Priority Queue

Queue with privilege

Nattee Niparnan

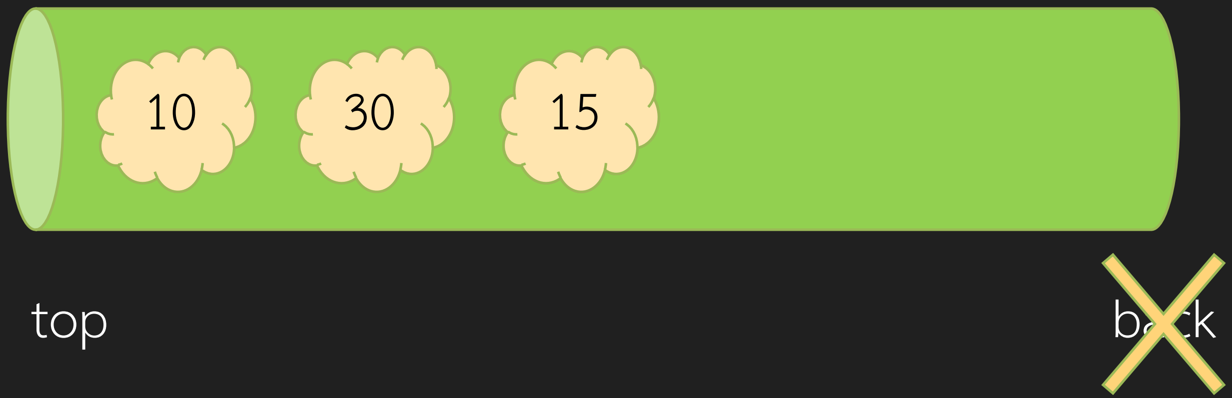
Intro

- Priority Queue is....
  - A queue with priority
  - Item with high priority is promoted to the front of the queue
    - There is no back of the queue
  - Priority is defined by having more value
    - Comparison, by default, is to use operator <, i.e., if item A < B is true, then B has higher priority
    - We can have custom comparator
- Has the same interface as stack

Example

For intuitive purpose only!
While the result is correct,
This is not really how
priority_queue work internally.

push(10)
push(30)
push(20)
pop()
push(15)
pop()



Basic

```
#include <queue>
#include <iostream>

using namespace std;

int main() {
    priority_queue<int> pq;
    pq.push(10);
    pq.push(30);
    pq.push(20);
    cout << "Current size = " << pq.size() << " top is " << pq.top() << endl;
    pq.pop();
    pq.push(15);
    pq.pop();
    cout << "Current size = " << pq.size() << " top is " << pq.top() << endl;
}
```

```
size_t      q.size()
bool        q.empty()
void        q.push(T data)
void        q.pop()
T           q.top()
```

Limitation

- Same limitation as stack, queue
  - No iterator
    - No begin(), end()
    - Can only access top of the priority_queue
    - If we wish to access all members, we have to pop it all
  - Do not call top(), pop() when the priority_queue is empty
- The data type must be comparable (similar to set and map)

Class in C++

I hope you have read the assignment

http://www.cplusplus.com/doc/tutorial/classes/

Quick Summary

- Syntax
  - class declaration must end with ;
  - Function definition can be outside the class
  - Access modifier is public:, private: protected:
  - constructor is a function with the same name of the class with no return type
- Object is a variable (instantiation) of a class
  - When declared, a constructor is called

Example 1

```cpp
#include <iostream>
#include <string>
using namespace std;

class Student {
public:
  void setFullname(string name,string surname) {
      this->name = name;
      this->surname = surname;
  }
  string getFullname() {
      return "[" + name + " " + surname + "]";
  }
private:
  string name,surname;
};

int main() {
    Student a;
    Student b;
    a.setFullname("nattee", "niparnan");
    cout << a.getFullname() << endl;
    cout << b.getFullname() << endl;
}
```

```cpp
#include <iostream>
#include <string>
using namespace std;

class Student {
public:
  void setFullname(string name,string surname);
  string getFullname();
private:
  string name,surname;
};

void Student::setFullname(string name,string surname) {
  this->name = name;
  this->surname = surname;
}

string Student::getFullname() {
  return "[" + name + " " + surname + "]";
}

int main() {
    Student a;
    Student b;
    a.setFullname("nattee", "niparnan");
    cout << a.getFullname() << endl;
    cout << b.getFullname() << endl;
}
```

Example 2: Constructor

```cpp
#include <iostream>
#include <string>
using namespace std;

class Student {
public:
  Student(float score) {   gpax = score;   }
  void setFullname(string name,string surname) {
      this->name = name;
      this->surname = surname;
  }
  string getFullname() { return "[" + name + " " + surname + "]"; }
  bool is1stHonor() { return gpax >= 3.6; }
private:
  string name,surname;
  float gpax;
};

int main() {
    Student a(2.95);
    a.setFullname("nattee", "niparnan");
    cout << a.getFullname() << endl;
    if (a.is1stHonor()) { cout << "YES" << endl; } else { cout << "NO" << endl;}
    // Student b; // <-- cannot compile because there is no default constructor
}
```

```
[nattee niparnan]
NO
```

Operator Overloading

How C++ has a function for each operator

Overview

- Let say we write a + b when a and b is an object of some classes.
  - This can be considered the same as calling a function plus(a,b)
  - C++ allow us to write a function for many operator and use it as an operator
  - For example we can write a function times(a,b) and let it be used as a * b
- This is call operator overloading

Example

```cpp
#include <queue>
#include <iostream>
#include <string>

using namespace std;

string operator*(string & lhs, const int & rhs) {
    string result = "";
    for (int i = 0;i < rhs;i++) {
        result = result + lhs;
    }
    return result;
}

int main() {
    string a = "abc ";
    cout << a * 3 << endl;
    //this gives "abc abc abc "
}
```

- Function must be named operator followed by the operator that we will overload
- Some operator takes two parameters (such as +, -, *, /, %)
- Some takes on (such as ++, --, !, *, &)

Using with data structure that require sorting

- We have seen several data structure that requires comparability of the data, such as set, map and priority queue
- If we want to use our class with these data structure, we need to tell them how can we compare a pair of them
- There are multiple ways to achieve this
  - Let us consider operator overloading

Overloading <

- As stated earlier, set, map and priority_queue use operator< to compare two elements
- It does not work if we overload operator>

```
1
attawith
```

```
class Student {
public:
  Student(float score, string a, string b) {
    name = a;
    surname = b;
    gpax = score;
  }
  bool is1stHonor() { return gpax >= 3.6; }
  //not good, now our data is public
  string name,surname;
  float gpax;
  //overloading <
  bool operator<(const Student& other) const {
    return gpax < other.gpax;
  }
};

int main() {
  Student a(2.95,"nattee","niparnan");
  Student b(4.00,"attawith","sudsang");
  cout << (a < b) << endl;
  priority_queue<Student> pq;
  pq.push(a);
  pq.push(b);
  cout << pq.top().name << endl;
}
```

Custom Comparator

Why custom?

- By overloading operator<, we have defined default ordering of that class
- What if we need another ordering, just for this priority_queue only
  - For example, Student is ordered by gpax by default
  - What if we want our priority_queue to order by name instead, while keep the Student default ordering elsewhere
  - Better, can we have multiple priority_queue with different ordering?
- Can be done via comparator class

Example

```cpp
#include <iostream>
#include <string>
#include <queue>
using namespace std;

class Student {//same as before};

class StudentByNameComparator {
public:
  bool operator()(const Student& lhs,
                  const Student& rhs) {
      return lhs.name < rhs.name;
  }
};

class GpaxThenName {
public:
  bool operator()(const Student& lhs,
                  const Student& rhs) {
      if (lhs.gpax == rhs.gpax)
        return lhs.name < rhs.name;
      return lhs.gpax < rhs.gpax;
  }
};
```

```cpp
int main() {
  Student a(2.95,"nattee","niparnan");
  Student b(4.00,"attawith","sudsang");
  Student c(4.00,"vishnu","kotrajaras");
  cout << (a < b) << endl;
  StudentByNameComparator comp1;
  GpaxThenName comp2;
  priority_queue<Student,
                 vector<Student>,
                 StudentByNameComparator> pq(comp1);
  pq.push(a);
  pq.push(b);
  cout << pq.top().name << endl;

  priority_queue<Student,
                 vector<Student>,
                 GpaxThenName> pq2(comp2);
  pq2.push(a);
  pq2.push(b);
  pq2.push(c);
  cout << pq2.top().name << endl;

}
```

```
1
nattee
vishnu
```

Another Method, lambda-function

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;

int main() {
  auto compare = [](const string& lhs, const string& rhs) {
    return lhs.size() < rhs.size();
  };

  cout << "Result of compare function = " << compare("xxx","z") << endl;

  priority_queue<string,vector<string>,decltype(compare)> pq(compare);
  pq.push("somchai");
  pq.push("z");
  pq.push("abc");
  while (pq.empty() == false) {
    cout << pq.top() << endl;
    pq.pop();
  }

}
```

```
somchai
abc
z
```

- Compare is a variable of function type
- This one orders by length of string

Templating of priority_queue

- priority_queue requires 3 template parameters
- `priority_queue<T, Container = vector<T>, Compare = less<T>>`
- The first one is required (which is the type of the data
- The second and the third is optional (it has default type)
  - Second is the container (for now, just don’t think about it)
  - Third is the class for comparator (the class that we use to compare)
    - This one is default to `less<T>`

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;

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

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

```
0
1
```

Using Comparator for set and map

- To use custom class with set and map, we need to do the same thing, let set and map know how to sort the data
    - Either make default ordering (overload<) in the custom class
    - Or use custom comparator when declare
- For set, the declaration is `set<T, Compare = less<T>>`
- For map, the declaration is `map<Key, T, Compare = less<Key>>`

Assignment

- Is any of vector<int>, set<int>, map<int,string>, queue<bool>, stack<vector<int>> comparable?
  - For any class that is "YES", how it is ordered?
  - For example, if vector<int> is comparable, how {1,2,3} is compared to {1,2,3,4} or {2,3,4}

Short Summary

Data Structure Summary

Data Structure	Pro	Cons	Remark
pair<T1,T2>	Nothings… It just a pair of two data type		
vector<T>	• Fast access [ ] • Fast append	• Slow find • Slow insert, Slow Erase	
set<T>	• Fast find • Item is sorted	• Slower to just append data than vector, stack, queue • Iterate is also slow • Takes lots of memory	Require comparator
map<Key,T>			• Associative data type • Also require comparator of Key_Type
stack<T>	• Very fast push, pop	• Very limited functionality but has special uses	• No iterator • Order of data coming out depends on something (stack, queue depends on WHEN it is pushed, pq depends on value) • PQ requires comparator
queue<T>			
priority_queue <T>	• Fast get max • Fast delete max • Data is sorted • Memory efficient	• Slower to just append data than vector, stack, queue • Very limited functionality	

more data structure

- C++ has more data structure not really covered right now
  - list is a vector with faster insert / erase but does not have fast access
  - unordered_set, unordered_map are set and map that the data is not sorted but is much faster
  - deque (pronounced DECK) is a queue that can push, pop at both ends
  - multiset, multimap are set and map that allows duplicate entries

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

```
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)

```
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 Big-Theta, not Big O?

Why cannot use Big-Theta?

Which one is better?

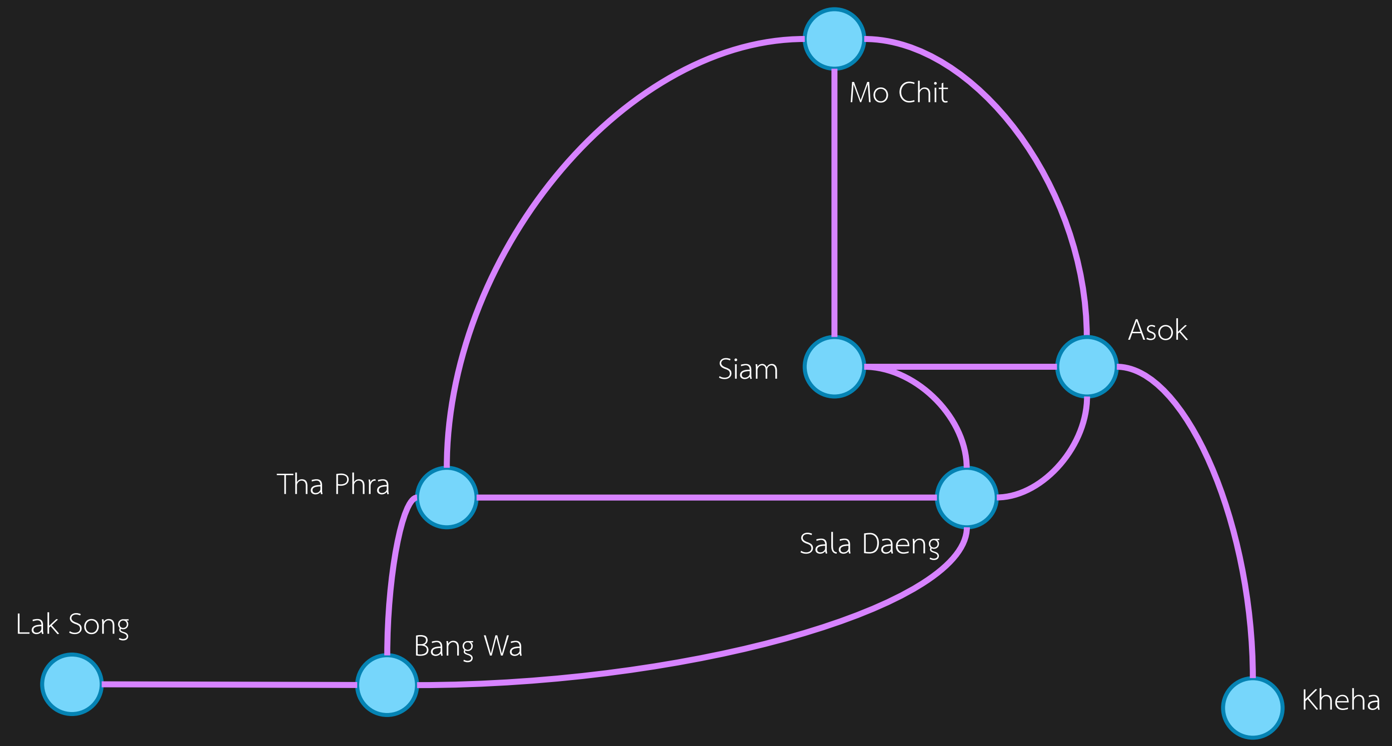
- v0.1 fast push
- v0.2 fast pop, top
- Depends on which operation we use most often
- The real version works like v0.2, we maintain some rules of the data that is stored in the vector such that
  - We know where max is (for fast top)
  - Much faster push, a little bit slower pop
  - Using structure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

Graph and Tree

Quick introduction

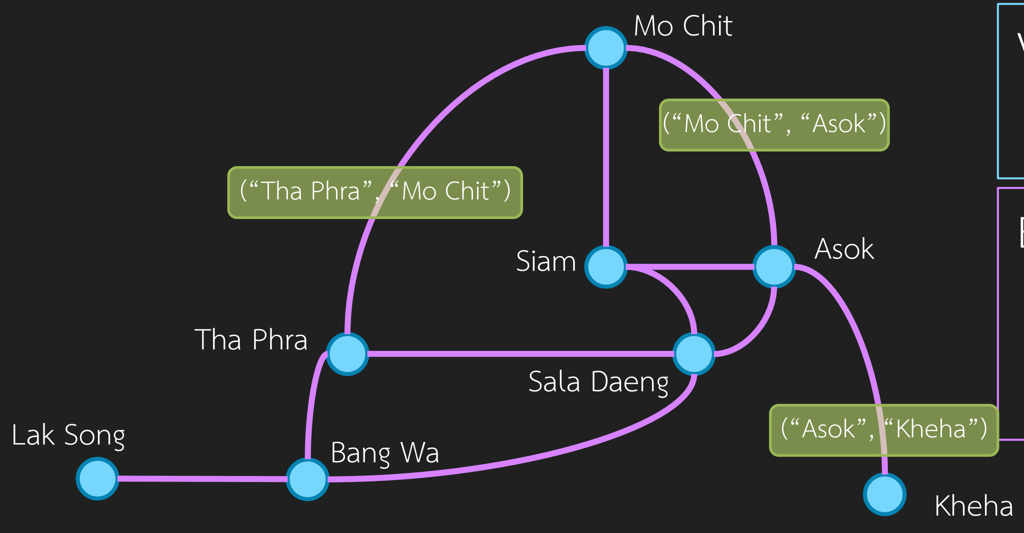
Graph

- 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

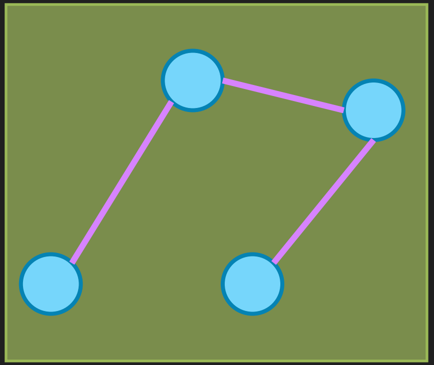


V = { "Mo Chit", "Siam", "Asok", "Sala Daeng", "Tha Phra", "Lak Song", "Bang Wa", "Kheha" }

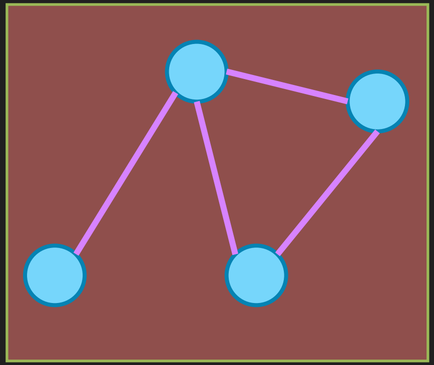
E = { ("Mo Chit", "Asok"), ("Mo Chit", "Siam"), ("Tha Phra", "Mo Chit"), ("Sala Daeng", "Tha Phra") … }

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)

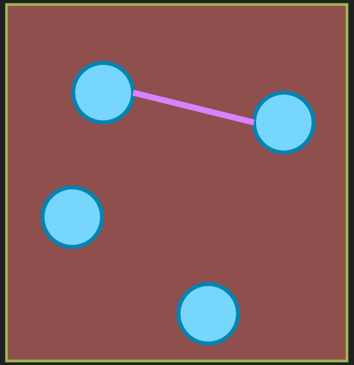


Is a tree



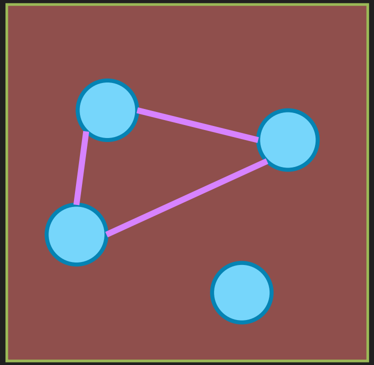
Not Tree

(too many edges)



Not Tree

(too many nodes)



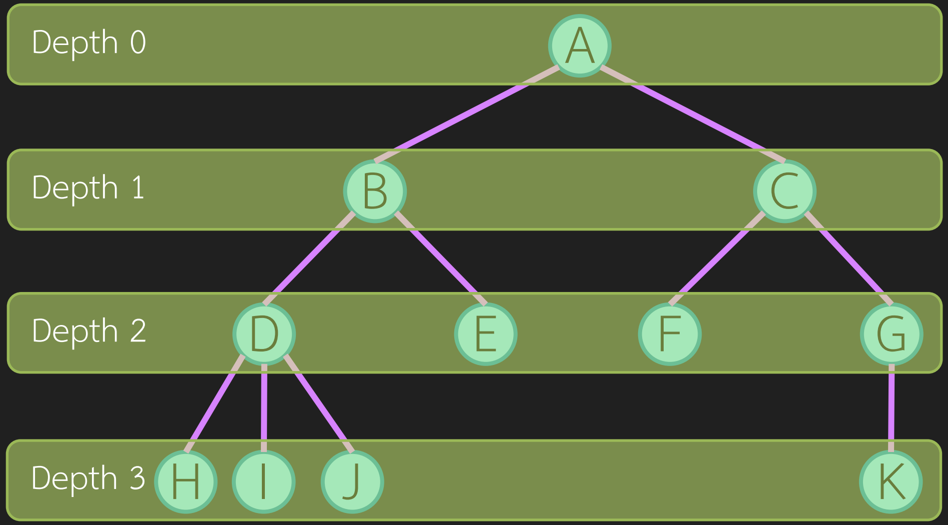
Not Tree

(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)

Root is A



A is the parent of B and C

B is the parent of D and E

K is a child of G

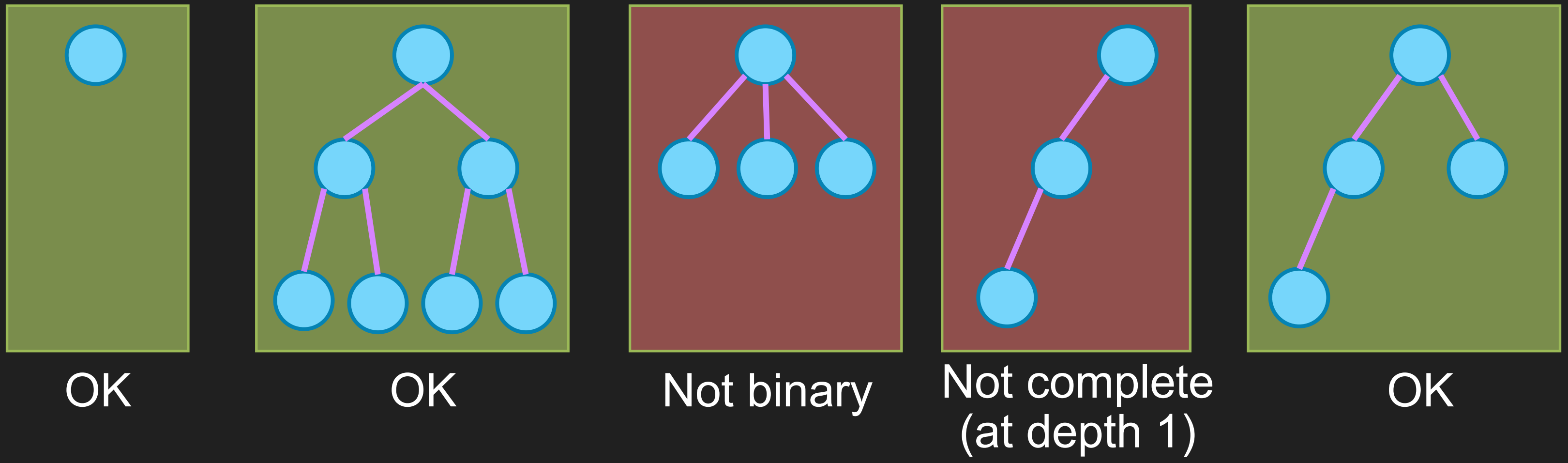
H, I and J are children of D

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

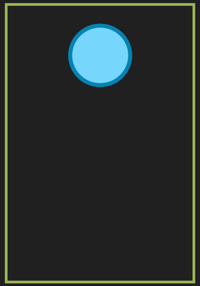


Exercise

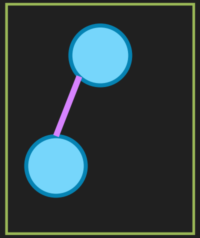
Hint: The answer is unique (There are exactly 1 way to draw a complete binary tree of size k)

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

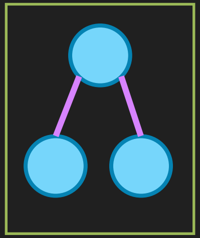
0 nodes
(blank tree)



1 nodes
(only root)



2 nodes



3 nodes

4 nodes

5 nodes

8 nodes

10 nodes

Special Property of a complete binary tree

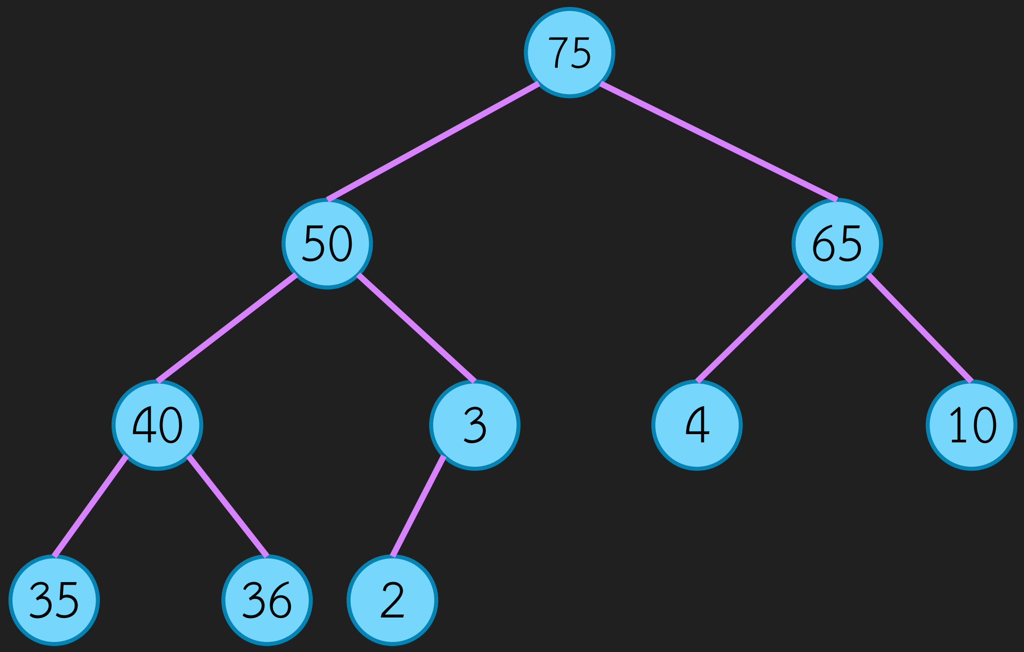
- There is exactly one way to go from any node to any node
- Maximum depth is $\log_2 n$ where n is the number of nodes
    - Because we require completeness and we have 2 possible children

Binary Heap

Using Complete Binary Tree to make priority_queue

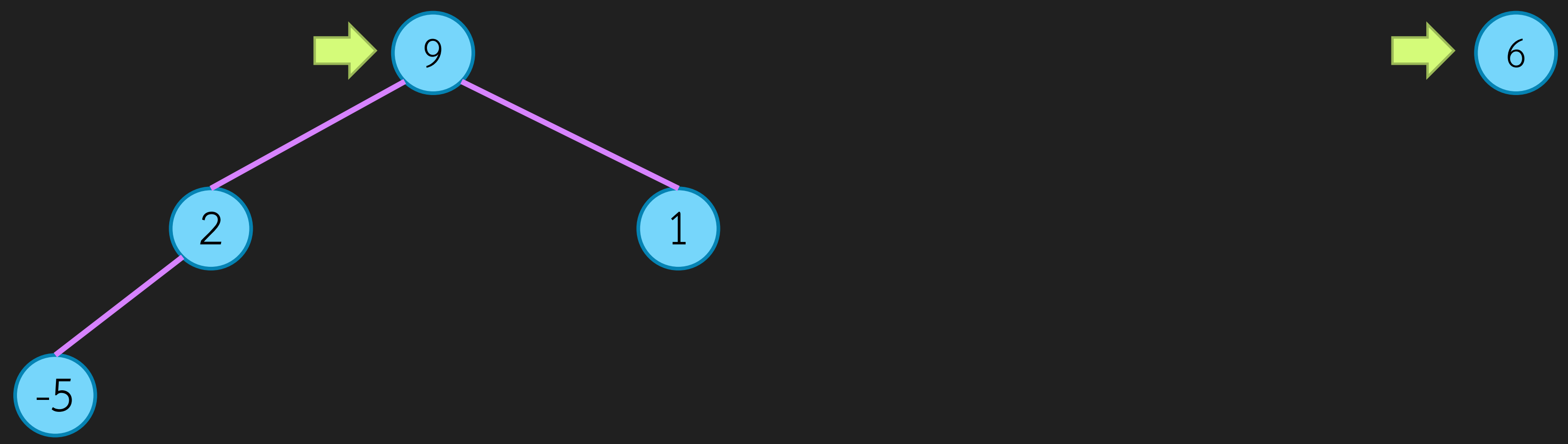
Binary Heap

- 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



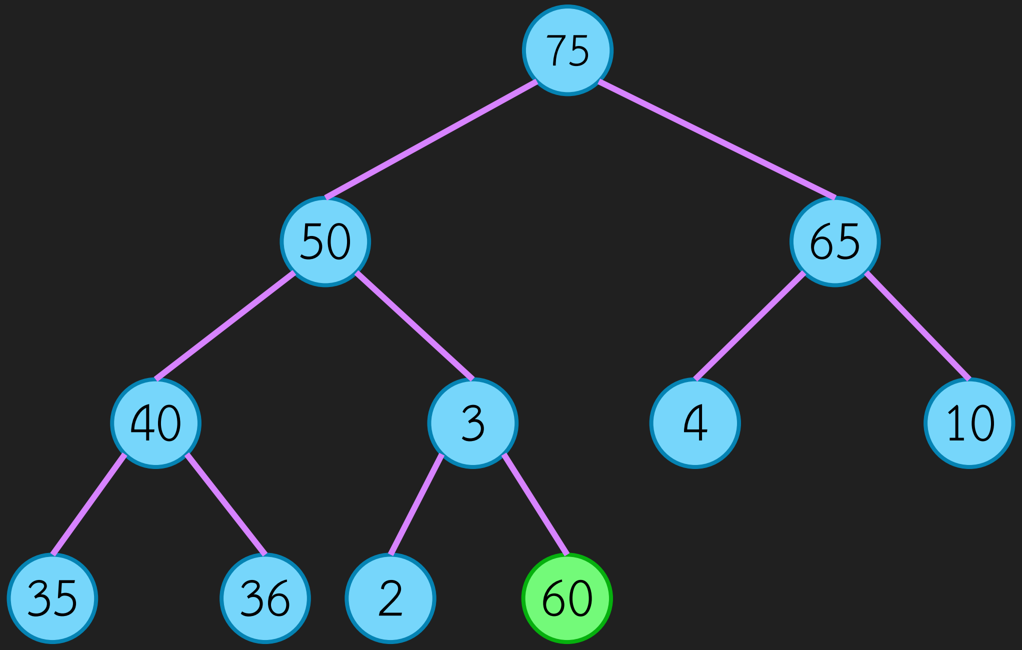
Getting Maximum Data

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



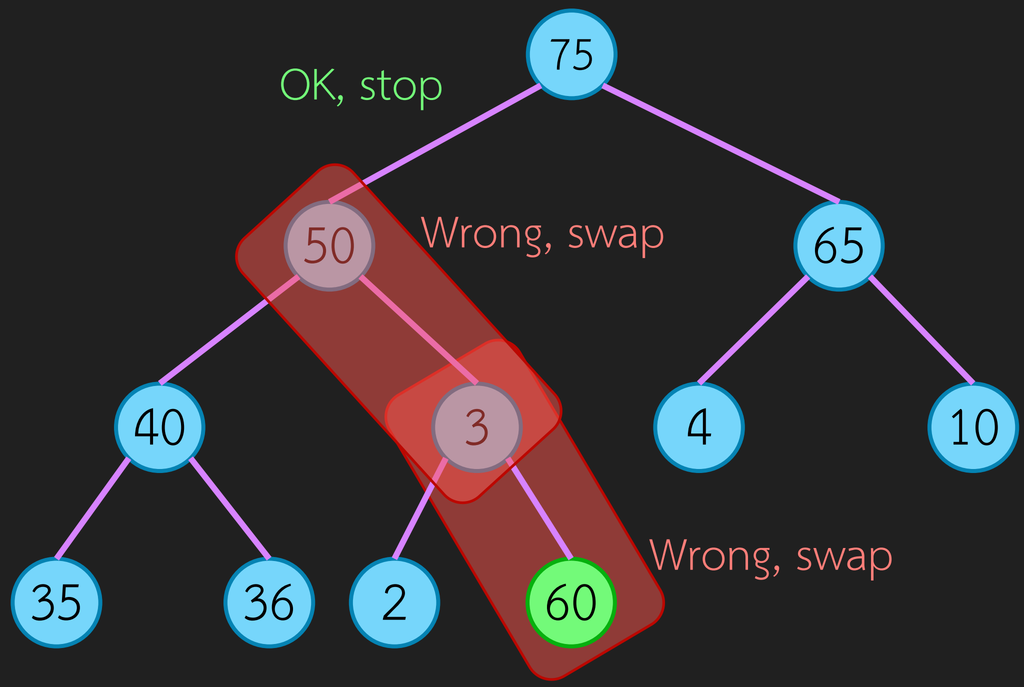
Adding data to Binary Heap

- 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

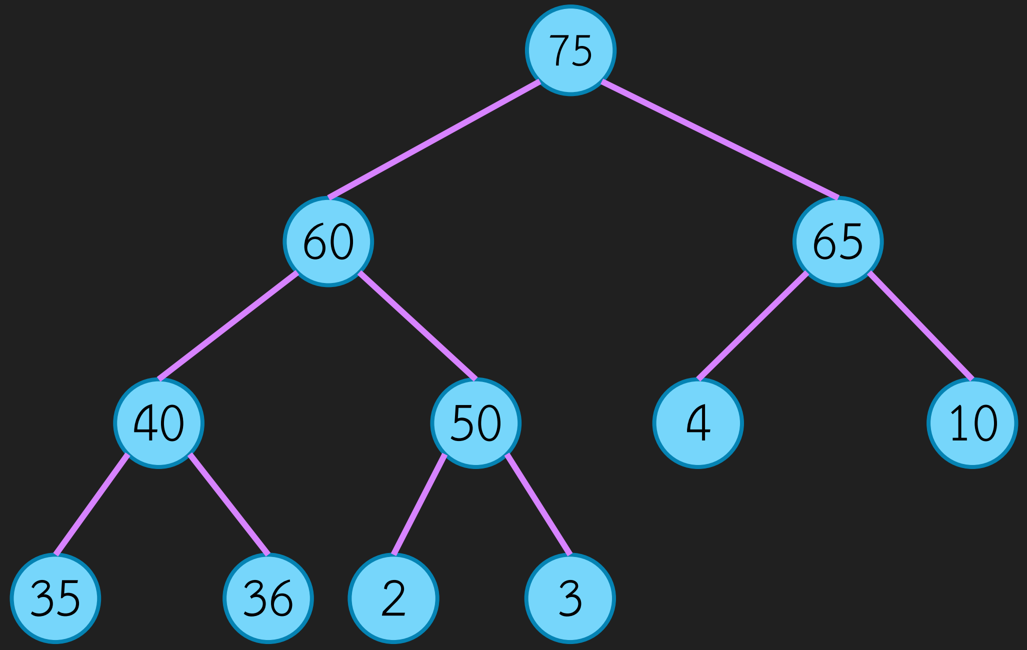
See that, after each swap, the blue swapped node does not violate value rules with its new children

Fix:

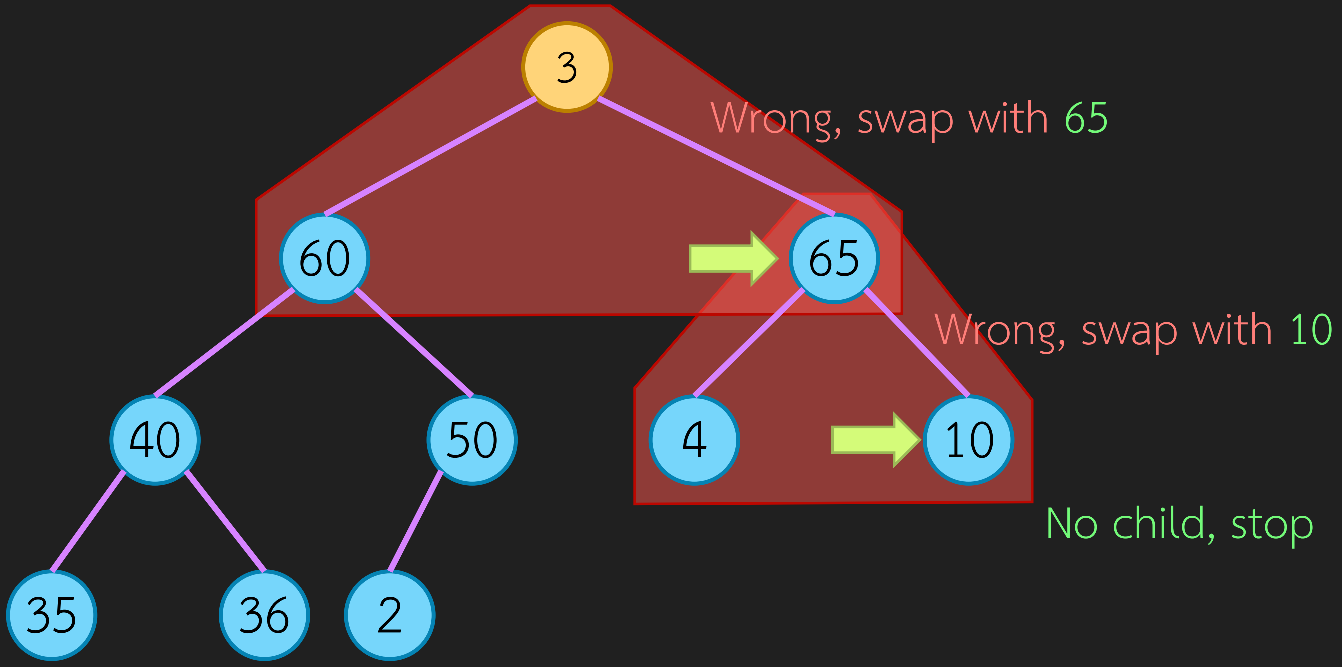
- Check where we just add a node, if value rules is broken, swap with parent
- After swap, re-check with new parent
- Keep doing until correct or at root

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
  - 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)

O(log n)

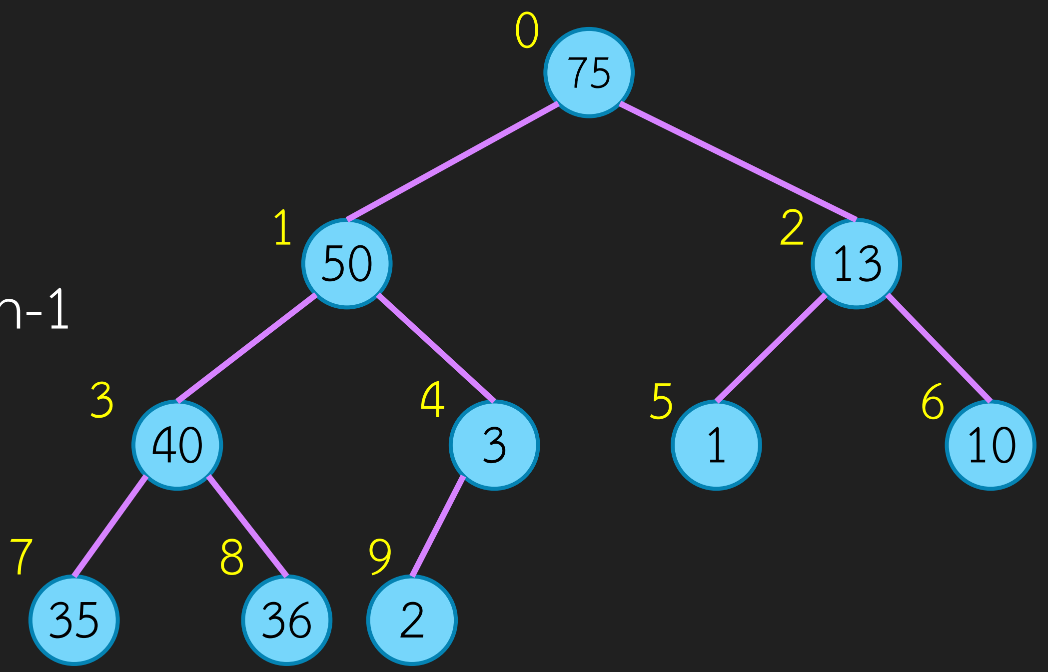
- Pop
  - Fixing value rules is O(h) where h is the maximum number of depth of the tree (we call this value tree height)

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

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

```
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

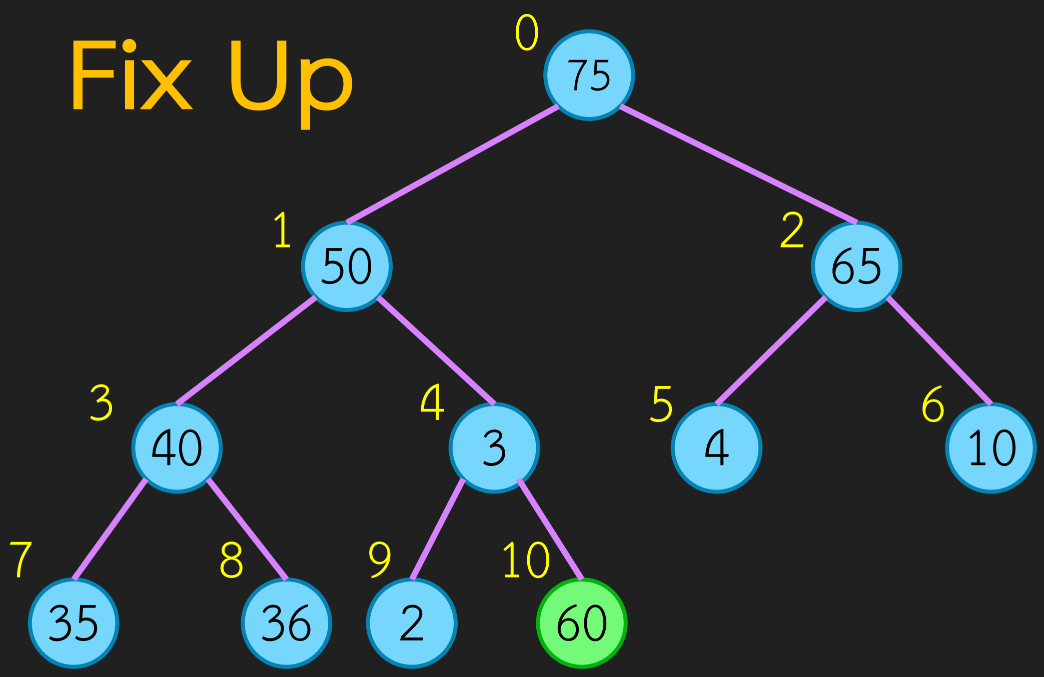
```
void expand(size_t capacity) {
  T *arr = new T[capacity]();
  for (size_t i = 0;i < mSize;i++) {
    arr[i] = mData[i];
  }
  delete [] mData;
  mData = arr;
  mCap = capacity;
}
```

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

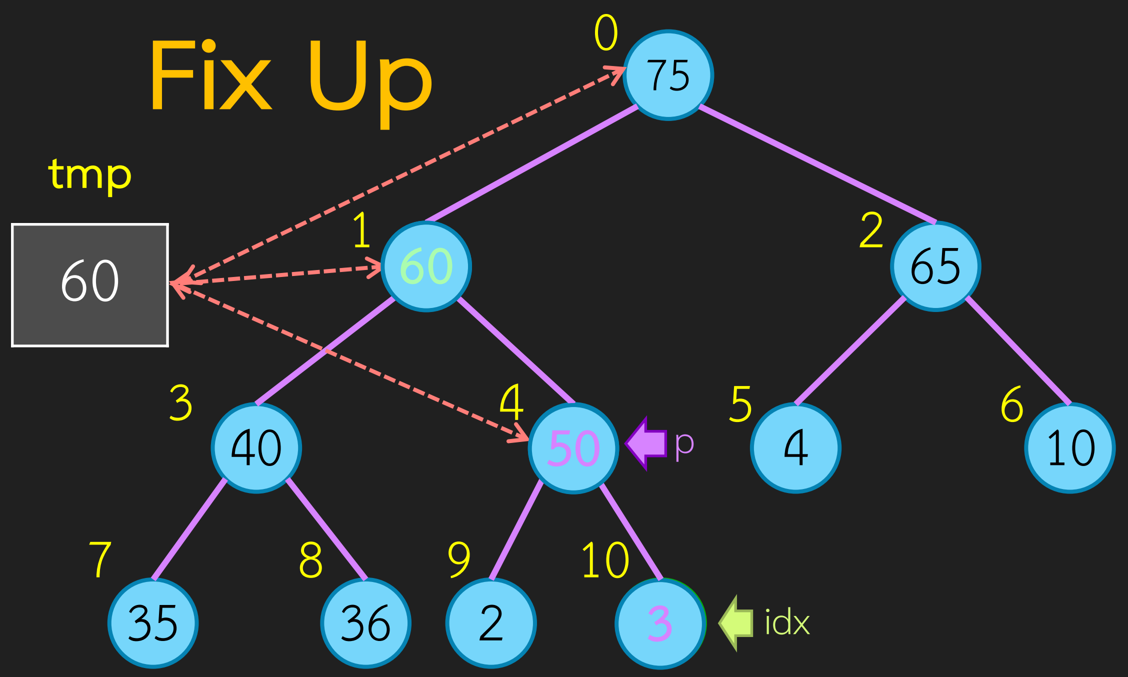


- Instead of actual swap, we perform insert and find appropriate position at the same time

```
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

```cpp
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 )
  { }
```

```cpp
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

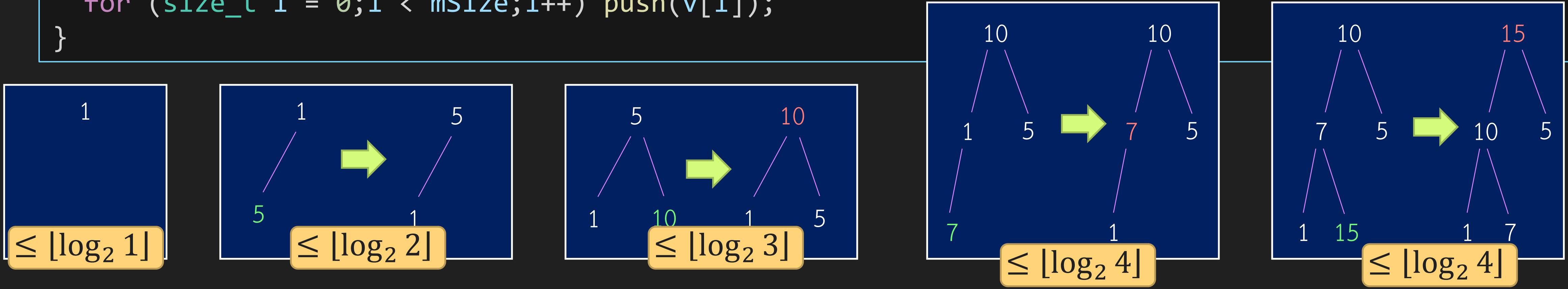
```
void pop() {
  mData[0] = mData[mSize-1];
  mSize--;
  fixDown(0);
}
```

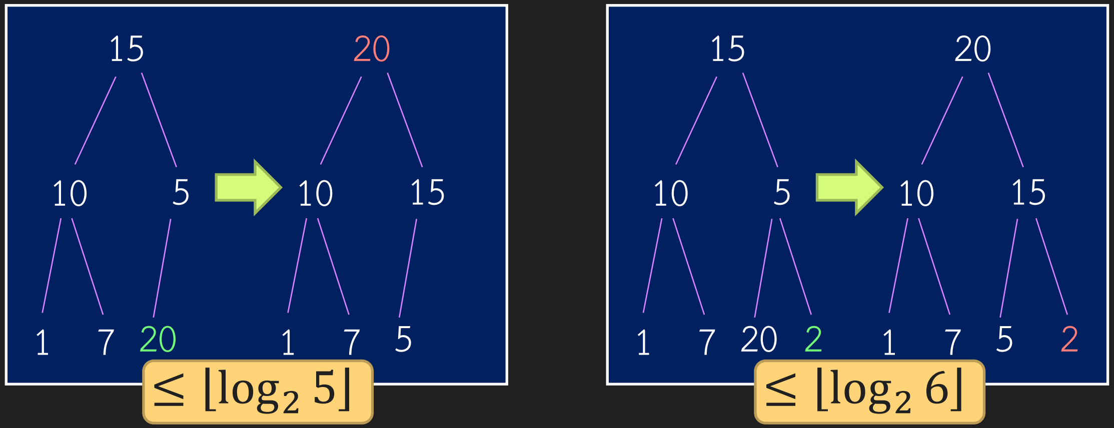
```
void fixDown(size_t idx) {
  T tmp = mData[idx];
  size_t c;
  while ((c = 2 * idx + 1) < mSize) {
    if (c + 1 < mSize && mLess(mData[c],mData[c + 1]) ) c++;
    if ( mLess(mData[c],tmp) ) break;
    mData[idx] = mData[c];
    idx = c;
  }
  mData[idx] = tmp;
}
```

- 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]);
}
```





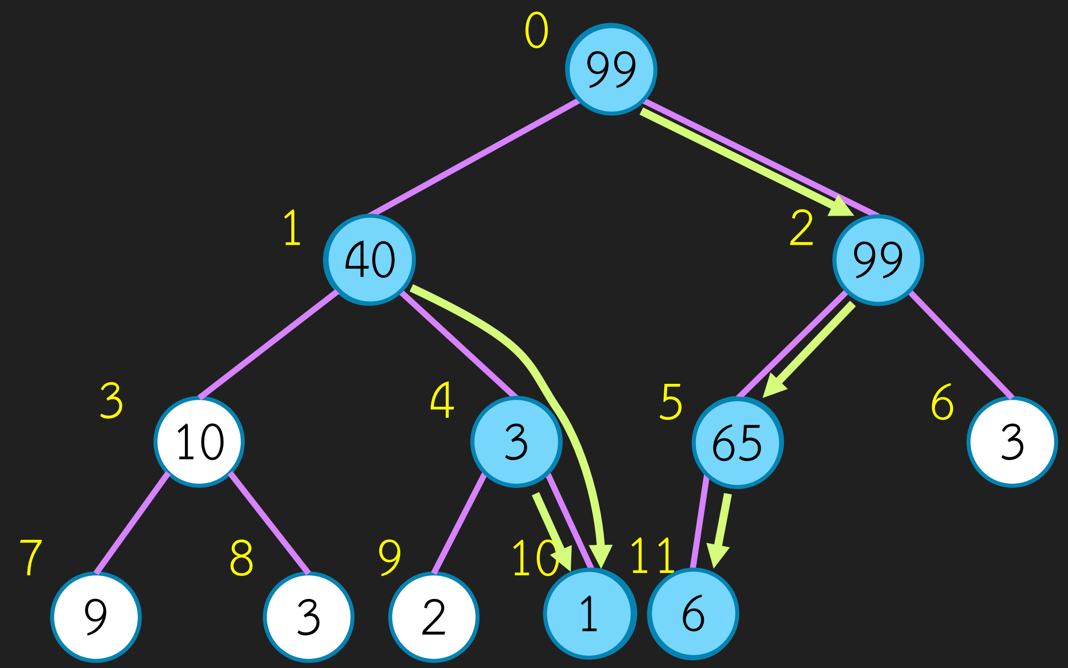
Total $\leq \lfloor \log_2 1 \rfloor + \lfloor \log_2 1 \rfloor + \cdots + \lfloor \log_2 n \rfloor$

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

$\lfloor \log_2 n! \rfloor$ 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);
}
```

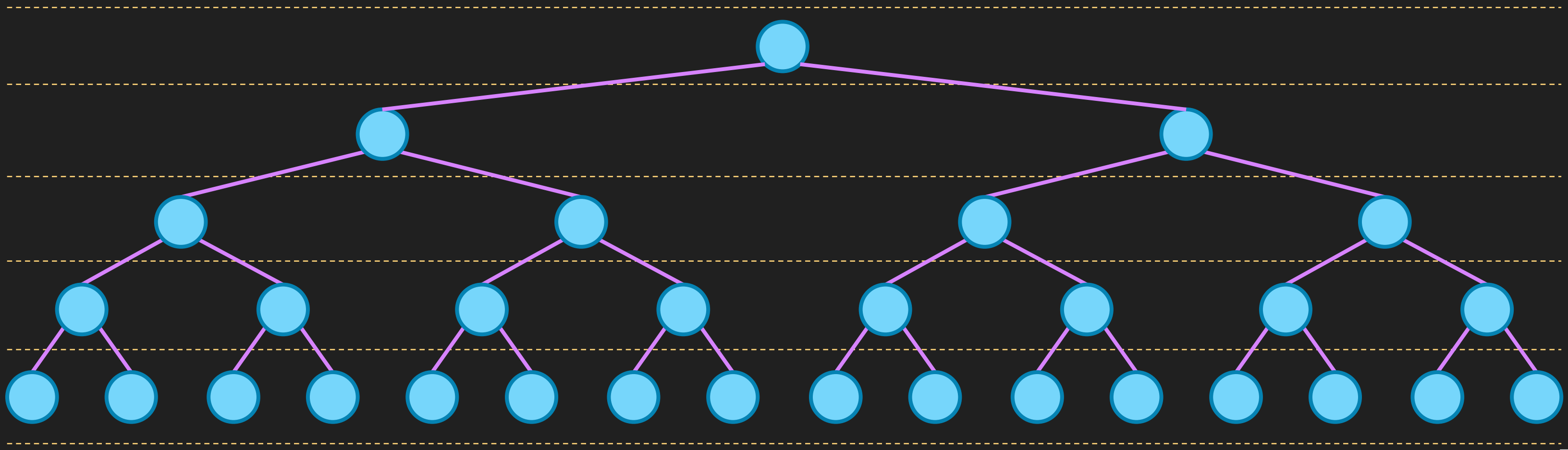


- Consider each node to be a Binary Heap
- Fix down from back to front

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
k	2^k	h-k

total = $\sum_{k=0}^{h} k2^{h-k}$

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

$$= 2^h \sum_{k=0}^{h} k2^{-k} \quad < 2^h \underbrace{\sum_{k=0}^{\infty} k2^{-k}}_{\text{This is 2}} \quad = 2^h 2 = 2^{h+1}$$

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