CS 32 Week 10
Discussion 1I

Srinath

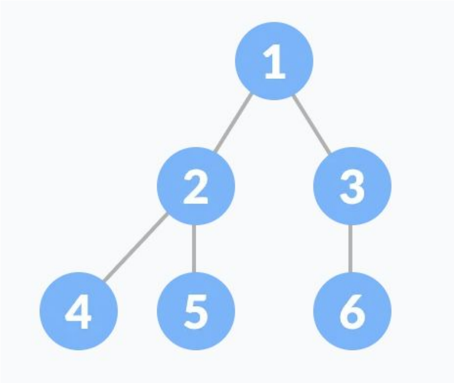
Outline

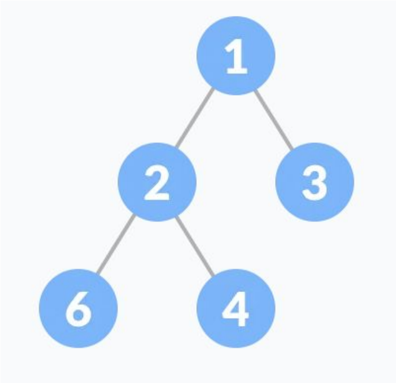
- Complete Binary Tree
- Heaps
- Priority Queue : STL
- Worksheet 9 - Heaps

Complete Binary Tree

Complete Binary Tree : Definition

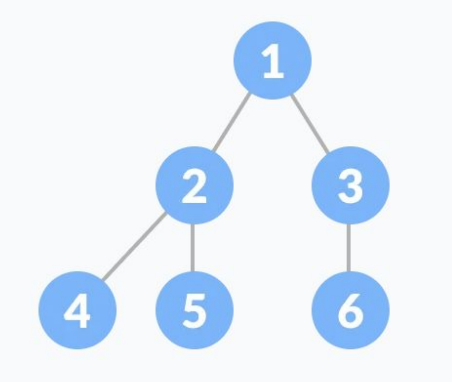
A binary tree with **all levels** completely **filled** except the lowest one, Lowest one is filled from left

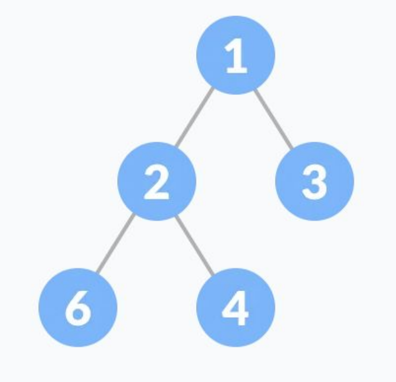


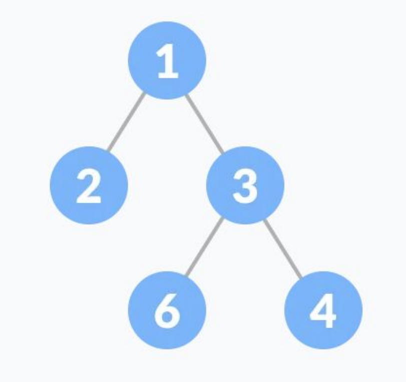


Complete Binary Tree : Definition

A binary tree with **all levels** completely **filled** except the lowest one, Lowest one is filled from left

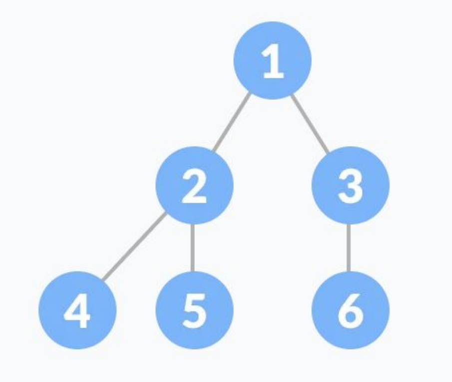


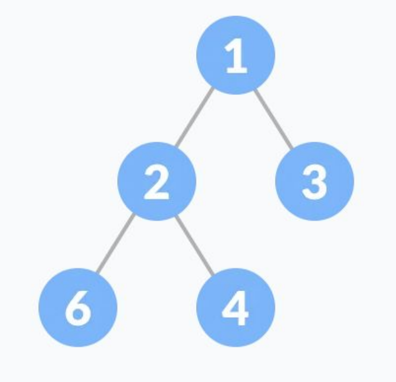


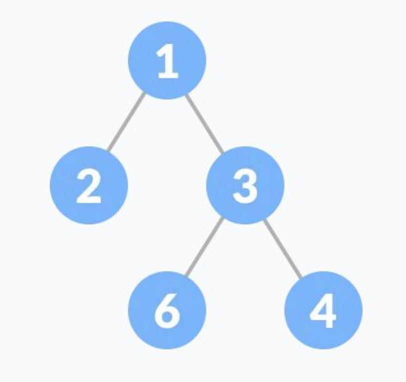


Complete Binary Tree : Definition

A binary tree with **all levels** completely **filled** except the lowest one, Lowest one is filled from left



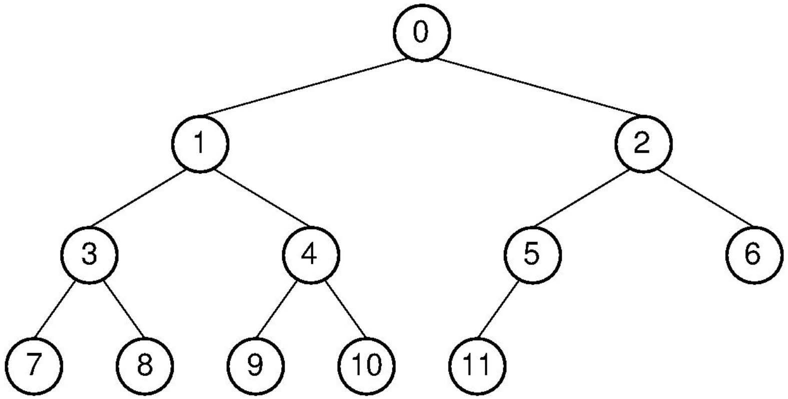




Not a CBT

Complete Binary Tree : Definition

A binary tree with **all levels** completely **filled** except the lowest one, Lowest one is filled from left



(a)

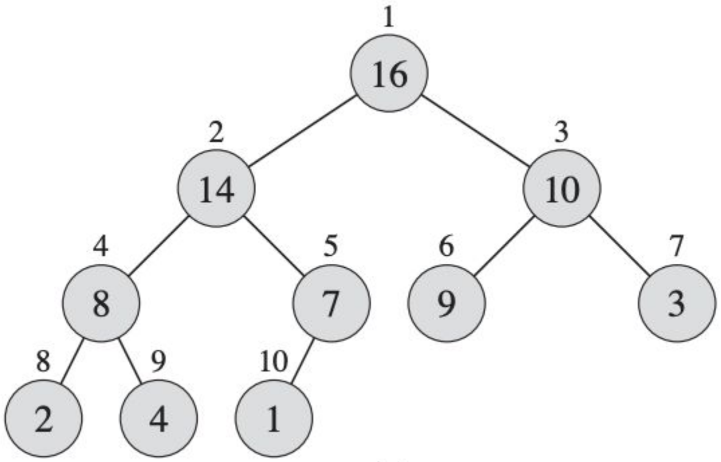
Complete Binary Tree : Representation

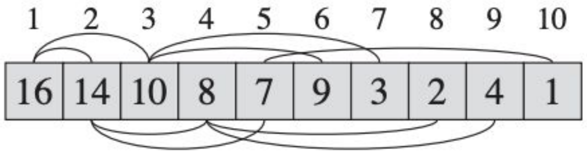
An **N** Node CBT can be represented using an array

For each node at **i,** its **left** child is **2*i, right** child is **2*i+1**, **parent** is ⌊i/2⌋

Or..

For each node at **i,** its **left** child is **2*i+1, right** child is **2*i+2**, **parent** is ⌊(i-1)/2⌋(if you prefer **0 -** indexing)

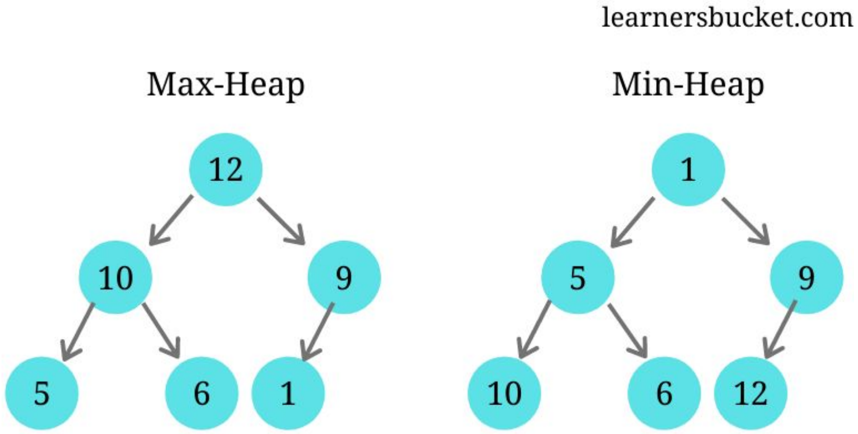




Heaps

Heaps : Definition

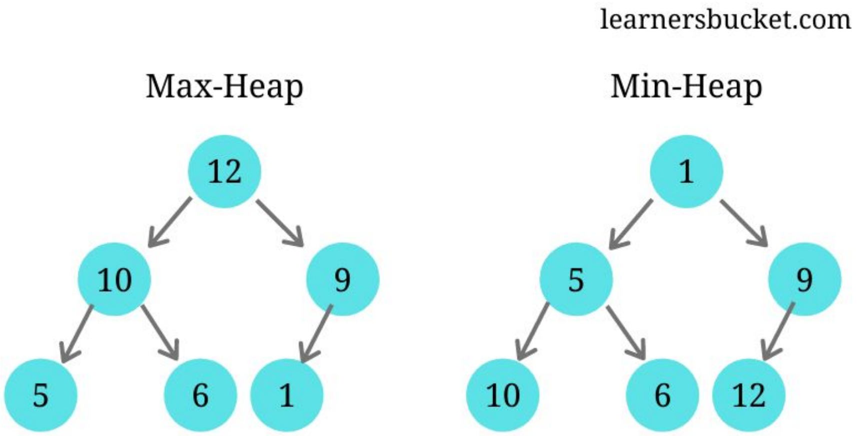
Heap is a CBT, satisfying **heap-property**.
For a **max-heap,** all parents have greater value than their children.
For a **min-heap,** all parents have smaller value than their children.



Heaps : Definition

As it is a CBT, you can use an array or vector to represent it.

```
//heap of valuetype double
//may also use fixed-size array for heap
vector<double> heap;
```



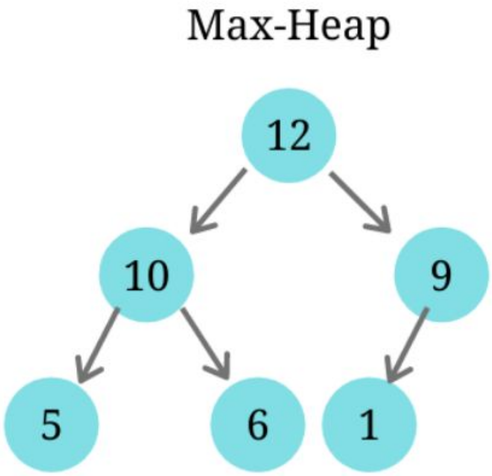
Heaps : Insertion

How to insert a new element into existing heap?

```
void insert(vector<double>& heap, const double& val) {
  //insert a value val to heap
}
```

Heaps : Insertion

How to insert a new element(say 25) into existing heap?



Heaps : Insertion

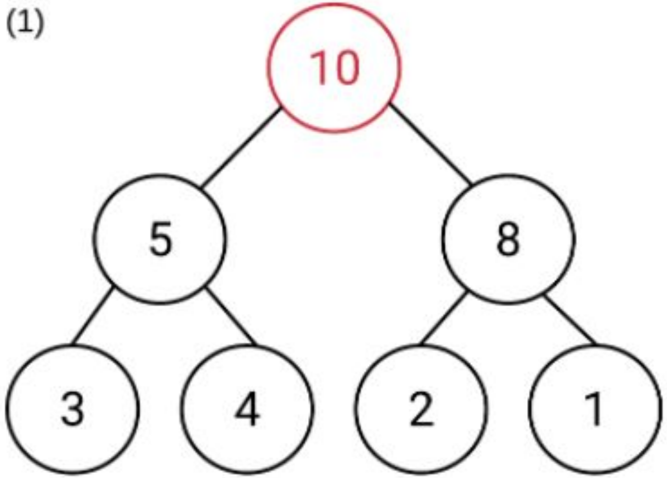
```
void insert(vector<double>& heap, const double& val) {
  heap.push_back(val);
  int cur_ind = heap.size() - 1;
  while(cur_ind != 0
        && heap[cur_ind] > heap[(cur_ind-1)/2]) {
    swap(heap[cur_ind], heap[(cur_ind-1)/2]);
    cur_ind = (cur_ind-1)/2;
  }
}
```

Heaps : Deletion

We are interested in deleting the root element of heap in general.
Why?

Heaps : Deletion

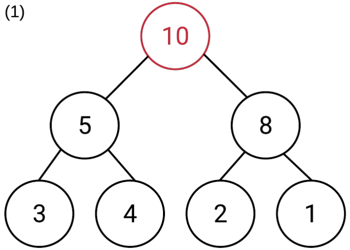
We are interested in deleting the root element of heap in general.
Why? - We use up(consume) the max/min element in general



Heaps : Deletion

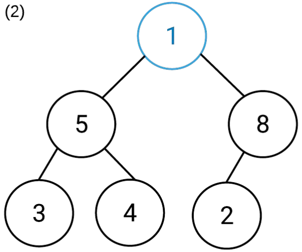
Deleting from this heap

(1)



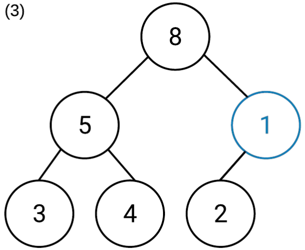
Starting with this max heap

(2)



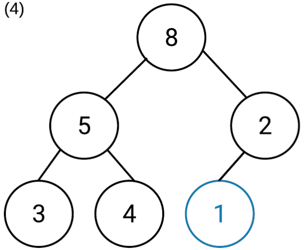
Step 1: the bottom most, left most node, the 1 node, gets placed at the root

(3)



Step 2: Because 1 is less than both of its children, it swaps with the larger element, the 8 node

(4)



Step 3: Once again, 7 is bigger than its parent, the 6 node, so it gets swapped

Heaps : Deletion

Recursive

Swap **root** and **last** elements
pop the last element.
call MAX-HEAPIFY(A, 0);

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Heaps : Deletion

Non-Recursive

```
void remove(vector<double>& heap) {
  swap(heap[0], heap[heap.size() - 1]); //swap root to leaf
  heap.pop_back();
  int sz = heap.size();
  int cur = 0;
  while (2 * cur + 1 < sz) {
    if (2 * cur + 2 >= sz) { //only left child exists
      if (heap[2 * cur + 1] > heap[cur]) {
        swap(heap[2 * cur + 1], heap[cur]);
        cur = 2 * cur + 1;
      }
      else break;
    }
    else {
      //larger than both left and right
      if (heap[cur] > heap[2 * cur + 1] && heap[cur] > heap[2 * cur + 2])
        break;
      //pick the larger element of left and right
      if (heap[2 * cur + 1] > heap[2 * cur + 2]) {
        swap(heap[cur], heap[2 * cur + 1]);
        cur = 2 * cur + 1;
      }
      else {
        swap(heap[cur], heap[2 * cur + 2]);
        cur = 2 * cur + 2;
      }
    }
  }
}
```

Heaps : Max or Min

How to get Max or Min element?

```
double get_max(const vector<double>& heap) {
  //return maximal element of the heap
}
```

Heaps : Max or Min

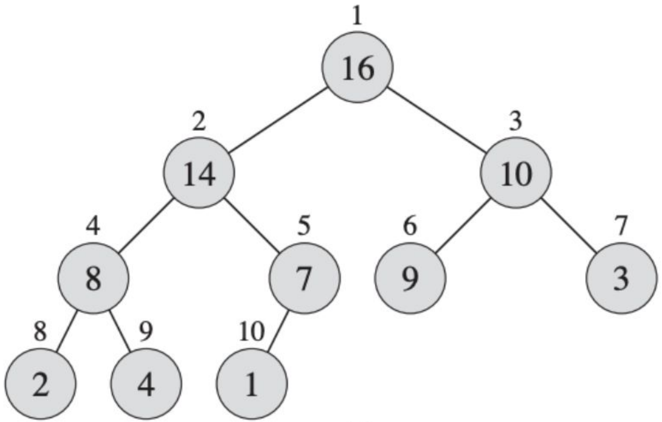
How to get Max or Min element?

```
double get_max(const vector<double>& heap) {
  return heap[0];
}
```

Heaps : HeapSort

Given a heap(that means it satisfies the heap property)
How to get the elements sorted?

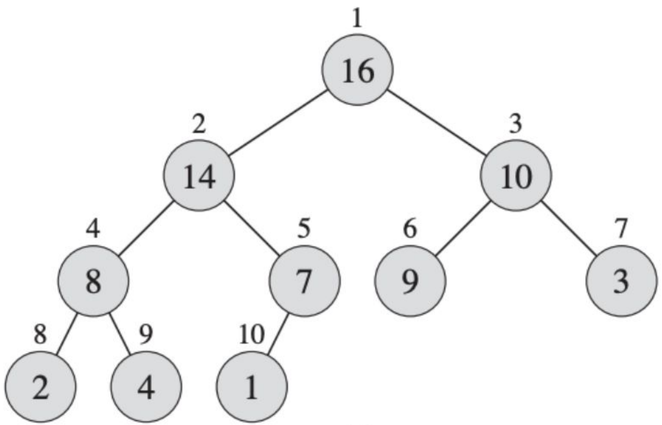
```
void heap_sort(vector<double>& heap) {
  //sort the heap
}
```



Heaps : HeapSort

Given a heap(that means it satisfies the heap property)
How to get the elements sorted?

```
void heap_sort(vector<double>& heap) {
  if (heap.size() <= 1) return;
  double val = heap[0]; //save the largest
  remove(heap); //remove the largest
  heap_sort(heap); //sort the rest
  heap.push_back(val); //add largest back
}
```



Heaps : Complexity

	Average	worst
Insertion:		
Deletion:		
Get_max for max heap:		
Heap_sort:		

Heaps : Complexity

	Average	worst
Insertion:	O(logN)	O(logN)
Deletion:	O(logN)	O(logN)
Get_max for max heap:	O(1)	O(1)
Heap_sort:	O(NlogN)	O(NlogN)

Priority Queue : STL

Priority Queue :

A linear data structure.

Looks like a queue, but totally different. (queue uses linked list, **priority_queue** uses **heap**).

For standard types, the priority is **larger** values (max heap), but like **set** and **map**, one can **overload** the **< operator** or define a priority comparator.

Like a heap, a priority_queue is not totally sorted. But **its top element is guaranteed** to have the **highest priority** among all elements. It **automatically adjust** the heap after each **pop** and **push**.

fx Member functions

(constructor)	Construct priority queue (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
top	Access top element (public member function)
push	Insert element (public member function)
emplace C++11	Construct and insert element (public member function)
pop	Remove top element (public member function)
swap C++11	Swap contents (public member function)

Priority Queue : Custom Comparator

```
struct LessThanByAge
{
  bool operator()(const Person& lhs, const Person& rhs) const
  {
    return lhs.age < rhs.age;
  }
};
```

then instantiate the queue like this:

```
std::priority_queue<Person, std::vector<Person>, LessThanByAge> pq;
```

If you just need min-priority queue instead of max

```
priority_queue <int, vector<int>, greater<int>> g
```

Priority Queue : Example

```
priority_queue<int> g1;
priority_queue<int, vector<int>, greater<int>> g2;
int b[5] = {3, 2, 6, 1, 8};
for (int i = 0; i < 5; ++i) {
  g1.push(b[i]);
  g2.push(b[i]);
}
while(!g1.empty()) {
  cout << g1.top() << endl;
  g1.pop();
}
while(!g2.empty()) {
  cout << g2.top() << endl;
  g2.pop();
}
```

Output:

Priority Queue : Example

```
priority_queue<int> g1;
priority_queue<int, vector<int>, greater<int>> g2;
int b[5] = {3, 2, 6, 1, 8};
for (int i = 0; i < 5; ++i) {
  g1.push(b[i]);
  g2.push(b[i]);
}
while(!g1.empty()) {
  cout << g1.top() << endl;
  g1.pop();
}
while(!g2.empty()) {
  cout << g2.top() << endl;
  g2.pop();
}
```

Output:

8

6

3

2

1

1

2

3

6

8

Priority Queue : Complexity

	Average	worst
push:	O(logN)	O(logN)
pop:	O(logN)	O(logN)
top:	O(1)	O(1)

Priority Queue : Sample Problem

How to use priority_queues(heaps) to keep track of the median of a data stream?

Priority Queue : Sample Problem

How to use priority_queues(heaps) to keep track of the median of a data stream?

Hint :

- Use 2 priority queues(PQ's)
- 1 for left half (max-priority)
- 1 for right half (min-priority)
- For every new element
  - push into left PQ.
  - get top element of left PQ, push it into right PQ.
  - pop top element of left PQ.

STL Data Structures : Summary

Unordered_set (Hash): sorted/unsorted**?.** insertion, deletion, look-up

Set (BST): sorted/unsorted**?** insertion, deletion, look-up.

Unordered_map (Hash): sorted/unsorted**?** insertion, deletion, look-up.

Map (BST): for mapping, sorted/unsorted?. insertion, deletion, look-up.

Priority_queue (heap): for knowing extreme values, sorted/unsorted?. knowing the max(min) from max(min) heap. insertion, deletion, look-up.

STL Data Structures : Summary

Unordered_set (Hash): fast for look-up, **unsorted**. O(1) for insertion, deletion, look-up.

Set (BST): for look-up, **sorted**. O(log N) for insertion, deletion, look-up.

Unordered_map (Hash): fast for mapping, **unsorted**. O(1) for insertion, deletion, map by key.

Map (BST): for mapping, **sorted**. O(log N) for insertion, deletion, map by key.

Priority_queue (heap): for knowing extreme values, **unsorted**. O(1) for knowing the max(min) from max(min) heap. O(log N) for insertion, deletion. O(N) for look-up.

References

Chapter 6 - HeapSort

- **Introduction to Algorithms, T.H Cormen**
- https://web.cs.ucla.edu/~srinath/static/pdfs/DataStructures&Algorithms_Cormen.pdf

Most content of the slides is taken from Yiyou Chen.

Thank You!!

Good luck for your finals!

Graphs

Graphs

A data structure to store key-value pairs.
Something like a dictionary.

Graphs

A data structure to store key-value pairs.
Something like a dictionary.

Graphs

A data structure to store key-value pairs.
Something like a dictionary.