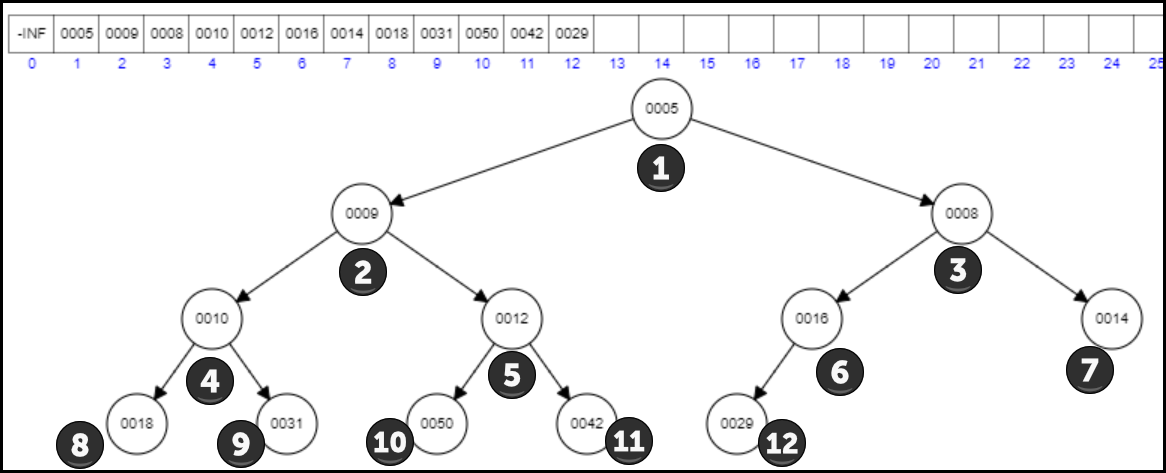
## Heaps

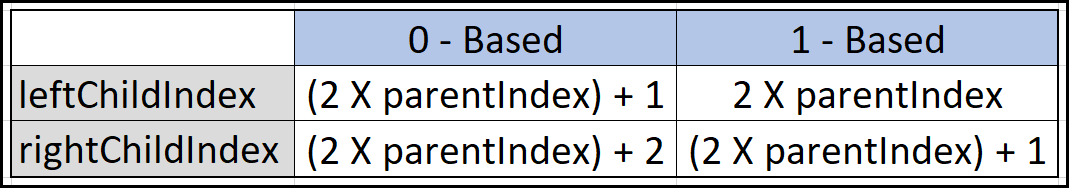
A **Heap** is a special type of binary tree. A heap is a binary tree that meets the following criteria:

1. Is a **complete binary tree**;
   1. A ‘Complete binary tree is defined as a binary tree that has all of its node in from top to bottom and left to right.
2. The value of each node must be **no greater than (or no less than)** the value of its child nodes.

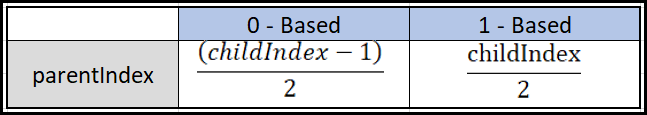
## 1 – Based Min Heap



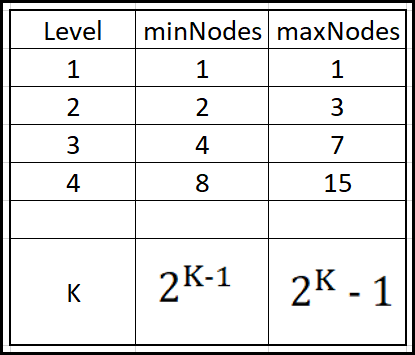
## Calculating Left and Right Child Indices



## Calculating parent index



## Complete Binary Tree (Number of minimum and maximum nodes in each level)



* A CBT of N nodes will have height/level of log2N.

## Min-Priority Queue/Min-Heap

1. insert -logN
2. get -O(1)
3. delete -logN
4. size() -O(1)
5. empty() -O(1)

|  |
| --- |
| #include "CodeLibrary.h"  using namespace *std*;  class MinHeap {  // 0 - Based min-heap!  *vector*<int> cbt;  int parentIndex(int ci) {  return (ci - 1) >> 1; // pi = (ci - 1) / 2.  }  int leftChild(int pi) {  return (pi << 1) + 1; // lci = (pi \* 2) + 1.  }  int rightChild(int pi) {  return (pi << 1) + 2; // rci = (pi \* 2) + 2.  }  // TC: IogN, SC: logN  void heapifyUpRecurssive(int childIdx) {  if (childIdx == 0) return;  int pi = parentIndex(childIdx);  if (cbt[pi] > cbt[childIdx]) {  *swap*(cbt[pi], cbt[childIdx]);  heapifyUpRecurssive(pi);  }  }  // TC: logN, SC: 1  void heapifyUpIterative(int childIdx) {  while (childIdx != 0) {  int pi = parentIndex(childIdx);  if (cbt[pi] < cbt[childIdx]) break;  *swap*(cbt[pi], cbt[childIdx]);  }  }  void heapifyDownRecurssive(int pi) {  if (pi >= cbt.*size*()) return;  int lci = leftChild(pi);  int rci = rightChild(pi);  int minIdx = pi;  if (lci < cbt.*size*() && cbt[lci] < cbt[minIdx])  minIdx = lci;  if (rci < cbt.*size*() && cbt[rci] < cbt[minIdx])  minIdx = rci;  if (pi != minIdx) {  *swap*(cbt[pi], cbt[minIdx]);  heapifyDownRecurssive(minIdx);  }  }  void heapifyDownIterative(int pi) {  while (true) {  int lci = leftChild(pi);  int minIdx = pi;  if (lci < cbt.*size*() && cbt[lci] < cbt[minIdx])  minIdx = lci;  int rci = rightChild(pi);  if (rci < cbt.*size*() && cbt[rci] < cbt[minIdx])  minIdx = rci;  if (pi == minIdx) break;  *swap*(cbt[pi], cbt[minIdx]);  pi = minIdx;  }  }  public:  void insert(int x) {  cbt.*push\_back*(x); // Add to end!  int idx = cbt.*size*() - 1;  heapifyUpRecurssive(idx);  }  int get() {  return cbt[0];  }  void remove() {  cbt[0] = cbt[cbt.*size*() - 1]; // Copy last value to 0th index  cbt.*pop\_back*(); // Removes last value  heapifyDownIterative(0);  }  int size() {  return cbt.*size*();  }  bool empty() {  return cbt.*size*() == 0;  }  };  int main(void)  {  MinHeap a;  a.insert(5); a.insert(9); a.insert(8); a.insert(10);  a.insert(12); a.insert(16); a.insert(14); a.insert(18);  a.insert(31); a.insert(50); a.insert(42); a.insert(29);  while (!a.empty()) {  *cout* << a.get() << " ";  a.remove();  }  return 0;  } |

## DELETION FROM MIN HEAP

1. Copy the last element to cbt[0].
2. Remove the last element.
3. heapifyDown(0).