COIS-2020:

Data Structures and Algorithms

Lab 6: Binary Heap

In this lab, you will be given a simple code for a min Heap, and you’re going to complete some of the methods to handle some edge cases.

Tasks for today:

There are mainly two tasks, and the logic of both of them is provided in their bodies:

1. Fully implement AddItem().
2. Fully implement HeapUp()

Use the following start-up code. The main code and a demo class are provided as .cs files. Use them to test your code.

Once you complete the above two tasks, upload the source code for your MinHeap class to Blackboard.

class BinaryHeap<T> : IEnumerable where T : IComparable

{

private T[] array;

private int count; // Number of elements (nodes)

public BinaryHeap(int size)

{

array = new T[size];

count = 0;

}

public T GetItem(int index)

{

return array[index];

}

private void SetItem(int index, T value)

{

while (index >= array.Length)

Grow(array.Length \* 2);

array[index] = value;

}

private void Grow(int newsize)

{

Array.Resize(ref array, newsize);

}

// Indices of left and right children

private int LeftChildIndex(int pos) { return 2 \* pos + 1; }

private int RightChildIndex(int pos) { return 2 \* pos + 2; }

private int GetParentIndex(int pos) => (pos - 1) / 2;

private T GetRightChild(int pos) => array[RightChildIndex(pos)];

private T GetLeftChild(int pos) => array[LeftChildIndex(pos)];

private T GetParent(int pos) => array[GetParentIndex(pos)];

// "Has" methods to determine if the indices exist

private bool HasLeftChild(int pos) {

if (LeftChildIndex(pos) < count)

return true;

else

return false;

}

private bool HasRightChild(int pos)

{

if (RightChildIndex(pos) < count)

return true;

else

return false;

}

private bool IsRoot(int pos) => pos == 0; // (true if element is root)

// Swap, uh, swaps two values given two indicies. This should be private but I originally had it public for some reason.

private void Swap(int position1, int position2)

{

T first = array[position1];

array[position1] = array[position2];

array[position2] = first;

}

public void AddItem(T value)

{

// This is part of la

//Insert Logic

// If the tree is empty, insert at the bottom (it does that already)

// if not, insert at the end,

// From the end you either need to swap to the root, and keep minheapify (Heapdown)

// or, you should probably implement move up (for lab 6, you need to implement

// for that you run a while loop, check if the current position both isn't the root, and is higher priority than a parent (in this case that probably means it's a lower value)

// if it is, swap with parent, and keep doing that until it's its in the

array[count] = value;

count++;

ReCalculateUp();

}

private void HeapDown()

{

//CompareTo

//this.CompareTo(value) returns < 0 if this < value

//this.CompareTo(value) returns >0 if this > value

int index = 0;

while (HasLeftChild(index))

{

var smallerIndex = LeftChildIndex(index);

if (HasRightChild(index) && (GetRightChild(index).CompareTo(GetLeftChild(index))<0) ) //there's a set of ( ) around the right expression that are redundant but hopefully easier to read

{

smallerIndex = RightChildIndex(index);

}

if (array[smallerIndex].CompareTo(array[index])>0) //If array[smallerindex]>= array[index]

{

break;

}

Swap(smallerIndex, index);

index = smallerIndex;

}

}

private void HeapUp()

{

//get the index of the last item

//loop through the list, comparing the child to the parent.

//if the parent is Greater than the child, swap them.

//if the parent is less than or equal to the child, stop.

//set our index to that of the parent and repeat.

}//end HeapUp

}

}