CS 32 Week 10 Discussion 1B

🎉 🎊 last discussion of the quarter!! 🎊 🎉

this week's topics:

heap, priority_queue, final review!

TA: **Yiyou Chen** / LA: **Ian Galvez**

a couple things before we start…

- ***First, please find a partner / form a group!*** (Yes, I mean right now.) We'll be doing some practice problems.
- After that, please fill out the LA feedback form if you haven't already now that it's the end of the quarter: https://tinyurl.com/S22LAFeedback
  - You can also use the QR code aha >>
  - It really helps me improve as an LA, and improve our section! I want to provide resources that y'all need/want
- Be sure to select the ***Student to LA End of Quarter Feedback*** option!

Topics

Complete binary tree

Binary Heap

- insertion
- Deletion
- Get maximum(minimum) from max(min) heap
- Heap sort
- Time complexity

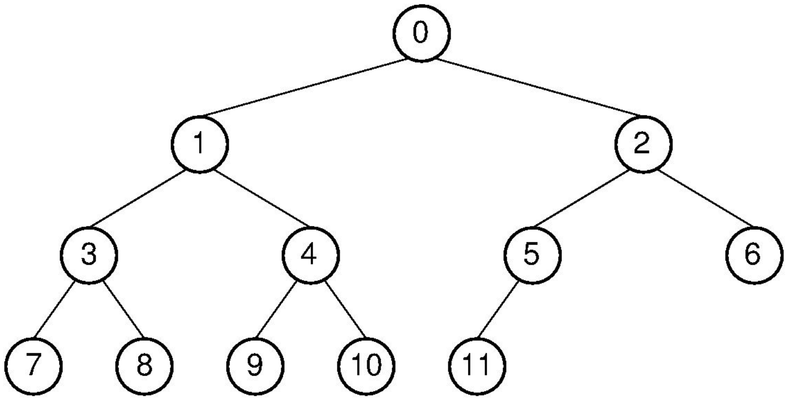
C++ STL priority queue

C++ STL summary

Final Review

Complete binary tree

A complete binary tree is a binary tree in which all the levels are **completely filled except possibly the lowest one,** which is filled **from the left**.

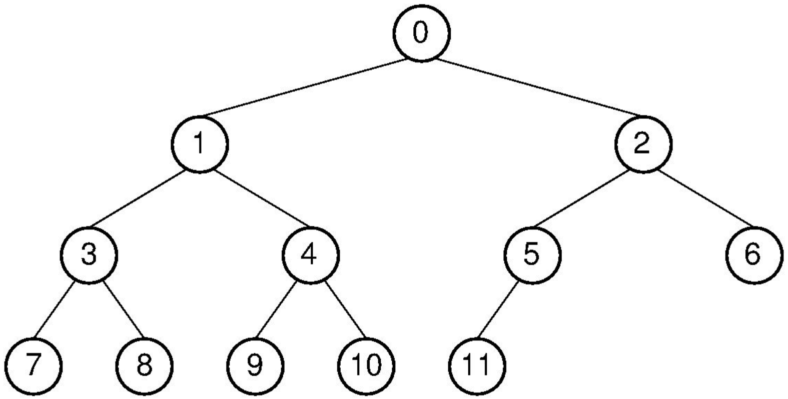


(a)

Image: https://opendsa-server.cs.vt.edu

Complete binary tree: array implementation

If the complete binary tree has N nodes with values saved in an array: a[N].
Then for all node i, its parent is ⌊(i-1)/2⌋. Its children are 2i+1 and 2i+2.
We also save the size of the array (or use a dynamically allocated array like vector).



(a)

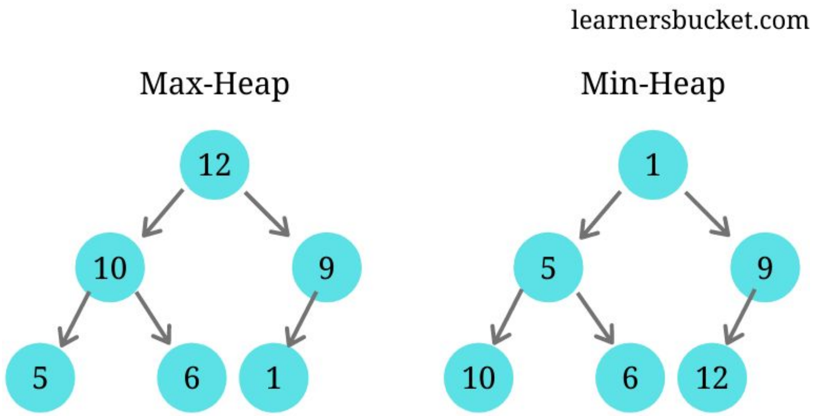
Image: https://opendsa-server.cs.vt.edu

Heap: definition

A max(min) Heap is a complete binary tree such that all parent nodes have values greater(less) than their children.

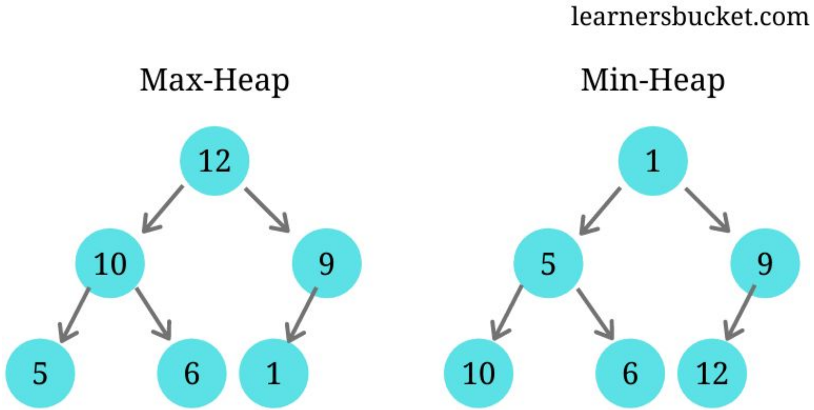
Therefore, the root of a max(min) heap has the greatest(smallest) value in the entire heap.

Heap is not totally sorted!!!



Heap: Definition

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



Heap: insertion

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

Heap: 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;
  }
}
```

Heap: deletion

```
void remove(vector<double>& heap) {
  //remove the root element of the heap
}
```

Heap: deletion

```
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;
      }
    }
  }
}
```

Heap: get maximum(minimum) of max(min) heap

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

Heap: get maximum (minimum)

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

Heap sort

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

Heap sort

```
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
}
```

Heap: 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)

STL: priority_queue (#include <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)

STL priority_queue: define priority 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;
```

One particular useful method is low priority first (min heap):

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

Source: stackoverflow

STL 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:

STL 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

STL priority_queue: complexity

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

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

C++ STL data structure 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.

review problems!

Exercise 1

AVL tree is a self-balancing BST where for each node, the heights of its left subtree and right subtree differ by at most 1. Please implement a function `isAVL()` to check if a BST is an AVL tree. You may use any helper functions.

```
struct Node{
  Node *left, *right;
  int val;
};
bool isAVL(Node* root) {


}
```

Exercise 2

Implement the destructor for a binary tree class defined as

```
class BT {
public:
  struct Node {
    Node *left, *right;
    vector<int*> v;
  };
  Node* root;
  ~BT();
};
```

```
BT::~BT() {

}
```

You may define and use helper functions.

Exercise 3

Consider the following array that has just been partitioned by the first step of the quicksort algorithm:

3 0 2 4 5 8 7 6 9

Could 5 be the initial pivot? What about 7? Explain why or why not.

Things I didn't cover but you probably want to know. Final review.

Ask me anything!

let’s take a break, then...

kahoot time!