```cpp
 /*
    This is an assignment I did for my data structures class where I created a templated, k-ary heap
  */
#ifndef HEAP_H
#define HEAP_H
#include <functional>
#include <stdexcept>
#include <algorithm>
#include <utility>
#include <iostream>
#include <vector>

template <typename T, typename PComparator = std::less<T> >
class Heap
{
public:
  /**
   * @brief Construct a new Heap object
   *
   * @param m ary-ness of heap tree (default to 2)
   * @param c binary predicate function/functor that takes two items
   *          as an argument and returns a bool if the first argument has
   *          priority over the second.
   */
  Heap(int m=2, PComparator c = PComparator());

  /**
  * @brief Destroy the Heap object
  *
  */
  ~Heap();

  /**
   * @brief Push an item to the heap
   *
   * @param item item to heap
   */
  void push(const T& item);

  /**
   * @brief Returns the top (priority) item
   *
   * @return T const& top priority item
   * @throw std::underflow_error if the heap is empty
   */
  T const & top() const;

  /**
   * @brief Remove the top priority item
   *
   * @throw std::underflow_error if the heap is empty
   */
  void pop();

  /// returns true if the heap is empty

  /**
   * @brief Returns true if the heap is empty
   *
   */
  bool empty() const;

    /**
   * @brief Returns size of the heap
   *
   */
  size_t size() const;

private:
  /// Add whatever helper functions and data members you need below
  std::vector<T> items;
  PComparator compare;
  int n;

  // trickle down function
```

```cpp
  void trickleDown(int index);

  // trickle up function
  void trickleUp(int index);

};

template <typename T, typename PComparator>
Heap<T,PComparator>::Heap(int m, PComparator c) {
  compare = c;
  n = m;
}

template <typename T, typename PComparator>
Heap<T,PComparator>::~Heap() {
}

// Add implementation of member functions here
template <typename T, typename PComparator>
size_t Heap<T,PComparator>::size() const{
  return items.size();
}

template <typename T, typename PComparator>
bool Heap<T,PComparator>::empty() const{
  return items.empty();
}

template <typename T, typename PComparator>
void Heap<T,PComparator>::push(const T& item) {
  items.push_back(item);
  trickleUp(items.size()-1);
}



// We will start top() for you to handle the case of
// calling top on an empty heap
template <typename T, typename PComparator>
T const & Heap<T,PComparator>::top() const
{
  // Here we use exceptions to handle the case of trying
  // to access the top element of an empty heap
  if(empty()){
    // ================================
    // throw the appropriate exception
    // ================================
    throw std::underflow_error("The Heap is Empty");
  }
  // If we get here we know the heap has at least 1 item
  // Add code to return the top element
  return items.front();
}


// We will start pop() for you to handle the case of
// calling top on an empty heap
template <typename T, typename PComparator>
void Heap<T,PComparator>::pop()
{
  if(empty()){
    // ================================
    // throw the appropriate exception
    // ================================
    throw std::underflow_error("The Heap is Empty");
  }
  items[0] = items.back();
  items.pop_back();
  trickleDown(0);
}

template <typename T, typename PComparator>
void Heap<T,PComparator>::trickleDown(int index) {
  // creating left and right children
  unsigned int left = (index * n) + 1;
```

```
 unsigned int right = (index * n) + 2;
  // checking leaf nodes
  if (right >= items.size() && (left >= items.size() || items.size() == 0 ) ) {
    return;
  }

  // going through all children
 for (int i = 0; i < n; i++) {
  if (right + i < items.size()) { // checking if right child exists
    if (compare(items[right+i], items[left])) {
     left = right + i;
   }
  }
 }
  // swapping if compare is true and recursive call.
  if (compare(items[left], items[index])) {
   std::swap(items[index], items[left]);
   trickleDown(left);
  }
}

template <typename T, typename PComparator>
void Heap<T,PComparator>::trickleUp(int index) {
  int parent = (index-1)/n; // parent node
  while (parent >= 0 && compare(items[index], items[parent]) ) { // checking if parent is valid
    std::swap(items[parent],items[index]); // swapping
    index = parent;
    parent = (index-1)/n;
  }
}

#endif
```