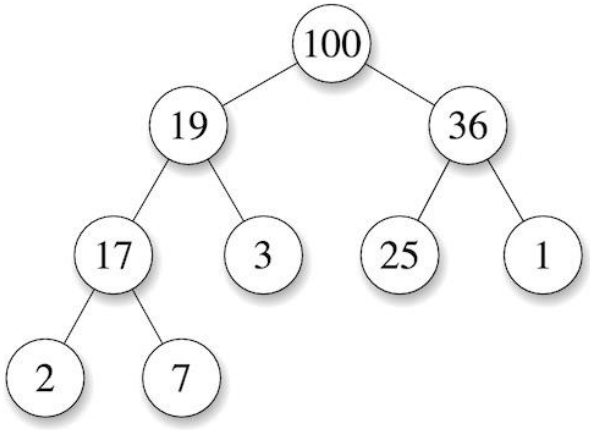
HEAPS

OVERVIEW

OVERVIEW

- **What is a heap?**

Heap of sand



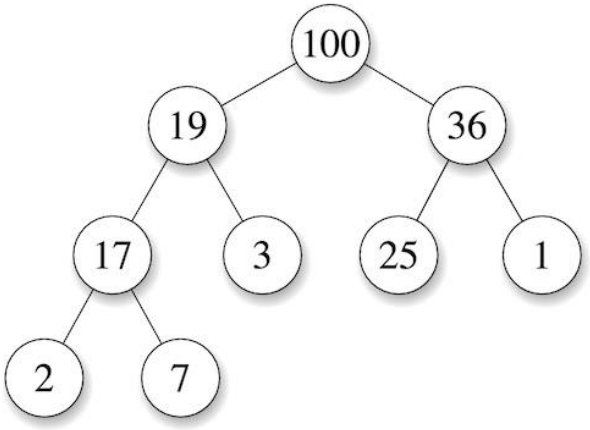
Heap data structure

OVERVIEW

- **Sorites paradox (from Wikipedia)**
  - 1,000,000 grains of sand is a heap of sand
  - A heap of sand minus one grain is still a heap
  - Hence 999,999 grains of sand is a heap
  - Hence 999,998 grains of sand is a heap
  - …
  - Hence 1 grain of sand is a heap

OVERVIEW

- **In computer science, a heap is a very useful data structure**
  - A heap must always be a complete binary tree
  - The value of every parent is greater than their children
  - The root of a heap contains the largest value



OVERVIEW

- **What are the operations on a heap?**

- **We can insert data values into the heap**
  - When we do this, we must move data around to ensure the largest value stays at the top of the heap (in the root node)

- **We can remove the top value from the heap**
  - When we do this, we must move data around to ensure the largest value stays at the top of the heap (in the root node)

- **We can also print data in a heap to see what it contains**

OVERVIEW

- **What can we do with a heap?**

- **We can implement a priority queue**
  - Data with different priority values are put into queue
  - Highest priority data is removed from the queue
  - Used extensively in an O/S for task scheduling
  - Can also be used to simulate customer service scenarios

- **We can implement heap sort**
  - Loop over unsorted data inserting into heap
  - Loop again to remove data in descending order

HEAPS

HEAP INSERT

HEAP INSERT

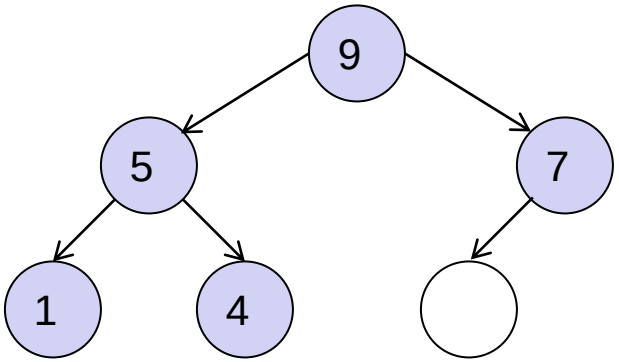
- **Goal of Heap Insert**
  - Add new data value into the heap
  - Make sure heap is still complete binary tree
  - Make sure all parents greater than all their children

- **Heap Insert Algorithm:**
  - Create new node to the right of all other leaf nodes on the bottom level of the tree
  - Store new value in the new node
  - If new value is greater than parent then swap the values
  - Move up to parent node and continue swapping as needed

HEAP INSERT

Insert value 12:

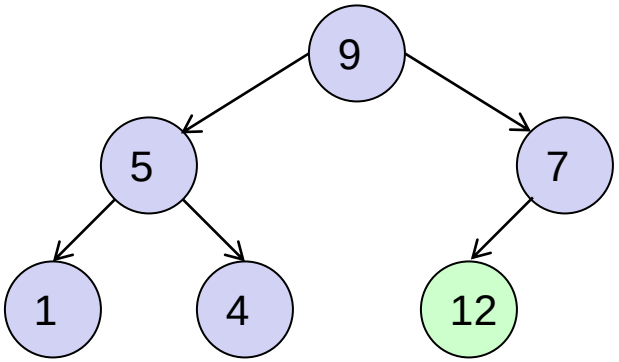
- Find insert location
- Add new node



HEAP INSERT

Insert value 12:

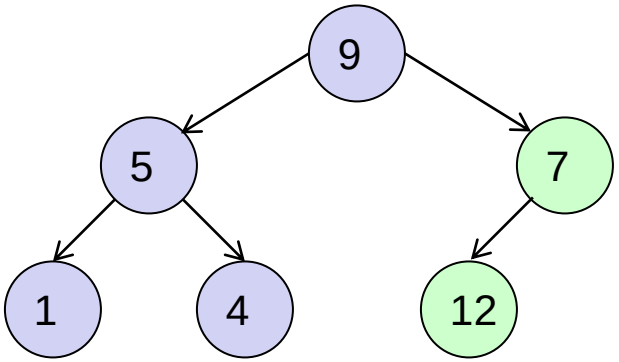
- Find insert location
- Add new node
- Insert value 12



HEAP INSERT

Insert value 12:

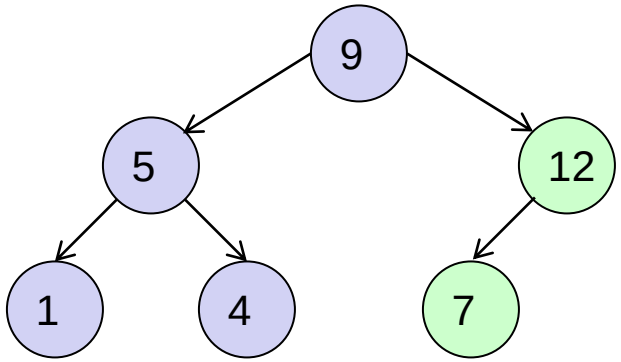
- Find insert location
- Add new node
- Insert value 12
- Compare parent and child



HEAP INSERT

Insert value 12:

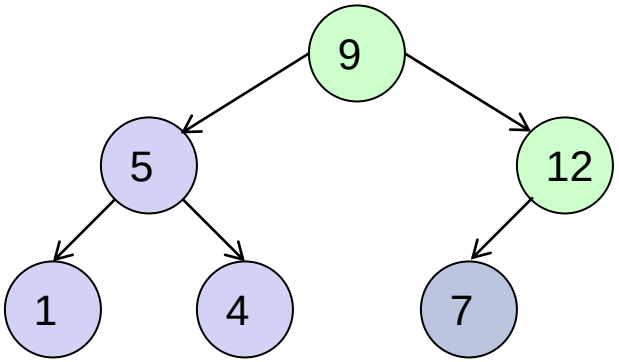
- Find insert location
- Add new node
- Insert value 12
- Compare parent and child
- 12 > 7 so swap values



HEAP INSERT

Insert value 12:

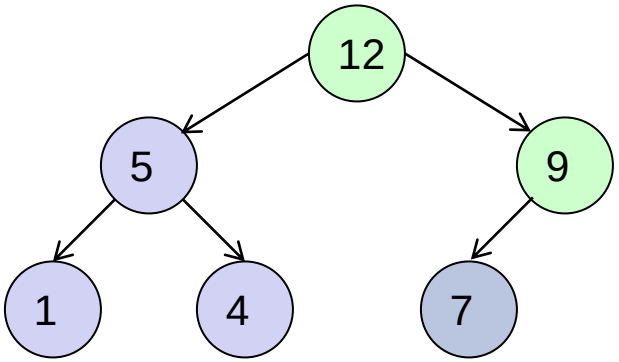
- Find insert location
- Add new node
- Insert value 12
- Compare parent and child
- 12 > 7 so swap values
- Compare parent and child



HEAP INSERT

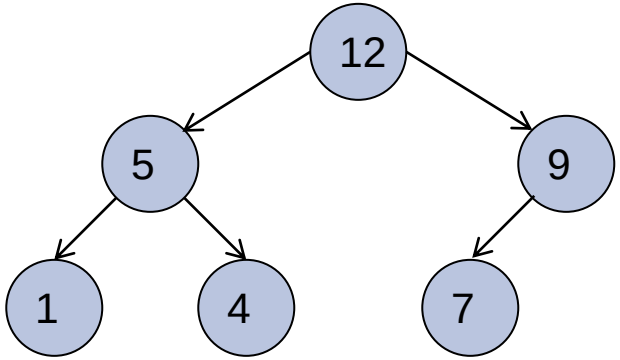
Insert value 12:

- Find insert location
- Add new node
- Insert value 12
- Compare parent and child
- 12 > 7 so swap values
- Compare parent and child
- 12 > 9 so swap values
- Stop comparisons (at root)



HEAP INSERT

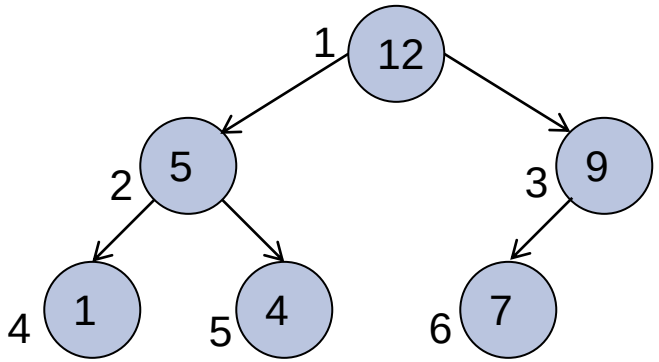
- **How do we find location for new node?**
  - This is very tricky if we use pointer based heap
  - It is very easy if we use an array based heap
  - We simply fill the array from left to right
  - We want to use array indices to go up/down the tree



12	5	9	1	4	7	
1	2	3	4	5	6	7

HEAP INSERT

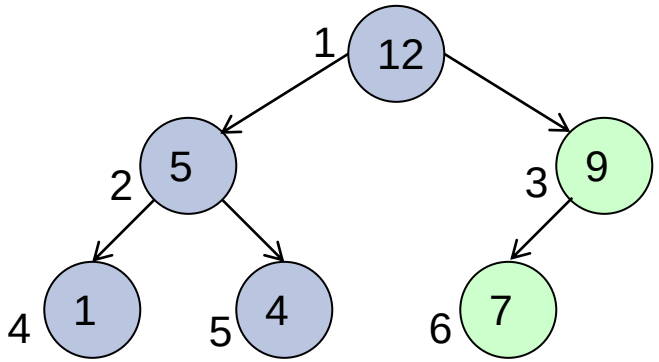
- **What is the secret to solution?**
  - Number nodes from top-bottom, left-right starting at one
  - Parent index = child index / 2
  - Left child index = parent index * 2
  - Right child index = parent index * 2 + 1

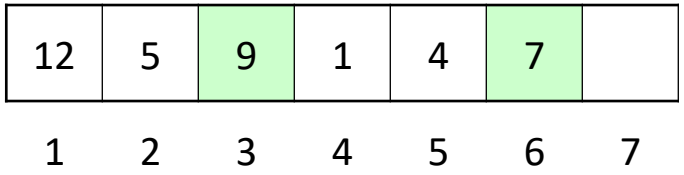


12	5	9	1	4	7	
1	2	3	4	5	6	7

HEAP INSERT

- **What is the secret to solution?**
  - Number nodes from top-bottom, left-right starting at one
  - Parent index = child index / 2
  - Left child index = parent index * 2
  - Right child index = parent index * 2 + 1

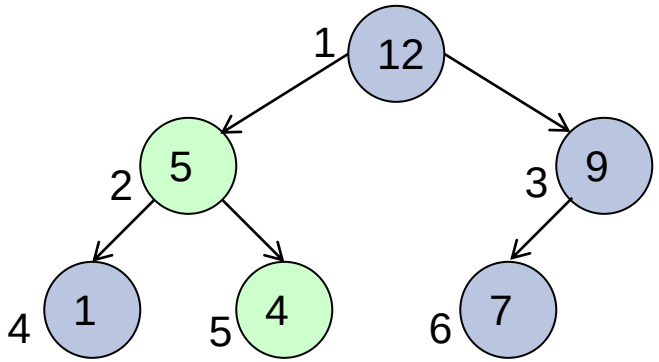


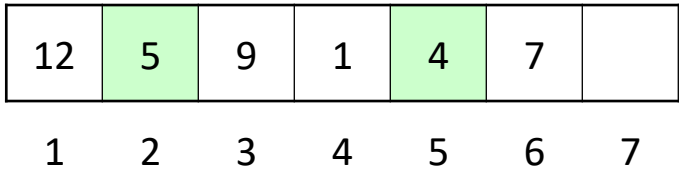


The parent of 7 node is in array location 6/2=3

HEAP INSERT

- **What is the secret to solution?**
  - Number nodes from top-bottom, left-right starting at one
  - Parent index = child index / 2
  - Left child index = parent index * 2
  - Right child index = parent index * 2 + 1

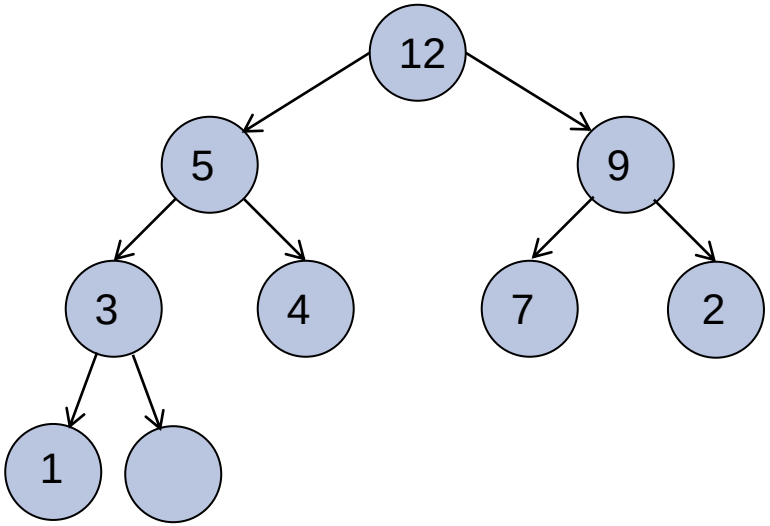




The right child of 5 node is in location 2*2+1=5

HEAP INSERT

Insert value 8:

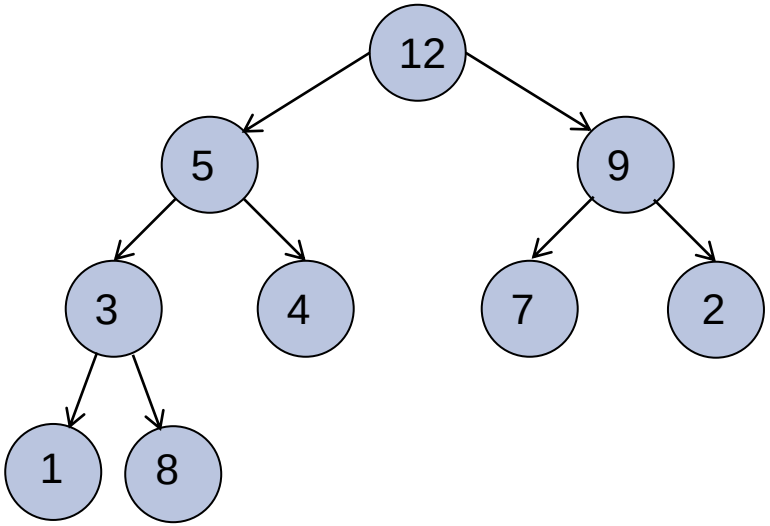


12	5	9	3	4	7	2	1			
1	2	3	4	5	6	7	8	9	10	11

new node at location 9

HEAP INSERT

Insert value 8:

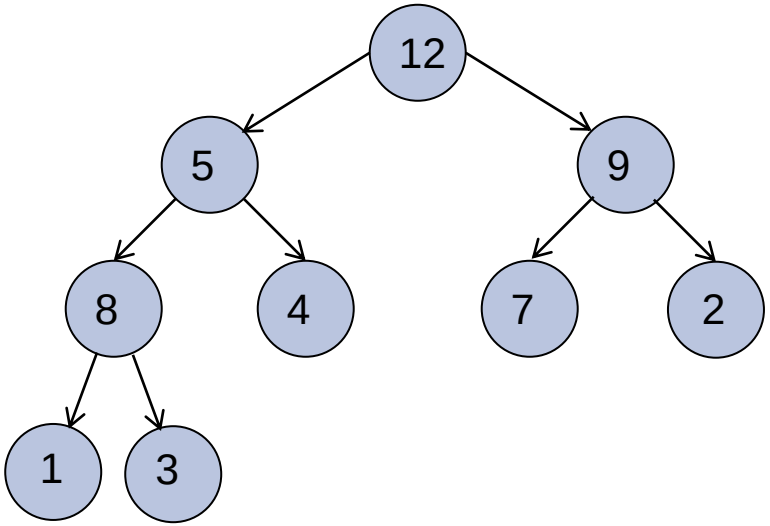


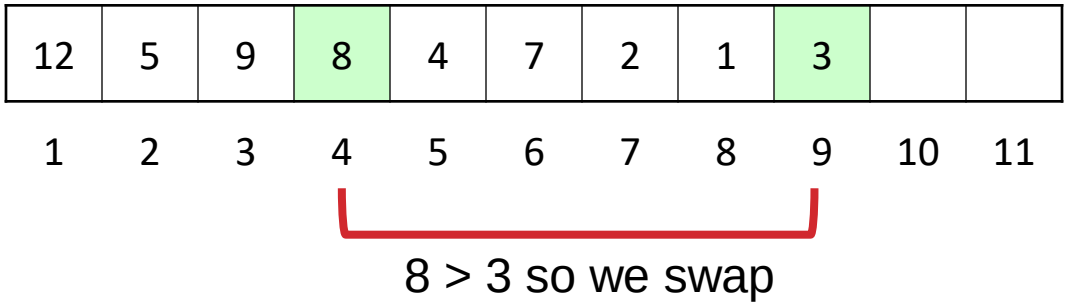
12	5	9	3	4	7	2	1	8		
1	2	3	4	5	6	7	8	9	10	11

store value 8

HEAP INSERT

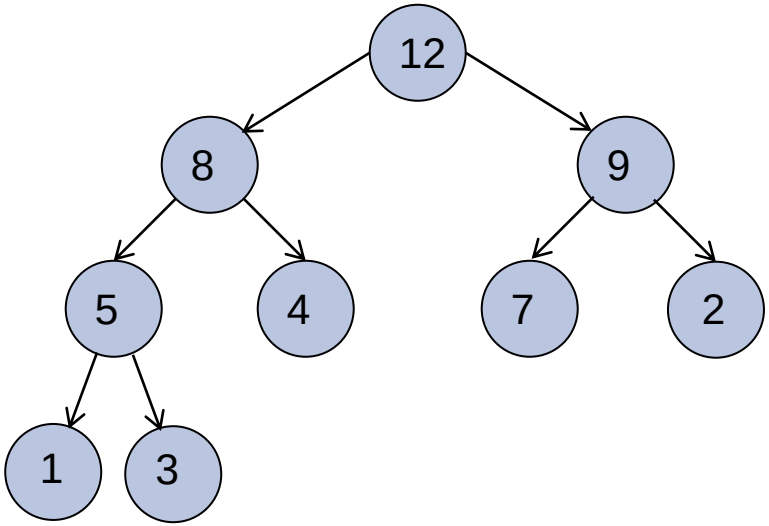
Insert value 8:

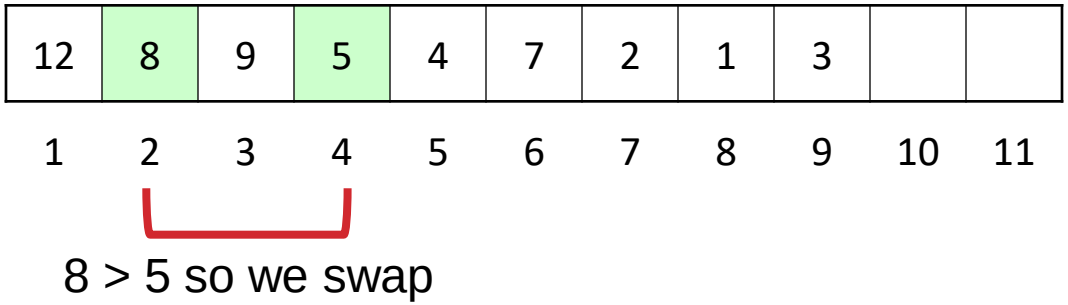




HEAP INSERT

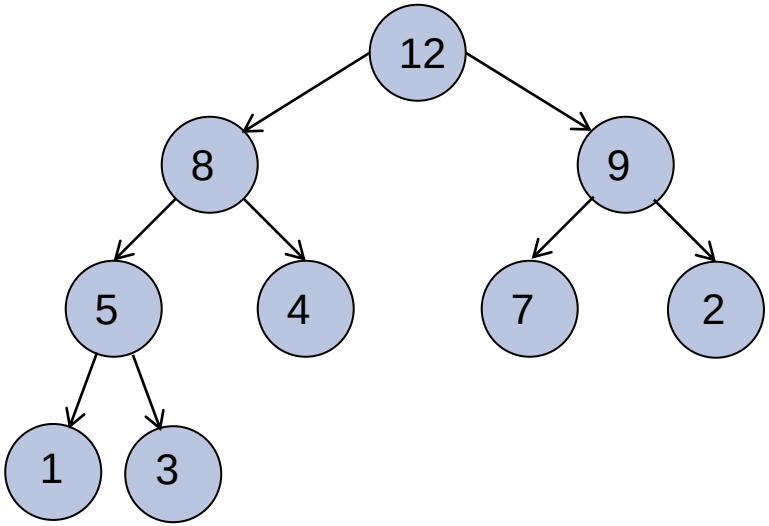
Insert value 8:

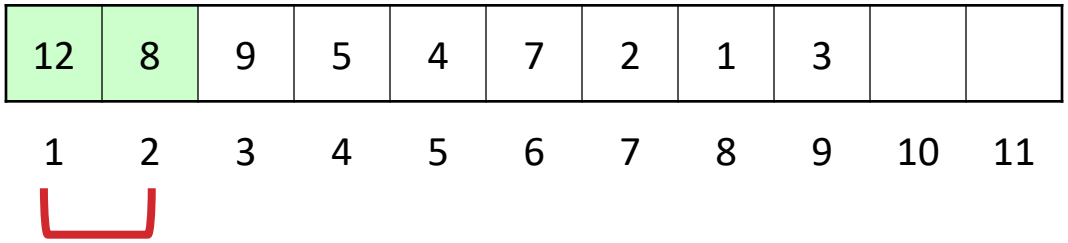




HEAP INSERT

Insert value 8:





8 < 12 so we stop swapping

HEAP INSERT

- **Heap Insert Analysis:**
  - Finding the new node location is O(1)
  - Inserting data value in new node is O(1)
  - A complete binary tree with N nodes has height of O(logN)
  - The max number of parent-child swaps will be O(logN)
  - In practice only ½ that number are required

- **Inserting data into heap is O(logN)**

HEAPS

HEAP REMOVE

HEAP REMOVE

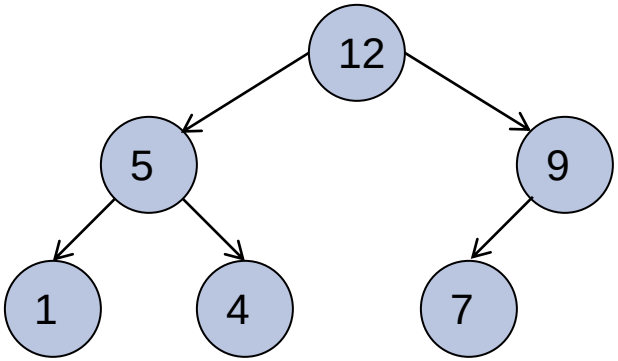
- **Goal of Heap Remove**
  - Remove largest data value from the heap
  - Make sure heap is still complete binary tree
  - Make sure all parents greater than all their children

- **Heap Remove Algorithm:**
  - Copy largest value (the root) into temporary variable
  - Copy value in rightmost leaf node into the root node
  - Delete the rightmost leaf node from the heap
  - Compare value at root to values of both children
  - If root is smaller than either child, swap value at root with value of largest child, and repeat as needed

HEAP REMOVE

Remove value 12:

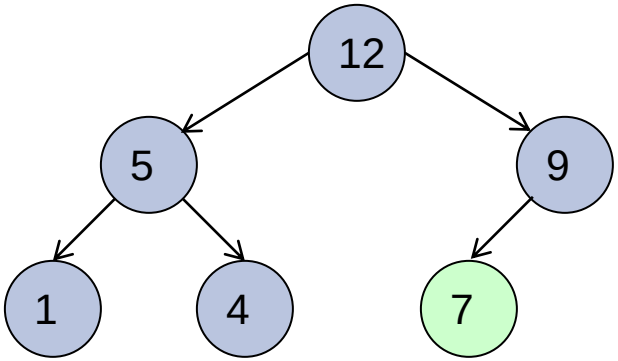
- Save copy of root value (12)



HEAP REMOVE

Remove value 12:

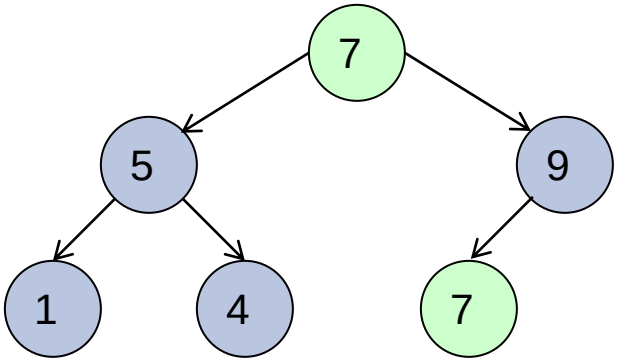
- Save copy of root value (12)
- Find rightmost leaf node



HEAP REMOVE

Remove value 12:

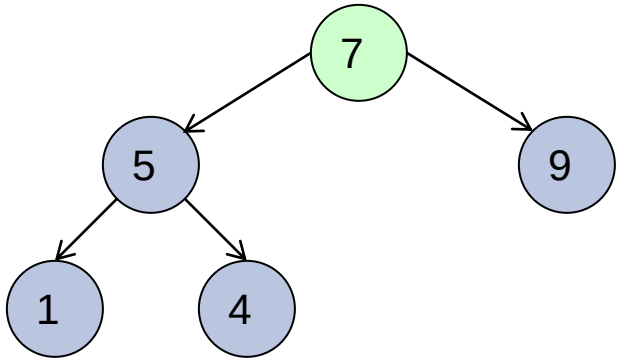
- Save copy of root value (12)
- Find rightmost leaf node
- Copy this value to root



HEAP REMOVE

Remove value 12:

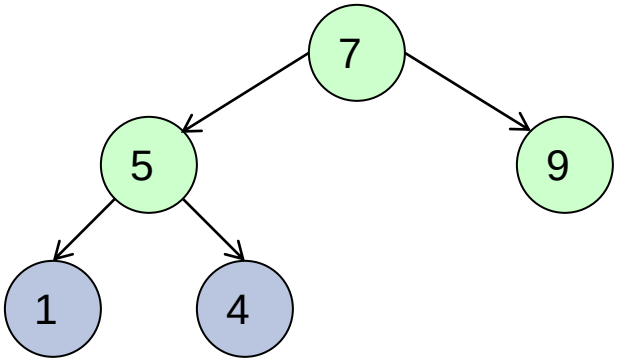
- Save copy of root value (12)
- Find rightmost leaf node
- Copy this value to root
- Delete rightmost leaf node



HEAP REMOVE

Remove value 12:

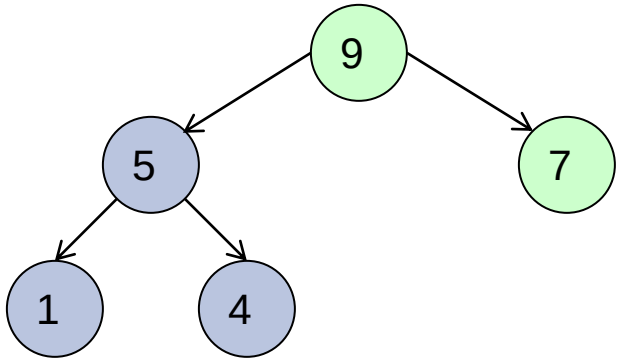
- Save copy of root value (12)
- Find rightmost leaf node
- Copy this value to root
- Delete rightmost leaf node
- Compare root to two children



HEAP REMOVE

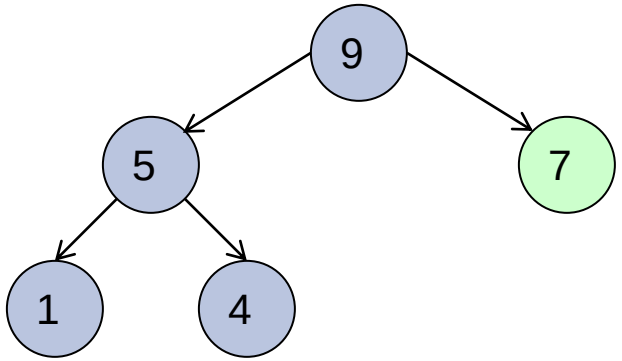
Remove value 12:

- Save copy of root value (12)
- Find rightmost leaf node
- Copy this value to root
- Delete rightmost leaf node
- Compare root to two children
- 9 > 7 so swap values in nodes



HEAP REMOVE

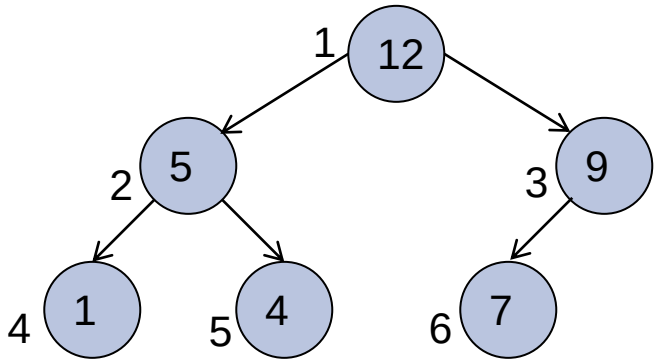
Remove value 12:



- Save copy of root value (12)
- Find rightmost leaf node
- Copy this value to root
- Delete rightmost leaf node
- Compare root to two children
- 9 > 7 so swap values in nodes
- Move down to 7 node
- Stop comparisons (no children)

HEAP REMOVE

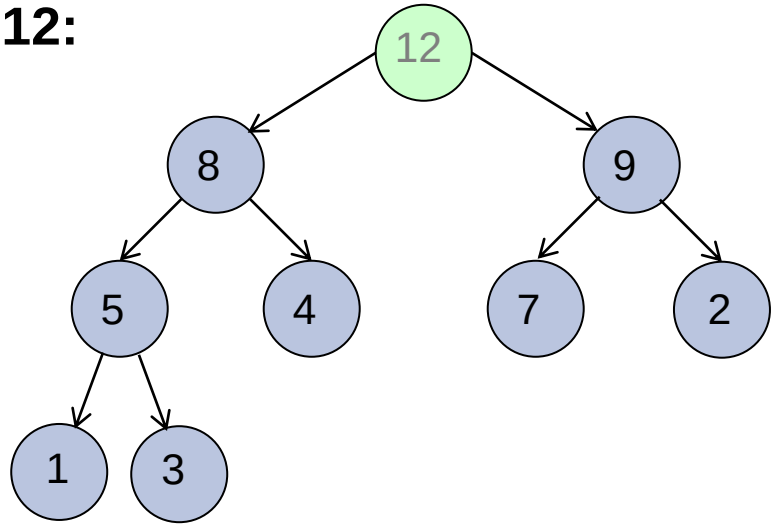
- **Heap removal also uses an array representation:**
  - Number nodes from top-bottom, left-right starting at one
  - Parent index = child index / 2
  - Left child index = parent index * 2
  - Right child index = parent index * 2 + 1



12	5	9	1	4	7	
1	2	3	4	5	6	7

HEAP REMOVE

Remove value 12:

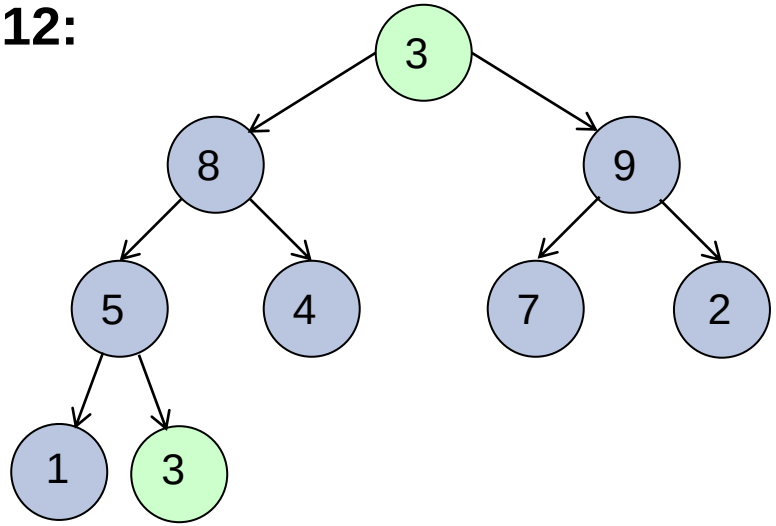


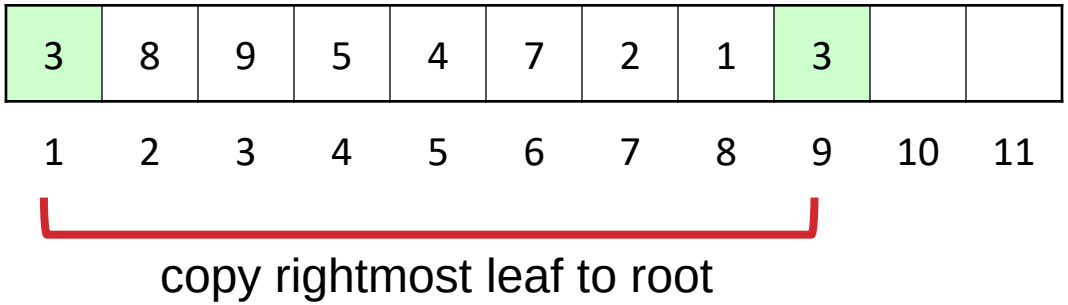
12	8	9	5	4	7	2	1	3		
1	2	3	4	5	6	7	8	9	10	11

remove root value

HEAP REMOVE

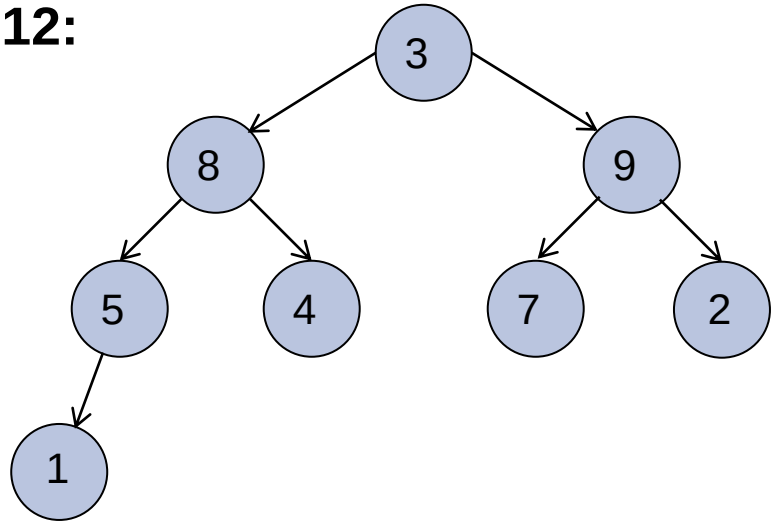
Remove value 12:





HEAP REMOVE

Remove value 12:

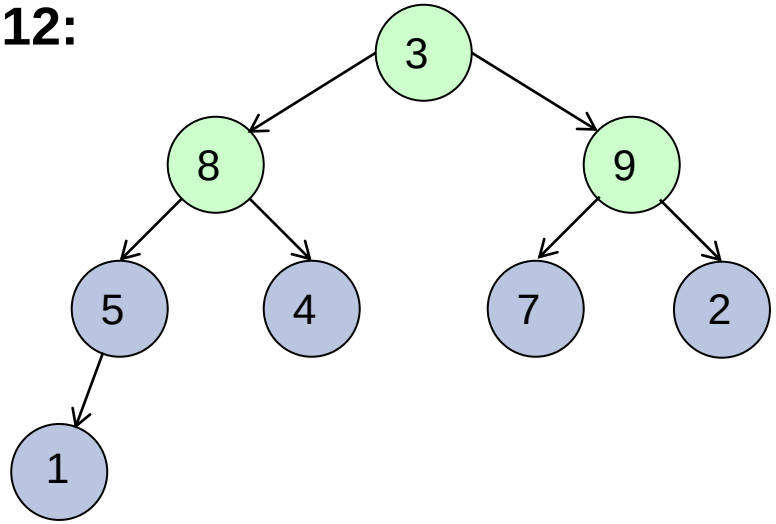


3	8	9	5	4	7	2	1			
1	2	3	4	5	6	7	8	9	10	11

delete rightmost leaf node

HEAP REMOVE

Remove value 12:

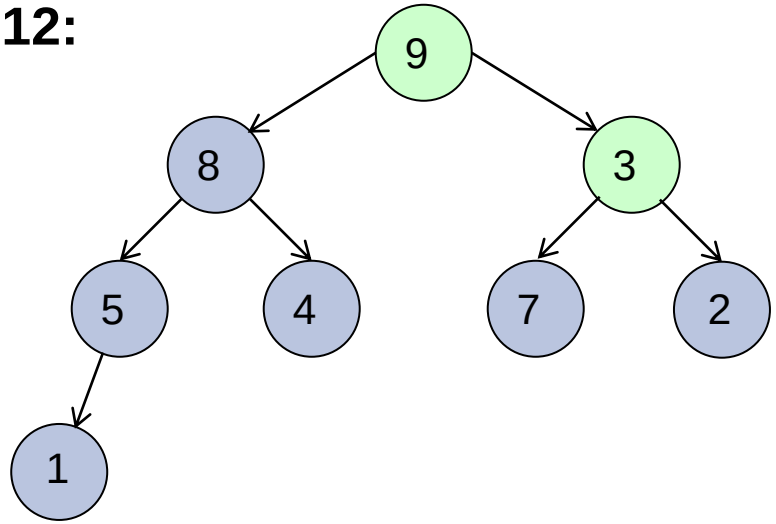


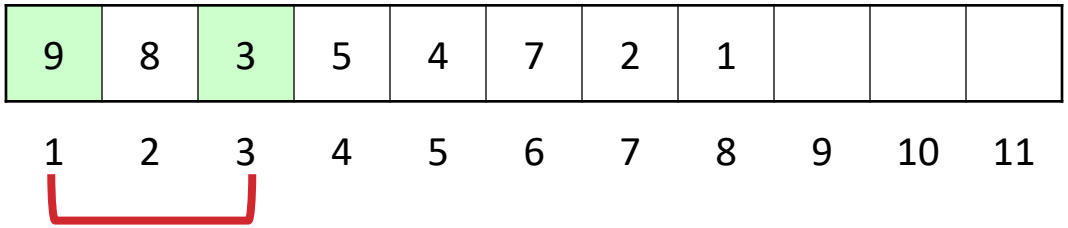
3	8	9	5	4	7	2	1			
1	2	3	4	5	6	7	8	9	10	11

compare root to two children

HEAP REMOVE

Remove value 12:

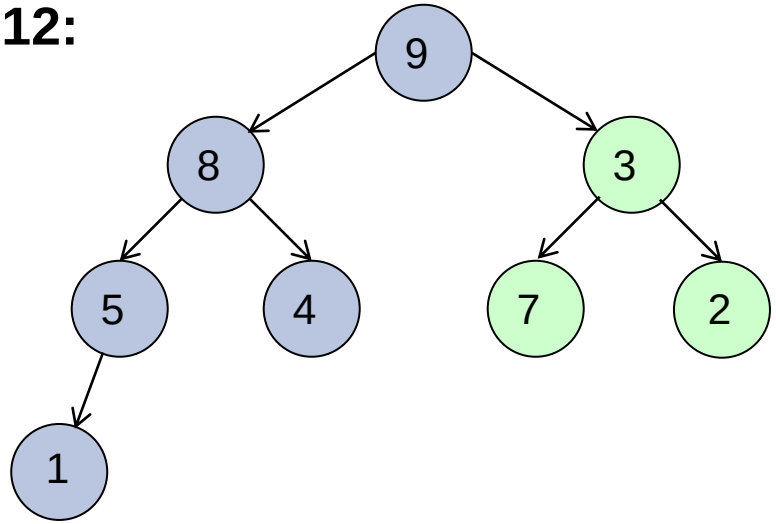




swap with largest child

HEAP REMOVE

Remove value 12:

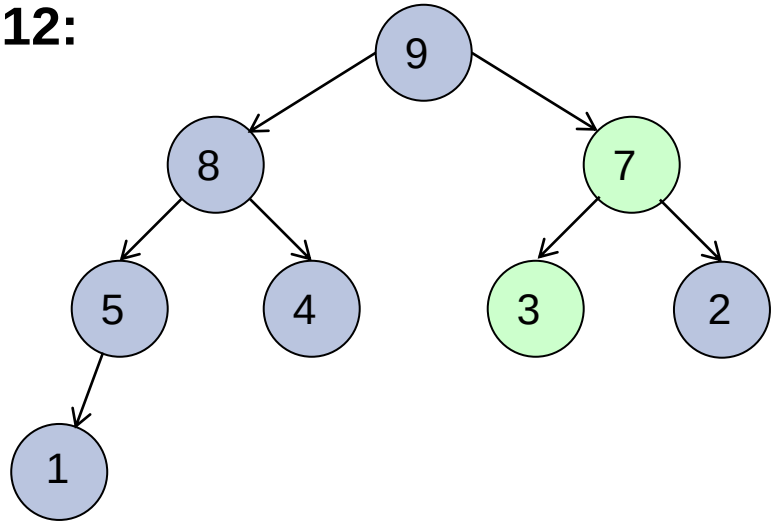


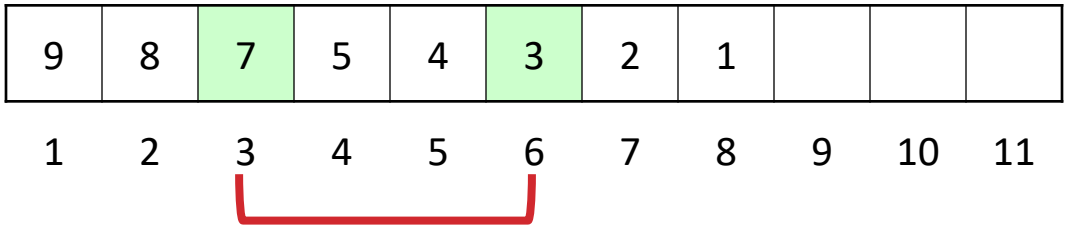
9	8	3	5	4	7	2	1			
1	2	3	4	5	6	7	8	9	10	11

compare swapped node to two children

HEAP REMOVE

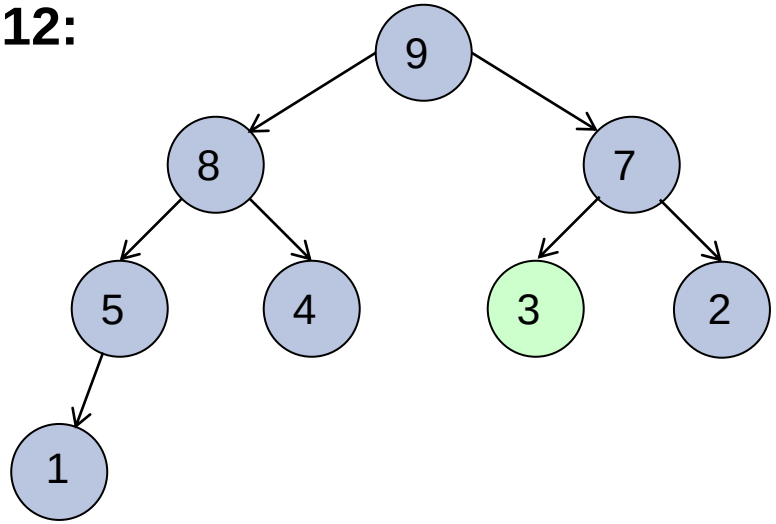
Remove value 12:





HEAP REMOVE

Remove value 12:



9	8	7	5	4	3	2	1			
1	2	3	4	5	6	7	8	9	10	11

stop comparison at leaf

HEAP REMOVE

- **Heap Remove Analysis:**
  - Finding the rightmost leaf node is O(1)
  - Copying data value to root node is O(1)
  - A complete binary tree with N nodes has height of O(logN)
  - The max number of parent-child swaps will be O(logN)
  - In practice only ½ that number are required

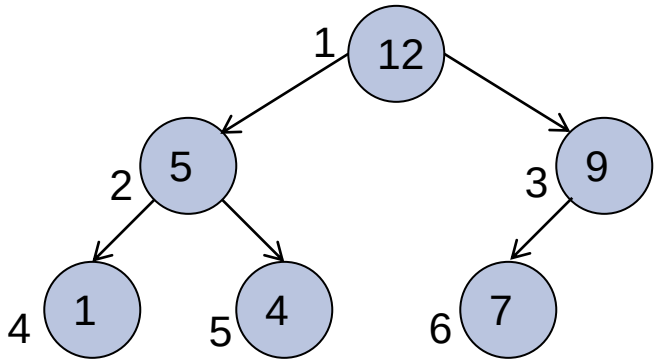
- **Removing data into heap is O(logN)**

HEAPS

HEAP IMPLEMENTATION

HEAP IMPLEMENTATION

- **We represent the heap using an array of nodes**
  - Number nodes from top-bottom, left-right starting at one
  - Parent index = child index / 2
  - Left child index = parent index * 2
  - Right child index = parent index * 2 + 1



12	5	9	1	4	7	
1	2	3	4	5	6	7

HEAP IMPLEMENTATION

```
const int MAX_HEAP_SIZE = 100;

class Heap
{
 public:
   // Constructors
   Heap();
   Heap(const Heap & heap);
   ~Heap();
```

HEAP IMPLEMENTATION

```
   // Methods
   bool Insert(int num);
   bool Remove(int &num);
   bool IsEmpty();
   bool IsFull();
   void Print();

 private:
   int Count;
   int heap[MAX_HEAP_SIZE + 1];
};
```

Both methods return true if successful and false otherwise

We could replace fixed size array with a vector if desired

HEAP IMPLEMENTATION

```
Heap::Heap()
{
    Count = 0;
    for (int index = 0; index <= MAX_HEAP_SIZE; index++)
       heap[index] = -1;
}
```

HEAP IMPLEMENTATION

```
Heap::Heap(const Heap & h)
{
    Count = h.Count;
    for (int index = 0; index <= MAX_HEAP_SIZE; index++)
       heap[index] = h.heap[index];
}
```

HEAP IMPLEMENTATION

```
bool Heap::Insert(int num)
{
    // Check for full heap
    if (IsFull())
        return false;

    // Calculate parent and child indices
    Count++;
    int child = Count;
    int parent = child / 2;
```

New child node created to right of current rightmost leaf node

HEAP IMPLEMENTATION

```
    // Shuffle small values down the heap
    while ((child > 1) && (heap[parent] < num))
    {
       heap[child] = heap[parent];
       child = parent;
       parent = child / 2;
    }

    // Insert new entry in heap
    heap[child] = num;
    return true;
}
```

We make room for new data value much like insertion sort

We store new data value in the correct location in the heap

HEAP IMPLEMENTATION

```
bool Heap::Remove(int &num)
{
    // Check for empty heap
    if (IsEmpty())
       return false;

    // Save value of top of heap
    num = heap[1];

    // Swap last value into root position
    heap[1] = heap[Count];
    heap[Count] = -1;
    Count--;
```

We move last node to the root position and decrease heap size

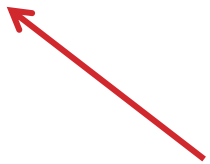
HEAP IMPLEMENTATION

```
// Shuffle data down the heap
int parent = 1;
int largest = 0;
while (parent != largest)
{
    // Compare nodes
    largest = parent;
    int left = parent * 2;
    int right = left + 1;
    if ((left <= Count) && (heap[left] > heap[largest]))
       largest = left;
    if ((right <= Count) && (heap[right] > heap[largest]))
       largest = right;
```

We save index of largest node (parent, left or right)

HEAP IMPLEMENTATION

```
        // Swap data values if needed
        if (parent != largest)
        {
          int temp = heap[parent];
          heap[parent] = heap[largest];
          heap[largest] = temp;
          parent = largest;
          largest = 0;
        }
    }
    return true;
}
```



We swap data values and update the parent index to "move down" the heap before next iteration of while loop

HEAP IMPLEMENTATION

```
void Heap::Print()
{
    for (int index = 1; index <= Count; index++)
        cout << heap[index] << " ";
    cout << endl;
}
```

HEAP IMPLEMENTATION

```
bool Heap::IsEmpty()
{
    return (Count == 0);
}

bool Heap::IsFull()
{
    return (Count == MAX_HEAP_SIZE);
}
```

HEAPS

TESTING THE HEAP

TESTING THE HEAP

```
int main()
{
    // Create heap
    Heap test;

    // Insert data into the heap
    const int SIZE = 15;
    for (int index = 0; index < SIZE; index++)
    {
        // Insert into the heap
        int Value = rand() % SIZE;
        cout << "Insert " << Value << endl;
        bool result = test.Insert(Value);

        // Print heap;
        test.Print();
    }
    cout << endl;
```

Insert random data values into the heap

TESTING THE HEAP

```
    // Remove data from heap
    for (int index = 0; index < SIZE; index++)
    {
        // Remove from the heap
        int Value = 0;
        bool result = test.Remove(Value);
        cout << "Remove " << Value << endl;

        // Print heap;
        test.Print();
    }
    return 0;
}
```

Remove sequence of values from the heap

TESTING THE HEAP

```
Insert 13
13
Insert 1
13 1
Insert 12
13 1 12
Insert 10
13 10 12 1
Insert 8
13 10 12 1 8
Insert 10
13 10 12 1 8 10
Insert 1
13 10 12 1 8 10 1
Insert 12
13 12 12 10 8 10 1 1
```

Inserting a sequence of random values into the heap

Notice how largest value is always in root location

TESTING THE HEAP

```
Insert 9
13 12 12 10 8 10 1 1 9
Insert 1
13 12 12 10 8 10 1 1 9 1
Insert 2
13 12 12 10 8 10 1 1 9 1 2
Insert 7
13 12 12 10 8 10 1 1 9 1 2 7
Insert 5
13 12 12 10 8 10 1 1 9 1 2 7 5
Insert 4
13 12 12 10 8 10 4 1 9 1 2 7 5 1
Insert 8
13 12 12 10 8 10 8 1 9 1 2 7 5 1 4
```

TESTING THE HEAP

```
Remove 13
12 10 12 9 8 10 8 1 4 1 2 7 5 1
Remove 12
12 10 10 9 8 7 8 1 4 1 2 1 5
Remove 12
10 9 10 5 8 7 8 1 4 1 2 1
Remove 10
10 9 8 5 8 7 1 1 4 1 2
Remove 10
9 8 8 5 2 7 1 1 4 1
Remove 9
8 5 8 4 2 7 1 1 1
Remove 8
8 5 7 4 2 1 1 1
```

Removing sequence of values in decreasing order

Notice how next largest value is swapped into root location

TESTING THE HEAP

```
Remove 8
7 5 1 4 2 1 1
Remove 7
5 4 1 1 2 1
Remove 5
4 2 1 1 1
Remove 4
2 1 1 1
Remove 2
1 1 1
Remove 1
1 1
Remove 1
1
Remove 1
```

HEAPS

HEAP SORT

HEAP SORT

- **Since we can remove values from a heap in descending order, we can easily sort data using a heap**

- **Heap Sort Algorithm:**
  - Insert N unsorted data values into a heap
  - Remove N data values from a heap in decreasing order and fill sorted output array from “right to left”
  - Heap insert and remove are both O(logN) for one value
  - It will take O(N logN) to insert and remove N data values
  - Hence heap sort is O(N logN)

HEAP SORT

```
#include "heap.h"

void heap_sort(int data[], int count)
{
    // Insert random data
    Heap heap;
    for (int i = 0; i < count; i++)
       heap.Insert(data[i]);

    // Remove sorted data
    for (int i = count-1; i >= 0; i--)
       heap.Remove(data[i]);
}
```

HEAP SORT

```
Enter number of data values: 100
Enter range of data values: 100

unsorted
83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36
11 68 67 29 82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42
29 73 21 19 84 37 98 24 15 70 13 26 91 80 56 73 62 70 96 81
5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64 43 50
87 8 76 78 88 84 3 51 54 99 32 60 76 68 39 12 26 86 94 39

sorted
2 3 5 5 8 11 11 12 13 13 14 15 15 19 21 21 22 23 24 24
25 26 26 26 26 27 27 29 29 29 29 30 32 34 35 35 36 36 37 39
39 40 42 43 45 46 49 50 51 54 56 56 57 58 59 60 62 62 62 63
64 67 67 67 67 68 68 69 70 70 72 73 73 76 76 77 78 80 81 82
82 83 84 84 84 86 86 86 87 88 90 91 92 93 93 94 95 96 98 99
```

HEAPS

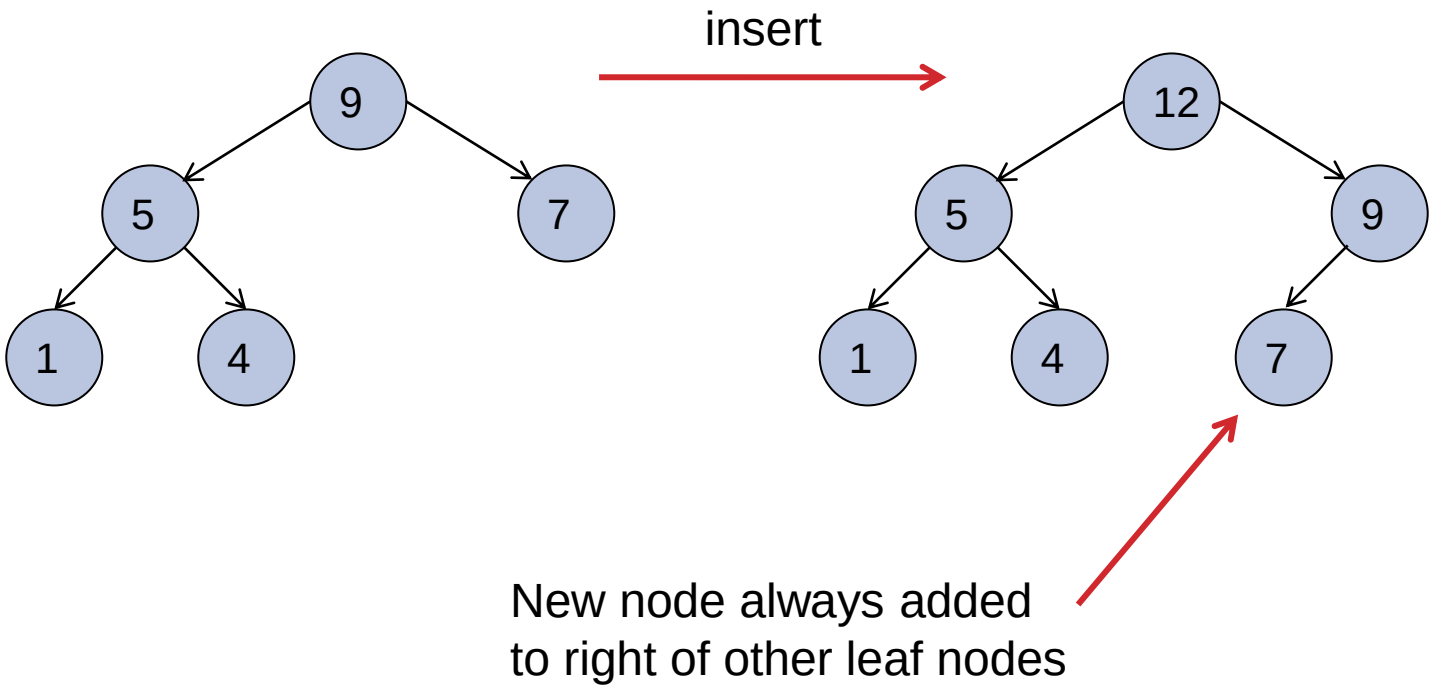
SUMMARY

SUMMARY

- **In this section, we described the design, implementation and use of heaps**
  - A heap is a complete binary tree where the value of the parent node is greater than the values of all children
- **Heaps have two fundamental operations:**
  - Insert – adds a new value into heap, and swaps data to maintain relationship between parent and children
  - Remove – removes largest value from heap, and swaps data to maintain relationship between parent and children

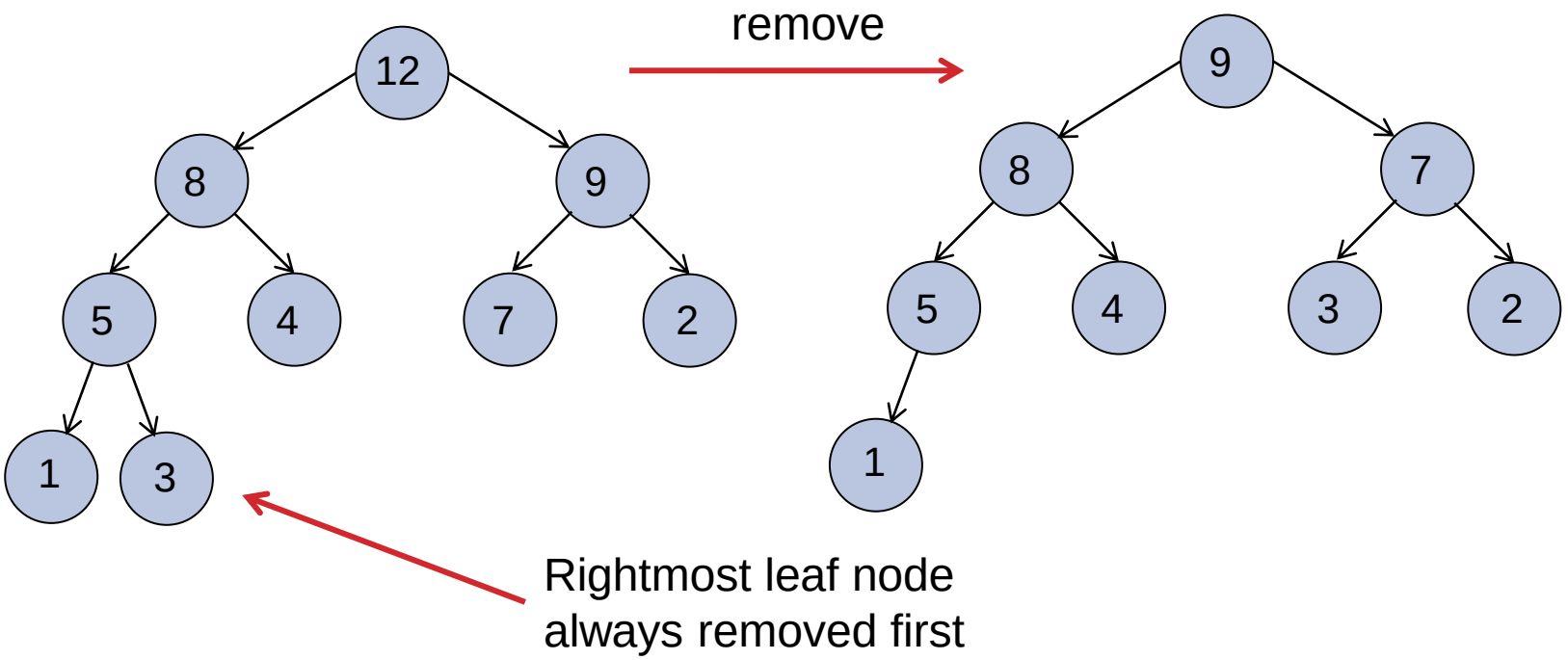
SUMMARY

- **Insert data value 12 into heap:**



SUMMARY

- **Remove largest value 12 from heap:**



SUMMARY

- **Heaps are normally implemented using an array**
  - The root value is stored in location one
  - Parent index = child index / 2
  - Left child index = parent index * 2
  - Right child index = parent index * 2 + 1

- **Heaps are used to implement priority queues in the operating system and other applications**

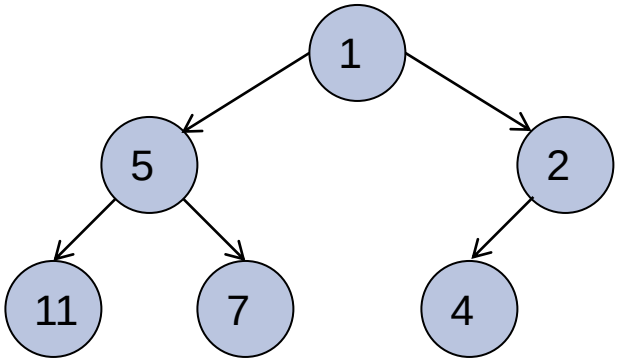
- **Heaps can also be used to implement heap sort by inserting and removing N values in O(N logN) time**

SUMMARY

- A standard heap is also known as a max heap because it stores the largest value at the root
- A min heap stores data in a complete binary tree with the smallest value at the root
- A min heap is often used in minimization algorithms (like finding the shortest path between nodes in graph)

SUMMARY

- **Option 1 for implementing a min heap:**
  - We could invert the data comparison logic to move smallest values up to the root of the tree during insertion and removal
  - This requires minor modifications to the loops inserting and removing values from the heap (see heap1.cpp)



SUMMARY

- **Option 2 for implementing a min heap:**
  - We can use max heap class but change the sign of the values when they are inserted and when they are removed
  - Finding max of a collection of negative values will really correspond to finding min of their positive values