Priority Queue

Featuring Binary Heap

Nattee Niparnan

Overview

- Simple Implementation of priority_queue
- Quick intro to Graph and Tree
- Binary Heap
- priority_queue with Binary Heap

priority_queue

- Queue by value
- Max-in-First-Out

```cpp
int main() {
  priority_queue<int> pq;
  pq.push(4);
  pq.push(20);
  pq.push(3);

  while (pq.empty() == false) {
    cout << pq.top() << endl;
    pq.pop();
  }
}
```

```
20
4
3
```

V0.1, priority_queue by vector

- Use vector to store data
- Push = simply push_back
- Top, pop = find the max value and return/erase
- max_element return iterator to max element

```
namespace CP {
  template <typename T>
  class priority_queue
  {
    protected:
      std::vector<T> v;
    public:
      bool empty()                       { return v.empty(); }
      bool size()                         { return v.size(); }

      void push(const T &e)                { v.push_back(e); }

      T top()     { return *std::max_element(v.begin(),v.end()); }

      void pop() {  v.erase(std::max_element(v.begin(),v.end()));}
  };
}
```

max_element

```
iterator max_element(iterator first, iterator last)
{
    if (first == last) return last;
    iterator largest = first;
    ++first;
    for (; first != last; ++first)
        if (*largest < *first)
            largest = first;
    return largest;
}
```

$O(n)$

V0.1 complexities

push

top()

pop()

V0.2 faster pop, top (and push??)

- v0.1 has many drawback
  - Consecutive call of top is slow (it shouldn't)
  - Both pop and top works almost the same
- v0.2 focus on slower push while keeps pop, top fast
- Make the vector sorted
  - Max will be at the back
  - Fast pop, top

```cpp
namespace CP {
  template <typename T>
  class priority_queue {
    protected:
      std::vector<T> v;
    public:
      bool empty() { return v.empty(); }
      size_t size() { return v.size(); }

      T& top()   { return v[v.size()-1]; }

      void pop() { v.erase(v.end()-1); }

      void push(const T& e) {
        // do something
      }
  };
}
```

v0.2 push

Complexity

$O(N \cdot log(N))$, where `N = std::distance(first, last)` comparisons on average.	(until C++11)
$O(N \cdot log(N))$, where `N = std::distance(first, last)` comparisons.	(since C++11)

- Maintain that the vector is sorted at every push

```
void push(const T& e) {
  v.push_back(e);
  std::sort(v.begin(), v.end());
}
```

O(n log n)

```
void push(const T& e) {
  auto it = v.begin();
  while (it < v.end() && *it <= e)
    it++;
  v.insert(it,e);
}
```

Θ(n)

Why Big-Theta, not Big O?

```
void push(const T& e) {
  v.insert(std::upper_bound(v.begin(),v.end(),e),e);
}
```

Upper bound is O(log n)

O(n)

Why cannot use Big-Theta?

Which one is better?

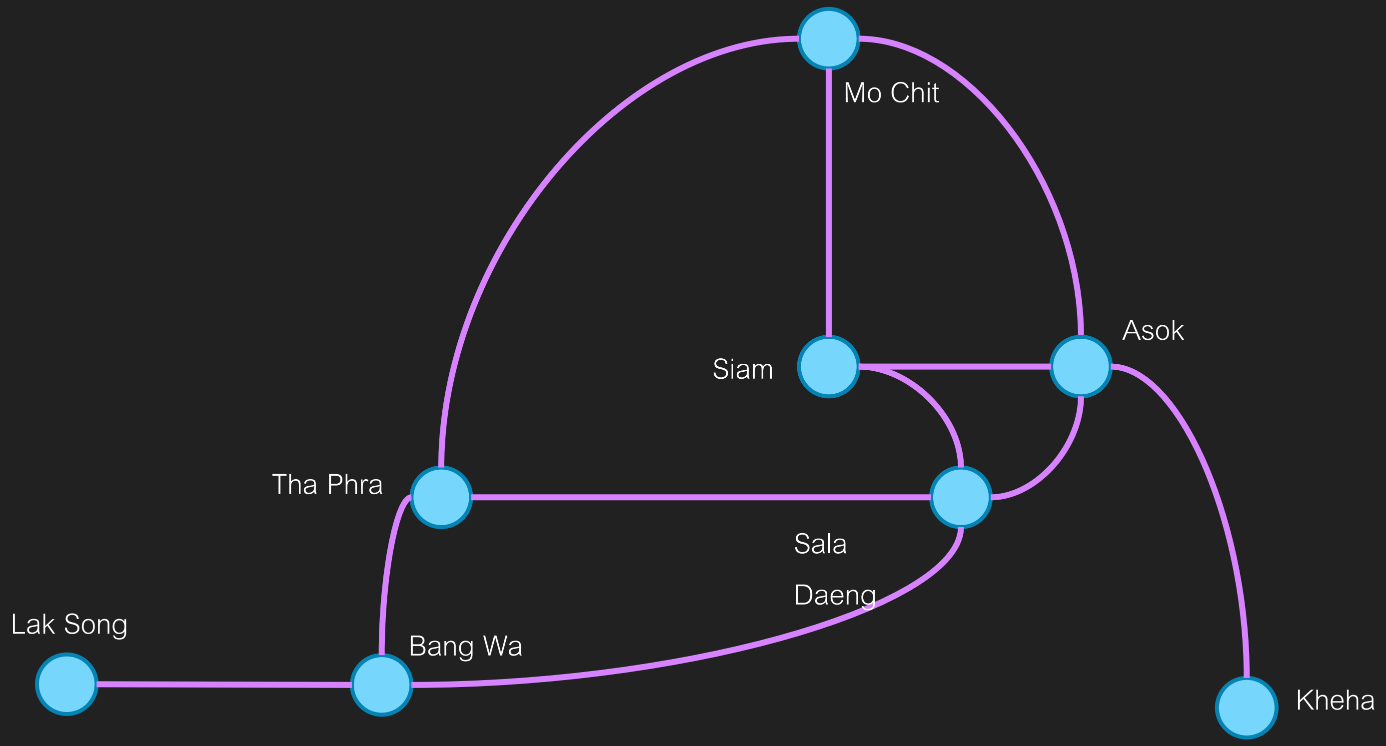
- v0.1 fast push
- v0.2 fast pop, top
- Depends on which operation we use most often
- The real version works like v0.2, we maintain some rules of the data that is stored in the vector such that
    - We know where max is (for fast top)
    - Much faster push, a little bit slower pop
    - Using structure call Binary Heap

Graph and Tree

Quick introduction

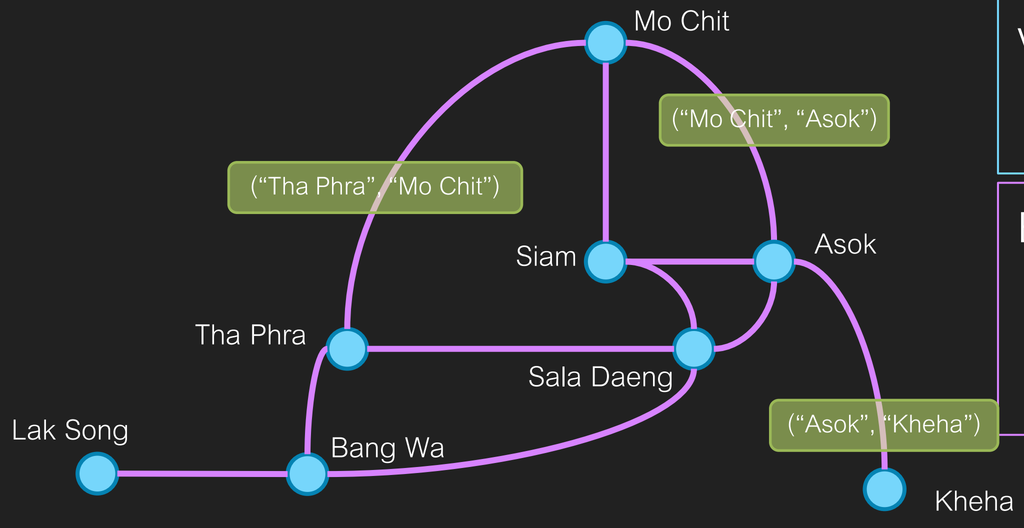
Graph

- Discrete Math Graph
- A math model that describe entities and connectivity between them



Graph Model

- Graph consists of two things
  - Nodes (vertex, vertices) are things we want to connect
  - Edges are pairs, each pair is a connectivity between two node
- Graph G = (V,E) where V is a set of nodes and E is a set of edges

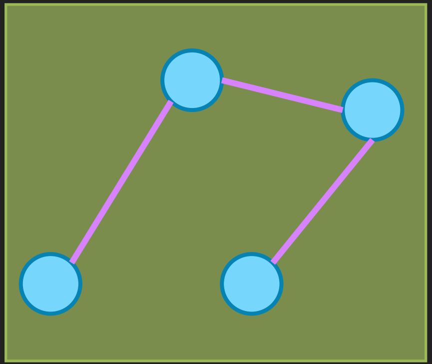


V = { "Mo Chit", "Siam", "Asok", "Sala Daeng", "Tha Phra", "Lak Song", "Bang Wa", "Kheha" }

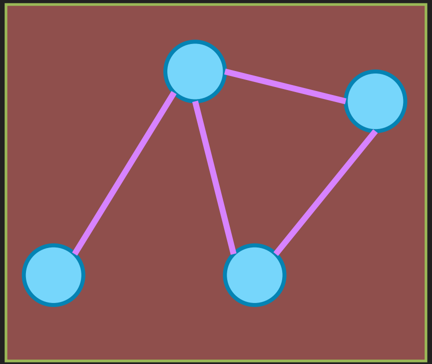
E = { ("Mo Chit", "Asok"), ("Mo Chit", "Siam"), ("Tha Phra", "Mo Chit"), ("Sala Daeng", "Tha Phra") … }

Tree

- A special kind of graph
  - Has N nodes and N-1 Edges
  - Every nodes must be connected (we can start from any node and can walk through edge to reach any node)

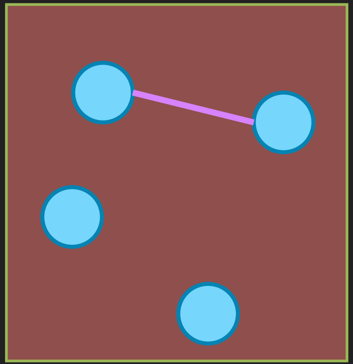


Is a tree



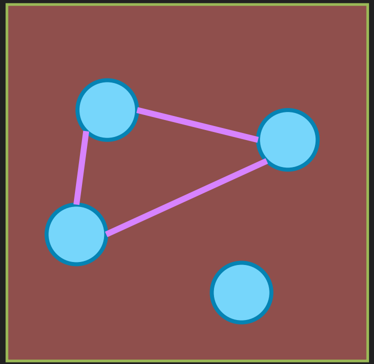
Not Tree

(too many edges)



Not Tree

(too many nodes)

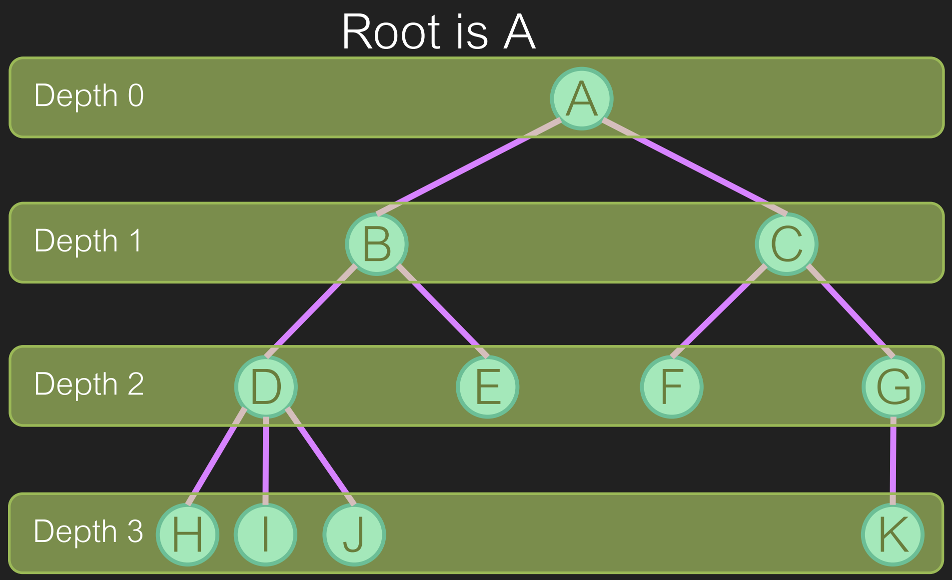


Not Tree

(not connected)

Rooted Tree

- Tree where one node is defined as a root
- For an edge in a rooted tree, a node that is closer to the root is called parent while the other node is called child
- Ancestor of node A = parent of parent .... of A
- Descendant of node A = child of child .... of A
- Root is usually drawn at the top and is consider as the starting point
    - Root is at level 0 (depth 0)
    - Children of root are drawn at the same level at level 1 (depth 1)
    - Children of children of root are at level 2 (depth 2)



A is the parent of B and C

B is the parent of D and E

K is a child of G

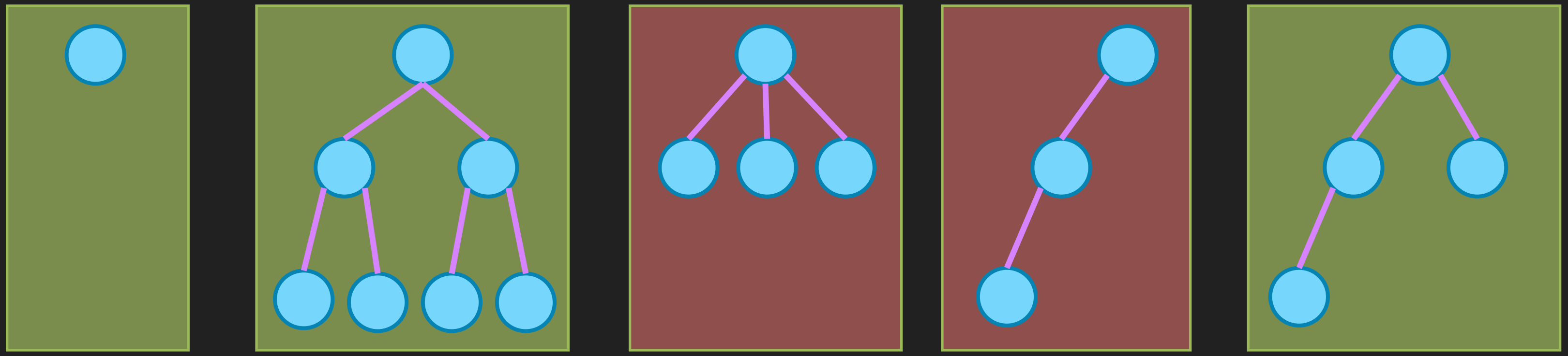
H, I and J are children of D

B is an ancestor of D, H, I and J

K is a descendant of G, C, and A

Complete Binary Tree

- Binary Tree = a tree that every node has at most 2 children
- Complete tree = the tree must be filled with every possible node at every level (except the deepest level which must be filled as far to the left as possible)
  - Blank tree is consider a complete binary tree



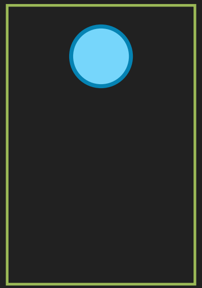
OK | OK | Not binary | Not complete (at depth 1) | OK

Exercise

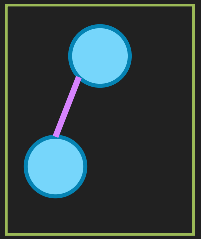
Hint: The answer is unique (There are exactly 1 way to draw a complete binary tree of size k)

- Draw a Complete Binary Tree that has 4, 5, 8, 10 nodes

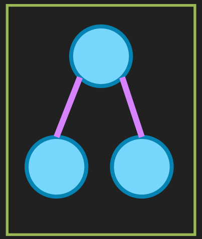
0 nodes
(blank tree)



1 nodes
(only root)



2 nodes



3 nodes

4 nodes

5 nodes

8 nodes

10 nodes

Special Property of a complete binary tree

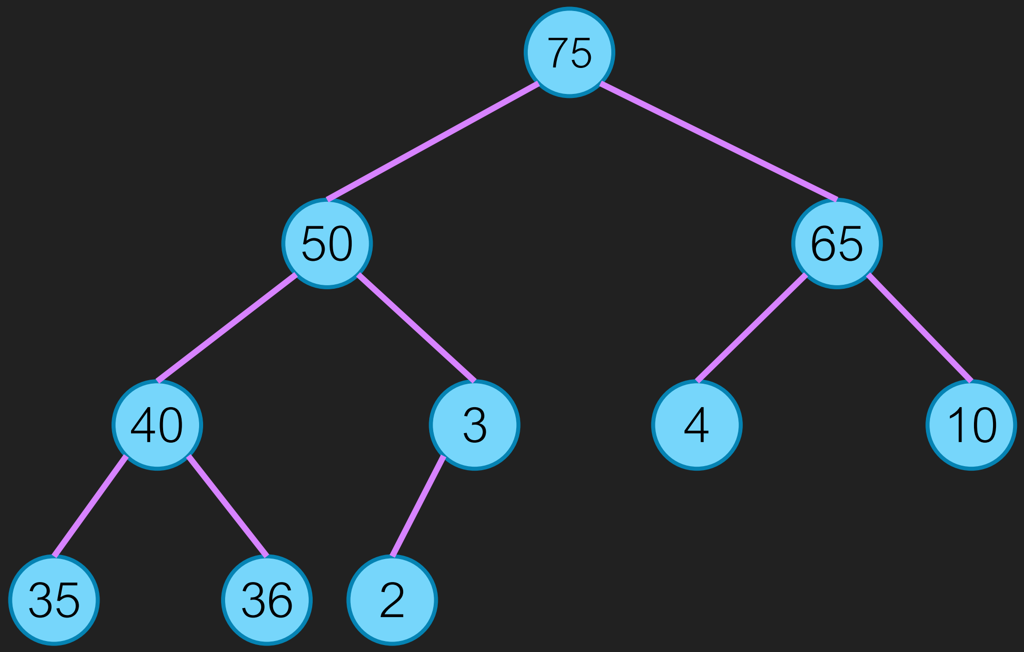
- There is exactly one way to go from any node to any node
- Maximum depth is $\log_2$ n where n is the number of nodes
  - Because we require completeness and we have 2 possible children

Binary Heap

Using Complete Binary Tree to make priority_queue

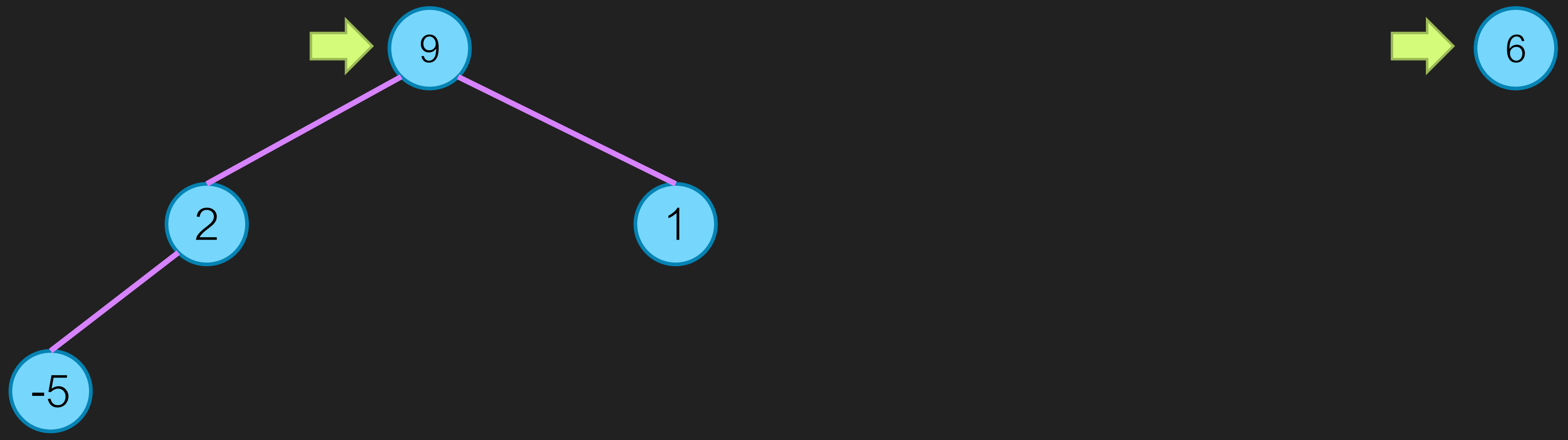
Binary Heap

- We use Complete Binary Tree to store data
  - A value is stored at the node
- When data is modified (via push or pop), we must maintain these rules
  1. Tree must always be Complete Binary Tree
  2. For any node, its value must be more than that of its children



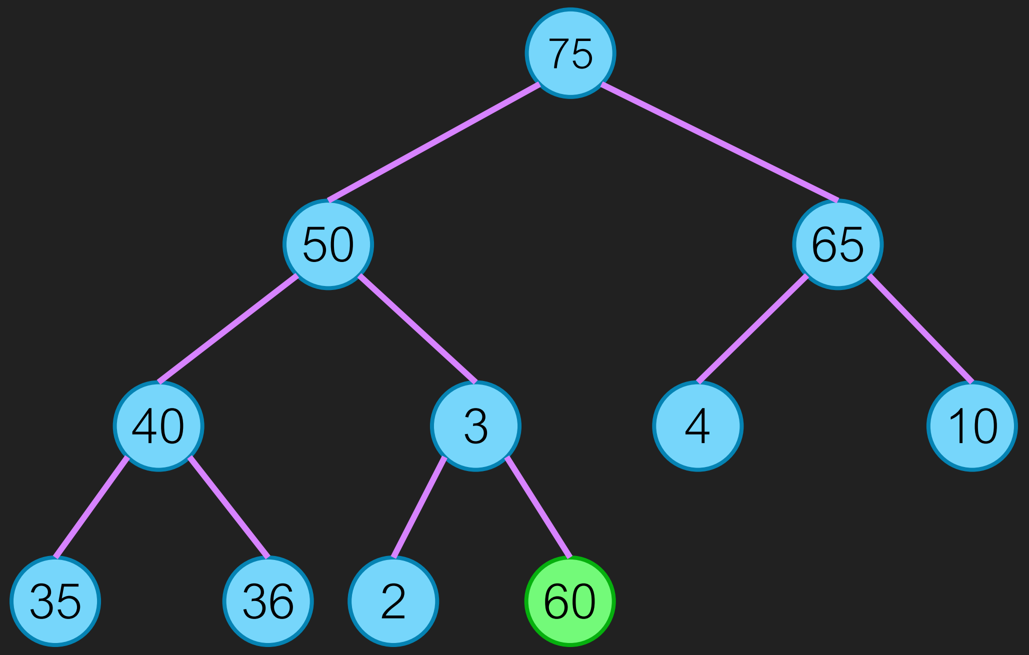
Getting Maximum Data

- Root contains highest value (because of rules 2.)
- Top() just simply return root



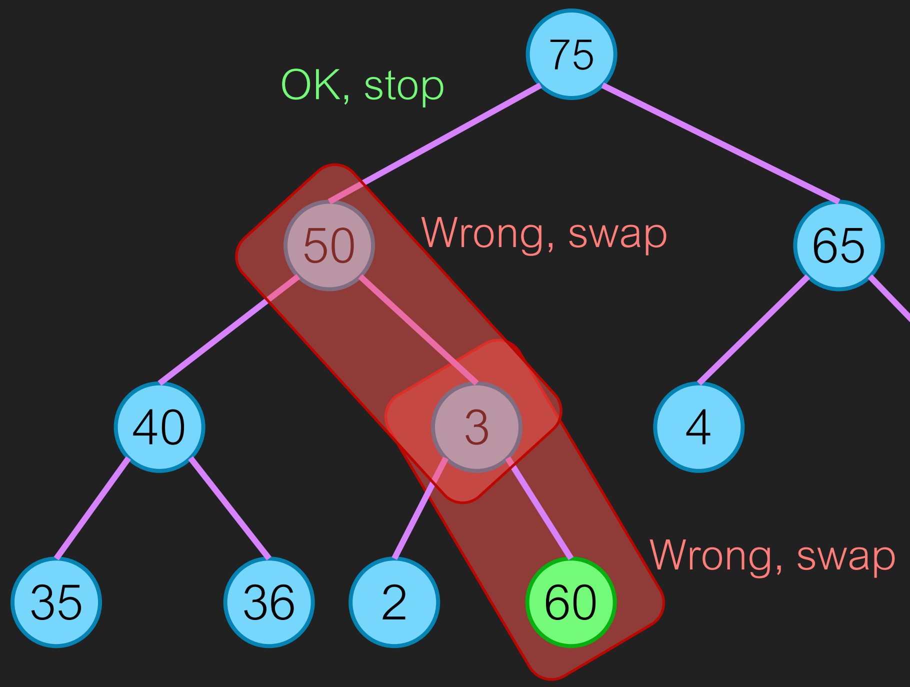
Adding data to Binary Heap

- Maintain Binary Heapness
    - structure of data
    - Value of data
- Structure rules says where the new node should be
    - Next to right-most child of the deepest level
    - But if we put the new data there, the value rules might be broken
        - Fix it



push(60)

Fix from adding a new node



Value rules: parent more than children

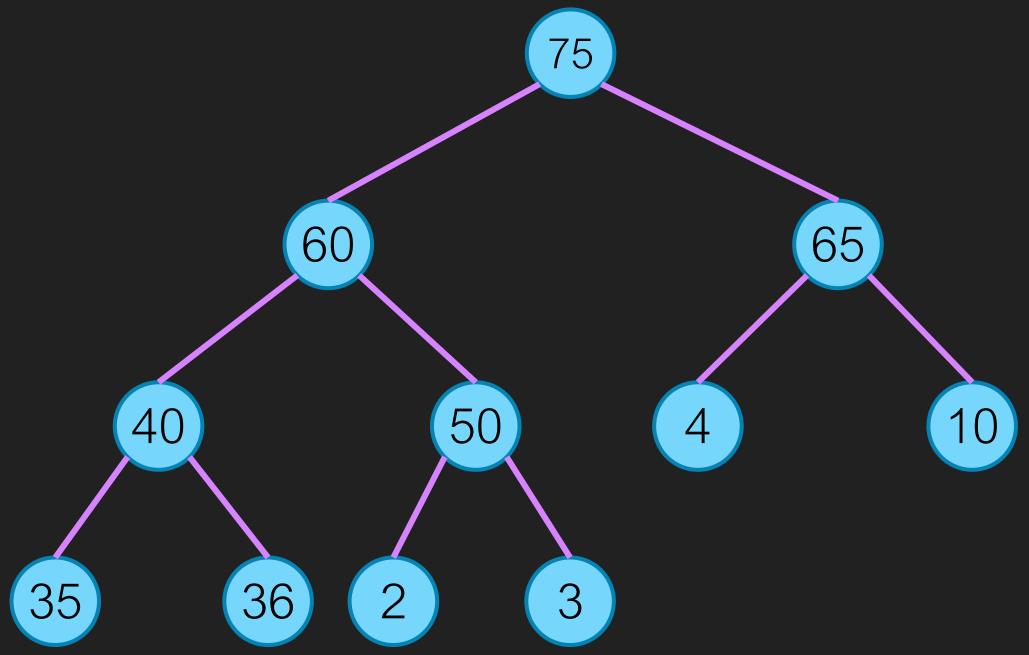
See that, after each swap, the blue swapped node does not violate value rules with its new children

Fix:

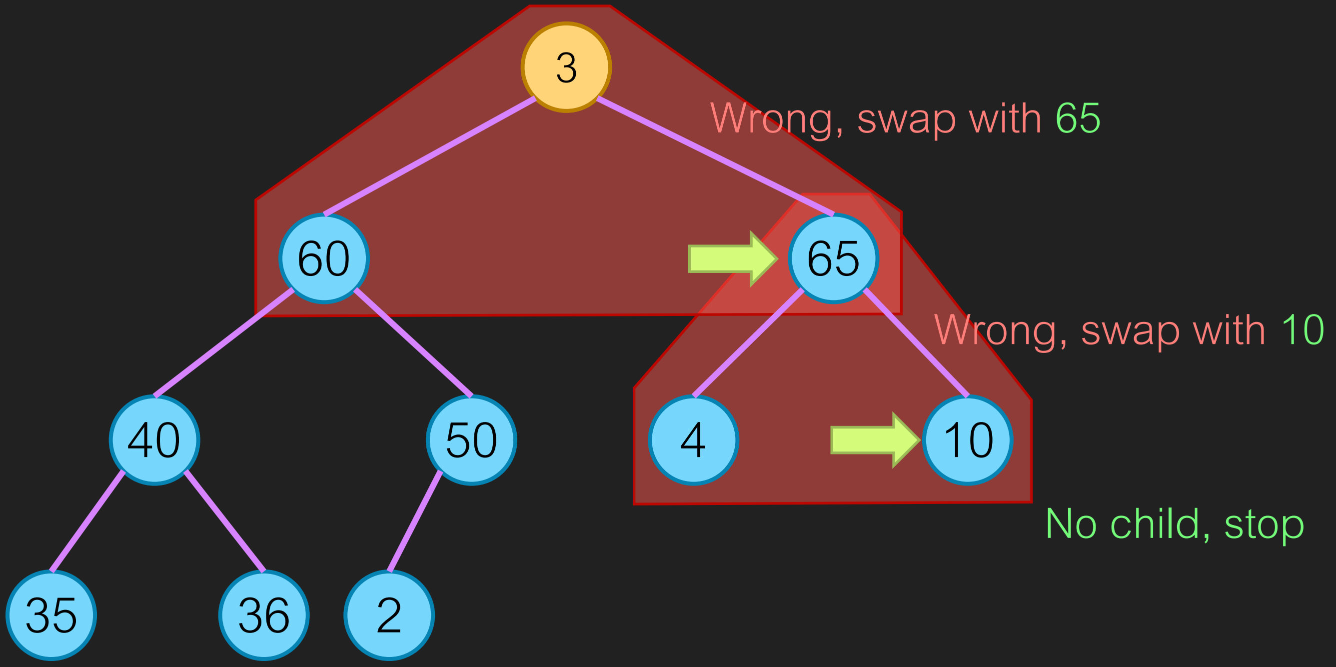
- Check where we just add a node, if value rules is broken, swap with parent
- After swap, re-check with new parent
- Keep doing until correct or at root

Delete maximum data

- Similar to push, we will try to maintain structure first
- Delete will remove root, find something to replace
  - Use the lowest, right-most node
- Value rules might be broken
  - Fix it



Fix from deleting root node



Fix:

- Start at replaced root, if value rules is broken, swap with maximum child
- After swap, re-check with new children
  - Beware! There is a case where we might have only one child
- Keep doing until correct or has no child

Value rules: parent more than children

See that, after each swap, the blue swapped node does not violate value rules with its parent and children

Analysis

- How fast is each push, pop
- Push
  - Add to a vector is O(1) amortized

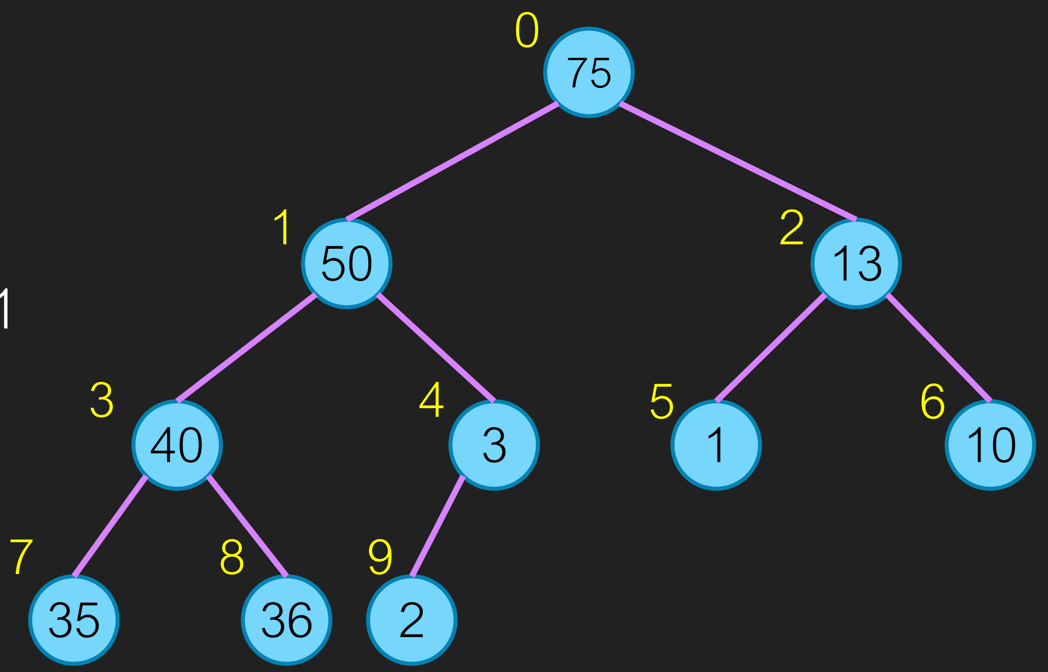
    O(log n)
  - Fixing value rules is O(h) where h is the maximum number of depth of the tree (we call this value tree height)
  - Notice that tree height is O(lg n)
- Pop
  - Fixing value rules is O(h) where h is the maximum number of depth of the tree (we call this value tree height)

    O(log n)

CP::priority_queue

How to store a tree?

- Use dynamic array
  - Each node can be labelled from 0 to n-1
- Root is at 0
- Left child of node i is at (i*2)+1
- Right child of node i is at (i*2)+2
- Parent of node i is at (i-1)/2



0	1	2	3	4	5	6	7	8	9
75	50	13	40	3	1	10	35	36	2

Layout

```
template <typename T,typename Comp = std::less<T> >
class priority_queue {
  protected:
    T *mData;
    size_t mCap;
    size_t mSize;
    Comp mLess;
    void expand(size_t capacity) {}
    void fixUp(size_t idx) {}
    void fixDown(size_t idx) {}
  public:
    //-------------- constructor ---------------
    priority_queue(priority_queue<T,Comp>& a);
    priority_queue(const Comp& c = Comp() );
    priority_queue<T,Comp>& operator=(priority_queue<T,Comp> other);
    ~priority_queue();
    //------------- capacity function ----------
    bool empty() const;
    size_t size() const;
    //----------------- access -----------------
    const T& top();
    //----------------- modifier ---------------
    void push(const T& element);
    void pop();
};
```

Same as CP::vector

Will talk about later

Fix value rules

Constructor

```cpp
priority_queue(const Comp& c = Comp() ) :
  mData( new T[1]() ),
  mCap( 1 ),
  mSize( 0 ),
  mLess( c )
{ }
```

```cpp
priority_queue(priority_queue<T,Comp>& a) :
  mData(new T[a.mCap]()),
  mCap(a.mCap),
  mSize(a.mSize),
  mLess(a.mLess)
{
  for (size_t i = 0; i < a.mCap;i++)
    mData[i] = a.mData[i];
}
```

- Using list initialize
- See that mData is dynamic array in the same way as vector
- mLess is something that is just either copied or default initialize
  - Will talk about it later

Destructor and Copy Assignment Operator

```
    ~priority_queue() {
      delete [] mData;
    }
```

- Using standard copy-and-swap idiom

```
priority_queue<T,Comp>& operator=(priority_queue<T,Comp> other) {
  using std::swap;
  swap(mSize,other.mSize);
  swap(mCap,other.mCap);
  swap(mData,other.mData);
  swap(mLess,other.mLess);
  return *this;
}
```

Push

```
void expand(size_t capacity) {
  T *arr = new T[capacity]();
  for (size_t i = 0;i < mSize;i++) {
    arr[i] = mData[i];
  }
  delete [] mData;
  mData = arr;
  mCap = capacity;
}
```

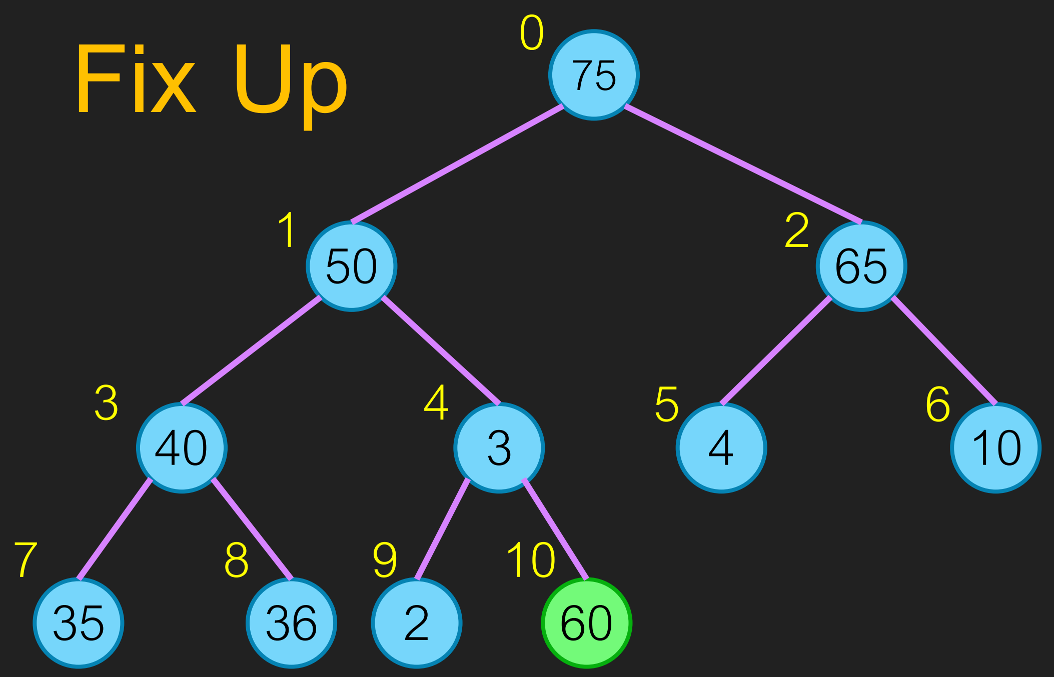
Same as

CP::vector

```
void push(const T& element) {
  if (mSize + 1 > mCap)
    expand(mCap * 2);
  mData[mSize] = element;
  mSize++;
  fixUp(mSize-1);
}
```

- See that the right-most child of the deepest level is at mData[mSize-1] and the new node should be at mData[mSize]
- We do the same thing as vector's push_back
- Then fix the value rule

Fix Up

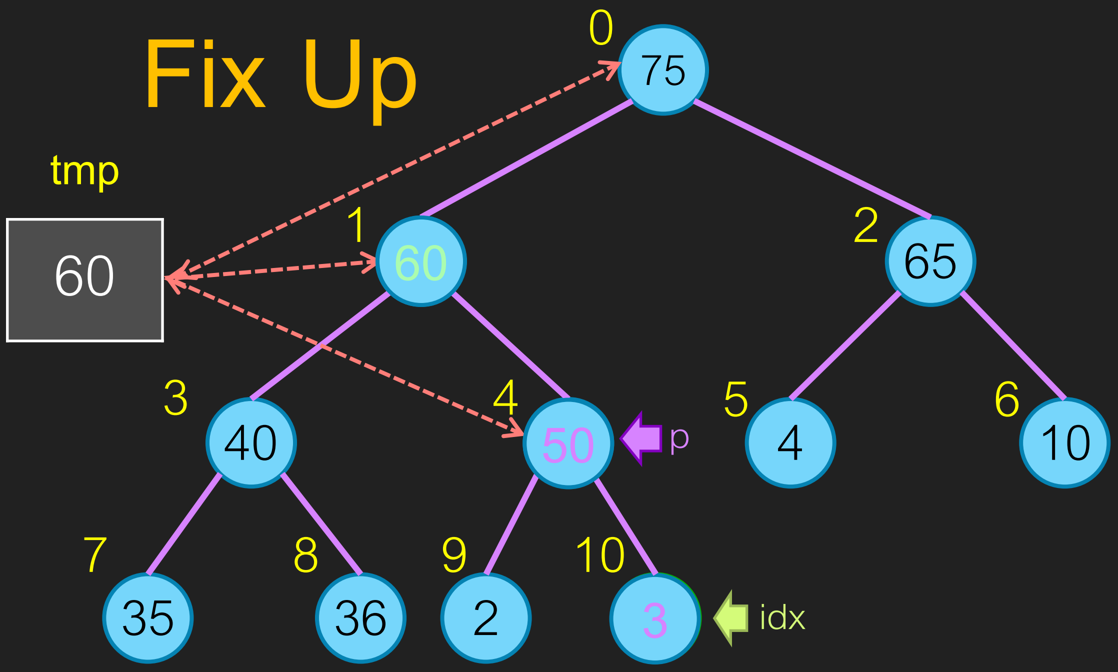


- Instead of actual swap, we perform insert and find appropriate position at the same time

```
void fixUp(size_t idx) {
  T tmp = mData[idx];
  while (idx > 0) {
    size_t p = (idx - 1) / 2;
    if ( tmp < mData[p] ) break;
    mData[idx] = mData[p];
    idx = p;
  }
  mData[idx] = tmp;
}
```

```
void fixUp(size_t idx) {
  while (idx > 0) {
    size_t p = (idx - 1) / 2;
    if ( mData[idx] < mData[p] ) break;
    T tmp = mData[p];
    mData[p] = mData[idx];
    mData[idx] = tmp;
    idx = p;
  }
}
```

Fix Up




```
void fixUp(size_t idx) {
  T tmp = mData[idx];
  while (idx > 0) {
    size_t p = (idx - 1) / 2;
    if (  tmp < mData[p] ) break;
    mData[idx] = mData[p];
    idx = p;
  }
  mData[idx] = tmp;
}
```

mData

0	1	2	3	4	5	6	7	8	9
75	60	13	40	50	1	10	35	36	2

mLess

- priority_queue allows a custom comparator
- Custom comparator X
  - We must be able to X(a,b) where X will compare a and b and return true only when a is less than b
  - X is a variable that implement operator()

```
int main() {
  less<int> x;
  greater<int> y;

  int a = 10;
  int b = 3;
  cout << x(a,b) << endl;
  cout << y(a,b) << endl;
}
```

mLess

- Initialize at constructor as variable mLess to be of type Comp in template
- Any comparison of our data (type T) must be done by mLess

```
template <typename T,typename Comp = std::less<T> >
class priority_queue {
  //...
  T *mData;
  size_t mCap;
  size_t mSize;
  Comp mLess;
  //...
  priority_queue(const Comp& c = Comp() ) :
    mData( new T[1]() ),
    mCap( 1 ),
    mSize( 0 ),
    mLess( c )
  { }
```

```
void fixUp(size_t idx) {
  T tmp = mData[idx];
  while (idx > 0) {
    size_t p = (idx - 1) / 2;
    if ( mLess(tmp,mData[p]) ) break;
    mData[idx] = mData[p];
    idx = p;
  }
  mData[idx] = tmp;
}
```

Pop

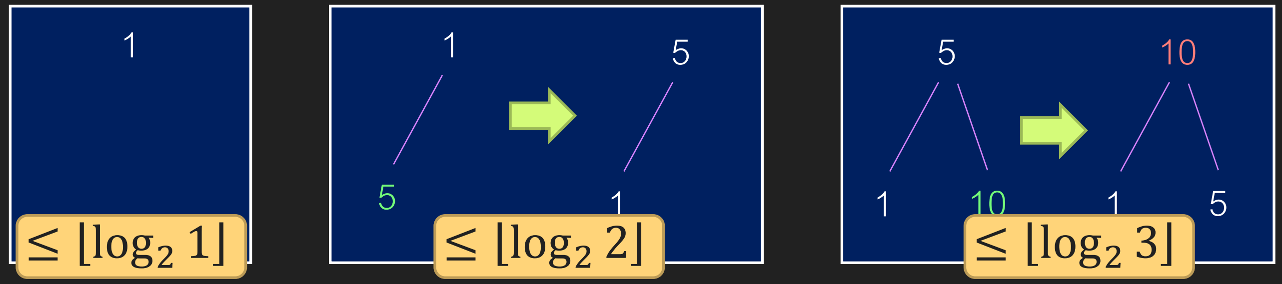
```
void pop() {
  mData[0] = mData[mSize-1];
  mSize--;
  fixDown(0);
}
```

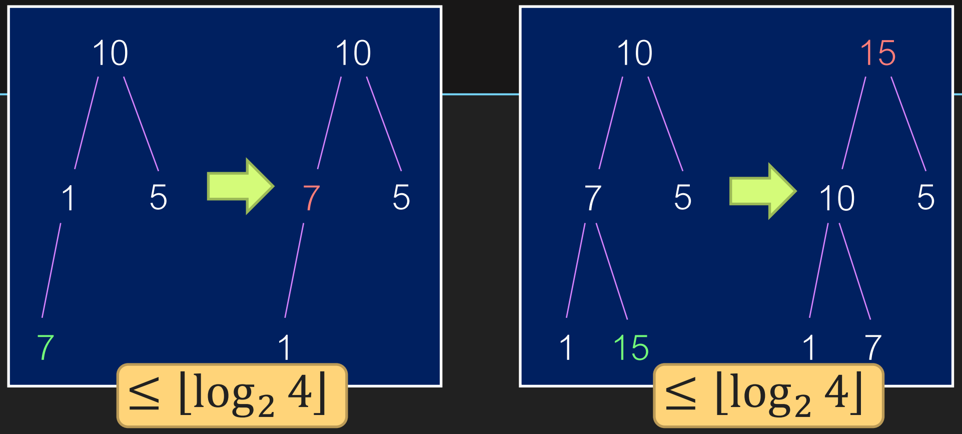
```
void fixDown(size_t idx) {
  T tmp = mData[idx];
  size_t c;
  while ((c = 2 * idx + 1) < mSize) {
    if (c + 1 < mSize && mLess(mData[c],mData[c + 1]) ) c++;
    if ( mLess(mData[c],tmp) ) break;
    mData[idx] = mData[c];
    idx = c;
  }
  mData[idx] = tmp;
}
```

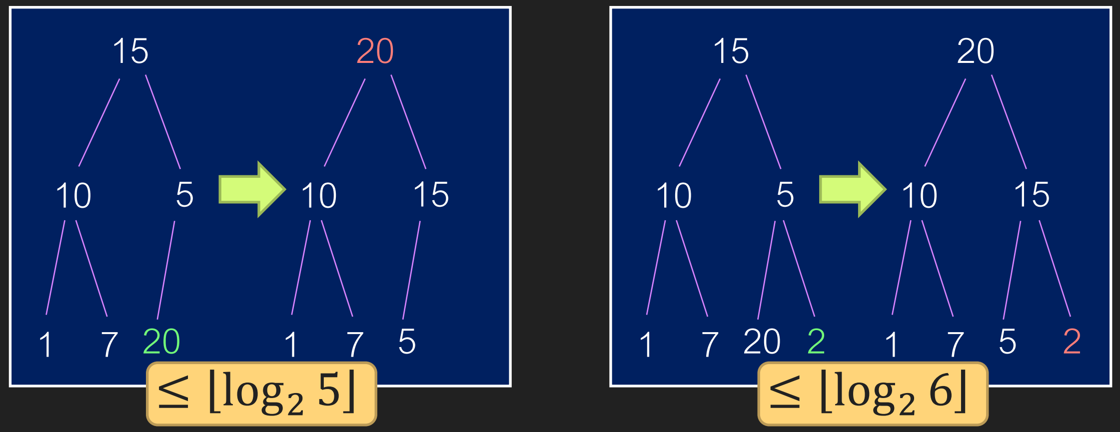
- While loop check if we have at least one child
- c is the index of highest value child
  - Must consider the case where we have only one child
- Exercise: read the rest yourself

Construct PQ from n data

```
priority_queue(std::vector<T> &v,const Comp& c = Comp() )  :
    mData( new T[v.size()]() ), mCap( v.size() ), mSize( 0 ), mLess( c ) {
  for (size_t i = 0;i < mSize;i++) push(v[i]);
}
```







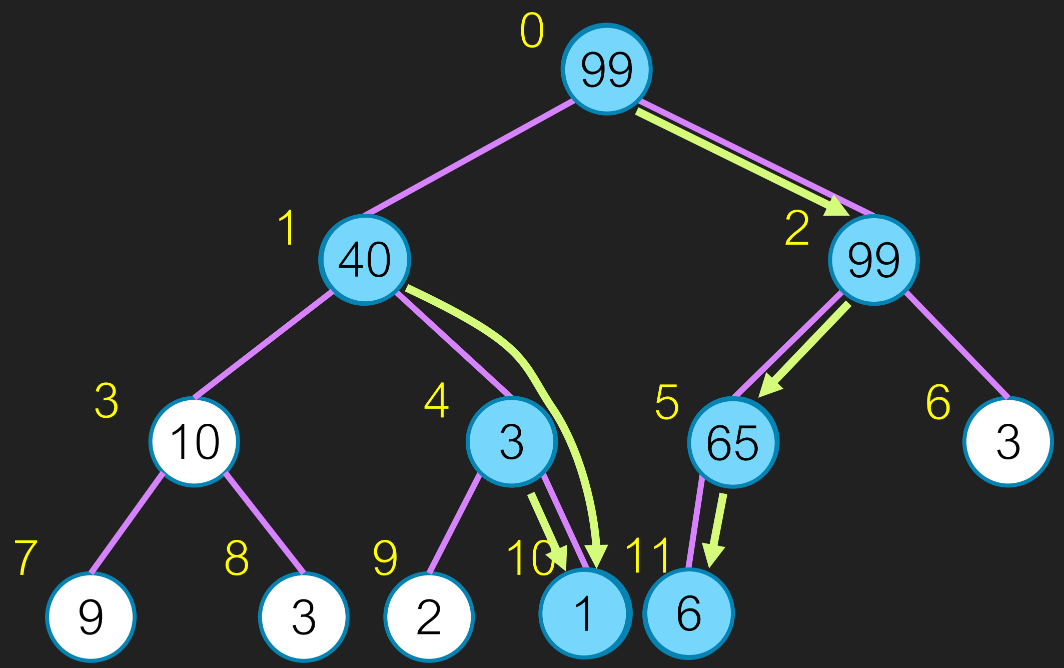
$$\text{Total} \le \lfloor \log_2 1 \rfloor + \lfloor \log_2 1 \rfloor + \cdots + \lfloor \log_2 n \rfloor$$

$$\le \lfloor \log_2 (1 \times 2 \times 3 \times \cdots \times n) \rfloor = \lfloor \log_2 n! \rfloor$$

$\lfloor \log_2 n! \rfloor$ is O(n log n)

Better Method

```
priority_queue(std::vector<T> &v,const Comp& c = Comp() )  :
    mData( new T[v.size()]() ), mCap( v.size() ), mSize( v.size() ), mLess( c )
{
  for (size_t i = 0;i < mSize;i++)  mData[i] = v[i];
  for (int i = mSize/2-1;i >= 0;i--) fixDown(i);
}
```

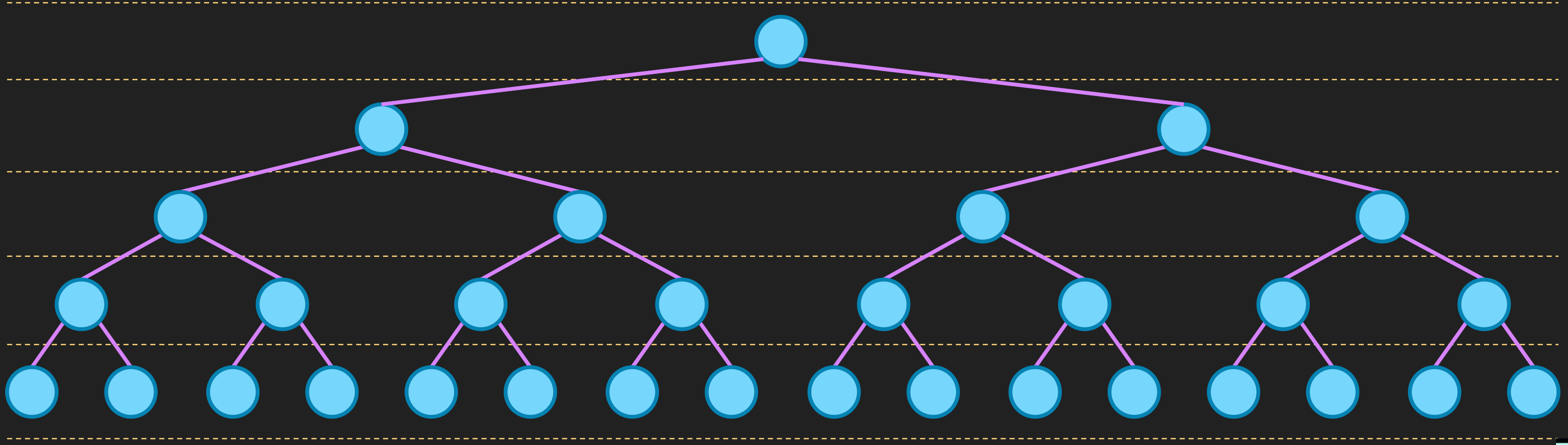


- Consider each node to be a Binary Heap
- Fix down from back to front

How fast?

Total node = 31

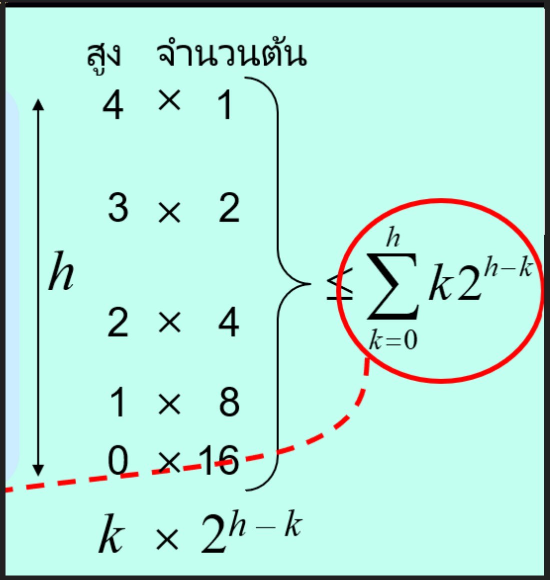
Tree height = $\log_2 31 = 4$



Depth	nodes	Max fix per node
0	1	4
1	2	3
2	4	2
3	8	1
4	16	0

Binary Tree Property:

- There are at most 2^k nodes at depth k
- For a three of height h, at depth k, fix down need at most h-k iterations



$$= 2^h \sum_{k=0}^{h} k2^{-k} < 2^h \boxed{\sum_{k=0}^{\infty} k2^{-k}} = 2^h 2 = 2^{h+1}$$

This is 2

$$= 2^{\log_2 n+1} = O(n)$$