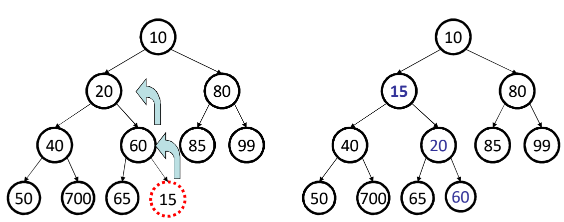


Implementing HeapPQ

```
template <class T>
class HeapPriorityQueue
{
  public:
    HeapPriorityQueue( int capacity = 100 );
    bool isEmpty( ) const;
    const T & peek( ) const;
    void add(  T & x );
    T remove( );
    void makeEmpty( );
    void print();

  private:
    int theSize;       // Number of elements in heap
    vector<T> arr;     // The heap array
    void buildHeap( );
    void bubbleDown( int hole );
};
```

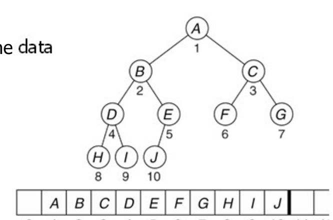
Implementing add

- Let's write the code to add an element to the heap:



Resizing a heap

- What if our array heap runs out of space?
  - What must we do here?
  - We must enlarge it.
  - We can simply copy the data into a larger array.

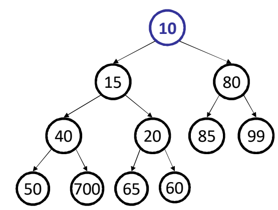


Implementing add

```
// Adds the given value to this priority queue in order.
template <class T>
void HeapPriorityQueue<T>::add(const  T & x )
{
    arr[ 0 ] = x;           // initialize sentinel
    if( theSize + 1 == arr.size( ) )
        arr.resize( arr.size( ) * 2 + 1 );

    // Bubble up
    int hole = ++theSize;
    for( ; x < arr[ hole / 2 ]; hole /= 2 )
        arr[ hole ] = arr[ hole / 2 ];
    arr[ hole ] = x;
}
```

Implementing peek

- Let's write code to retrieve the minimum element in the heap:

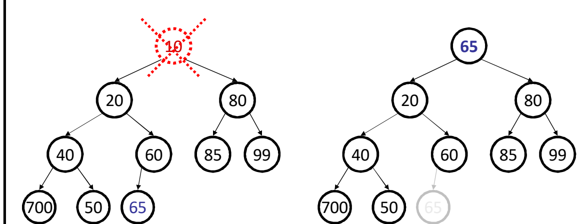


Implementing peek

```
// Returns the minimum element in this priority queue.
// precondition: queue is not empty
template <class T>
const T& HeapPriorityQueue<T>::peek() const{
    return arr[1];
}
```

Implementing remove

- Let's write code to remove the minimum element in the heap:



Implementing remove

```
// Remove the smallest item from the priority queue.
// Throw UnderflowException if empty.

template <class T>
T HeapPriorityQueue<T>::remove( )
{
    if( isEmpty( ) )
        throw UnderflowException( );
    T tmp = arr[1];
    arr[ 1 ] = arr[ theSize-- ];
    bubbleDown( 1 );
    return tmp;
}
```

Bubble down

```
// Internal method to bubble down in the heap.
// hole is the index at which the percolate begins.
template <class T>
void HeapPriorityQueue<T>::bubbleDown( int hole )
{
  int child;
  T tmp = arr[ hole ];

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

Summary: PQ Implementation Alternatives

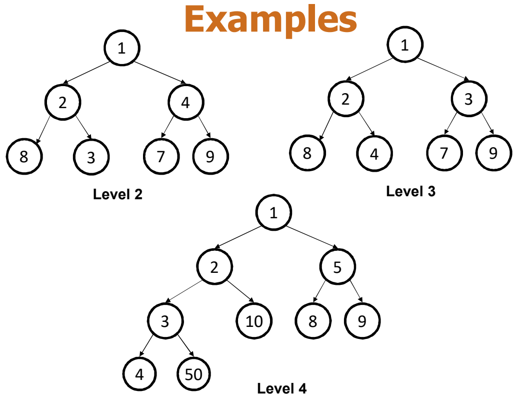
	Add()	Remove()	Peek()
Array	O(1)	O(n)	O(n)
Sorted array	O(n)	O(n)	O(1)
A simple linked list	O(1)	O(n)	O(n)
A sorted linked list	O(n)	O(1)	O(1)
A binary search tree	O(n)	O(n)	O(n)
AVL	O(log n)	O(log n)	O(log n)
Binary Heap	O(log n)	O(log n)	O(1)

Question

Which nodes in a min-heap could possibly contain the **fourth-smallest** element in the heap, assuming there are no equal elements? (root is at level 1)

a) anywhere except level 1 (root)
b) anywhere in level 2 and 3, but not 4
c) any where in level 3, or in the left half of level 4
d) anywhere in level 2, 3, or 4
e) anywhere in level 3, 4, 5

Examples



Example

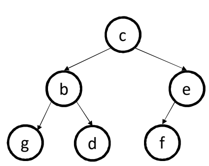
- The array representation of a min-heap is as follows: [13,28,71,29,40,84,87,45] What are the min-heap contents after adding the value 21?

[13, 21, 71, 28, 40, 84, 87, 45, 29]

Question

If we add the value "a" to this heap, what does the heap look like (array representation) afterwards?

a) c b e g d f a
b) a b c g d f e
c) c b a g d f e
d) a b c e g f c



Example

- Draw the tree for the min-heap that results from inserting 5, 8, 4, 9, 12, 6, 7, 3 in that order into an initially empty heap. Write your answer as the array representation of the resulting min-heap.

[3, 4, 5, 8, 12, 6, 7, 9]

Building a Heap

- Inserting N items into an empty heap one by one: *O(NlogN)* worst case.
- However we can build a heap out of N items in linear time *O(N)* by applying a bubble down routine to nodes in reverse level order.

`buildHeap` method

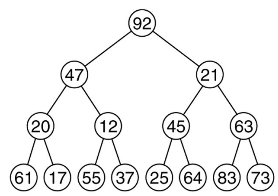
```
// Establish heap-order property from an arbitrary
// arrangement of items. Runs in linear time.

template <class Comparable>
void BinaryHeap<Comparable>::buildHeap( )
{
    for( int i = theSize / 2; i > 0; i-- )
        bubbleDown( i );
}
```

Example

- Given the following integers, build a heap in O(n) time using buildHeap method.

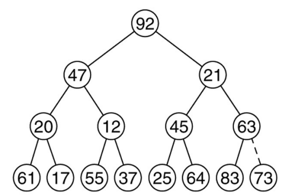
  92, 47, 21, 20, 12, 45, 63, 61, 17, 55, 37, 25, 64, 83, 73

Implementation of the linear-time buildHeap method



(a)

Initial complete tree

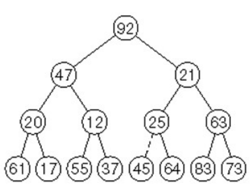


(b)

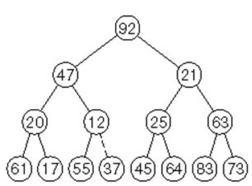
After bubbleDown(7)

52

(a) After bubbleDown(6); (b) after bubbleDown(5)



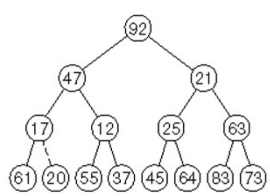
(a)



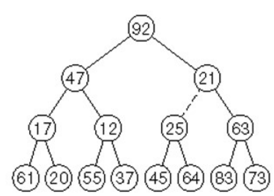
(b)

53

(a) After bubbleDown(4); (b) after bubbleDown(3)



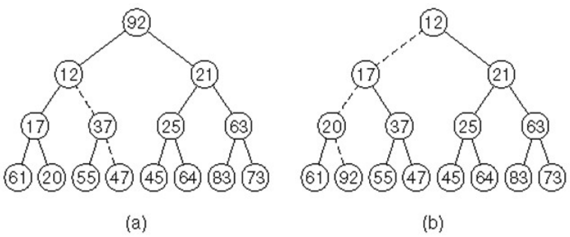
(a)



(b)

54

(a) After bubbleDown(2); (b) after bubbleDown(1) and buildHeap terminates



55

Analysis of `buildHeap`

- The linear time bound of `buildHeap`, can be shown by computing the sum of the heights of all the nodes in the heap, which is the maximum number of dashed lines.
- For the perfect binary tree of height h containing $N = 2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $N - H - 1$.
- Thus it is $O(N)$.

Question

Given the following numbers, construct a min-heap using the build-heap method: 9 8 7 5 3 1

What is the array representation of the heap?

a) 1 5 3 9 7 8
b) 1 3 5 7 8 9
c) 3 5 1 9 8 7
d) 1 3 7 5 8 9

Exercise

- Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.
- Example:

Input: [3,2,1,5,6,4] and k = 2

Output: 5

Algorithm

```
//Using min-heap
int kthSmallest(int arr[], int n, int k) {
   // Assuming there is a constructor to build a min heap of n
   // elements from a given array: O(n) time
   HeapPriorityQueue<int> mh (arr, n);

   // Do extract min (n-k) times
   for (int i = 0; i < n-k ; i++)
      mh.remove();

   // Return root
   return mh.peek();
}   // alternatively use max-heap and delete k-1 elements
```