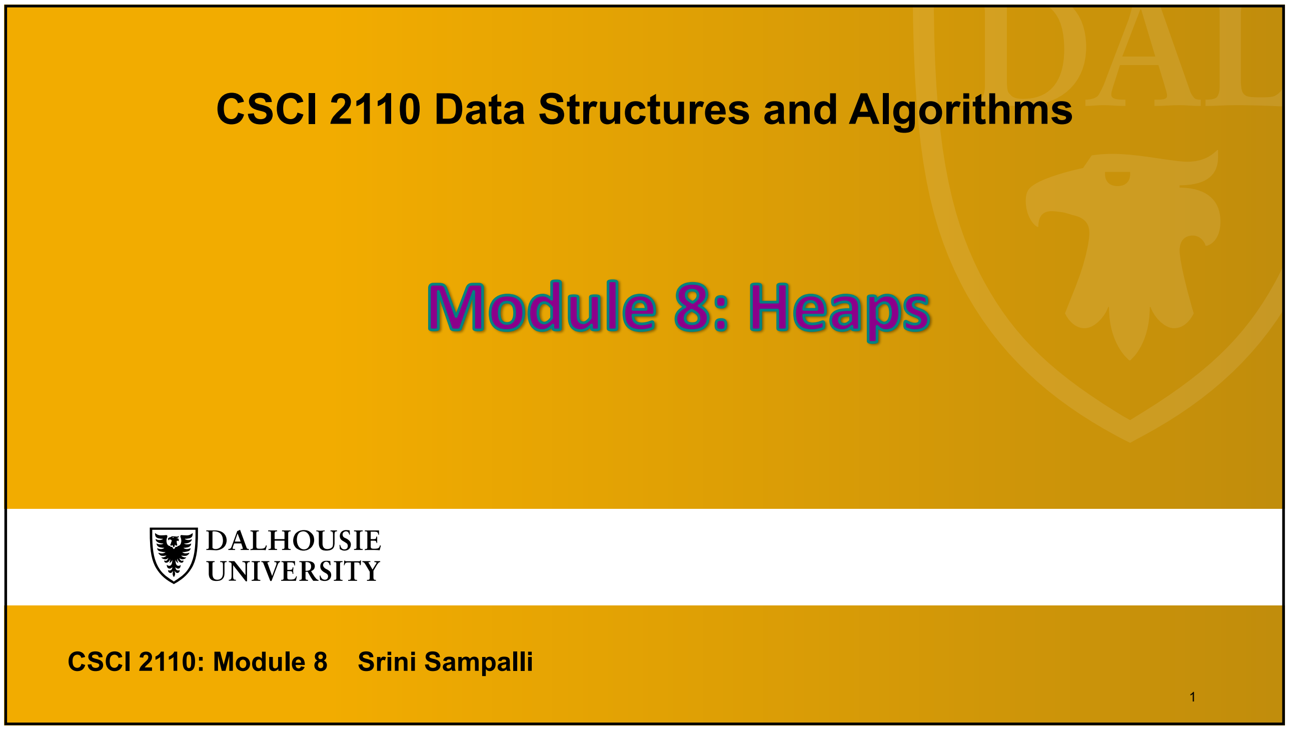


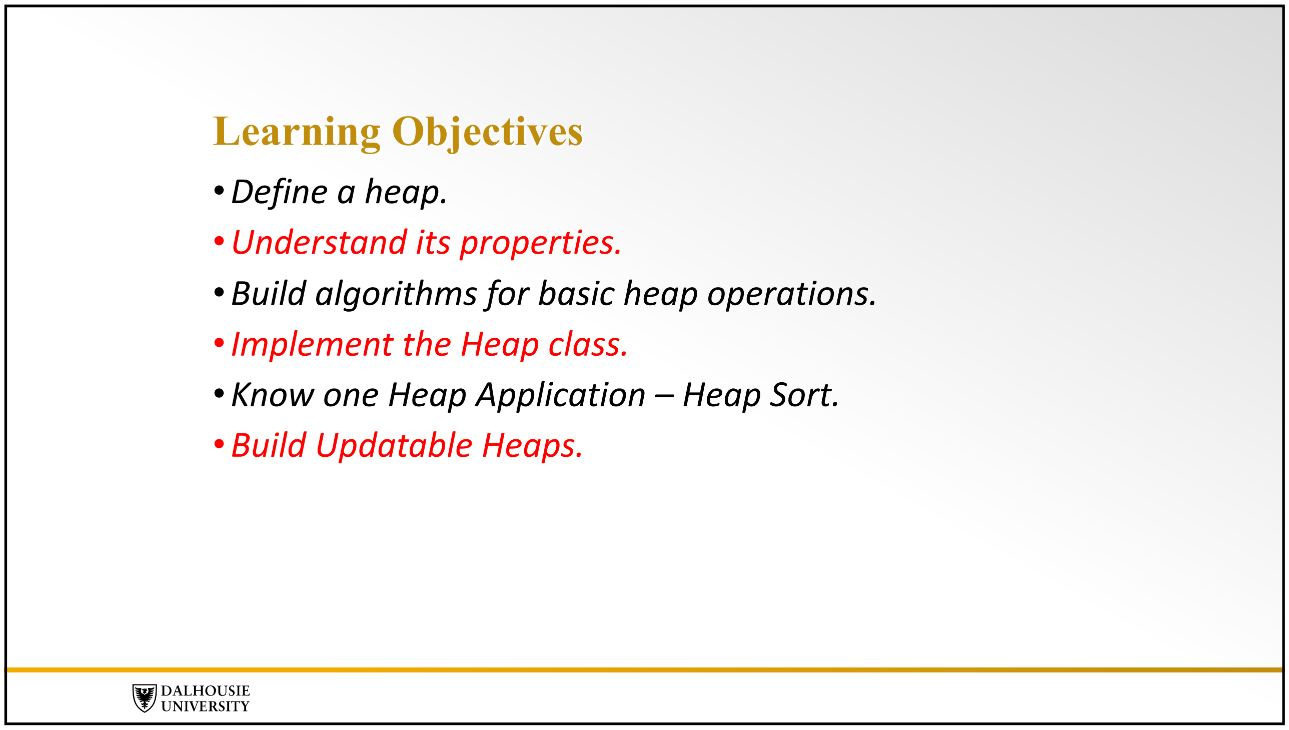
CSCI 2110 Data Structures and Algorithms

Module 8: Heaps

DALHOUSIE UNIVERSITY

CSCI 2110: Module 8 Srini Sampalli

1



Learning Objectives

- *Define a heap.*
- *Understand its properties.*
- *Build algorithms for basic heap operations.*
- *Implement the Heap class.*
- *Know one Heap Application – Heap Sort.*
- *Build Updatable Heaps.*

DALHOUSIE UNIVERSITY

What is a heap?

- Structurally, a heap is a **binary tree**.
- Functionally, a heap is a **priority queue**.
- What is a priority queue?
  - A data structure in which entries have different priorities.
  - The entry with the highest priority is the one to be removed next. **(means that entries will be processed in the order of their priorities)**

DALHOUSIE UNIVERSITY

Some applications of priority queues

- The **emergency room** in a hospital is a quintessential priority queue.
- **Scheduling different processes** in an operating system.
  - Processes arrive at different points of time, and take different amounts of time to finish executing.
  - The OS needs to ensure that all processes get fair treatment.
  - No single process should hog the CPU nor should any process starve for CPU time.
  - At the same time, ***some essential processes need to be executed on a priority basis*** – otherwise, the system won't function.
  - Hence a priority queue is used by the OS for process management.
- **Traffic queueing in network routers** for quality of service – voice traffic, video traffic, and 'other' data traffic are given different priorities.
- **Discrete event simulation** – events are added to the queue with their arrival time used as priority.

DALHOUSIE UNIVERSITY

Dynamic priorities

- In some applications, the priorities may dynamically change when the items are in the queue.
  - A good example is the ER situation.
  - The priority of a patient may be increased according to the severity of his or her condition.
- We will first focus on heaps that do not allow for the priority of an existing entry to be changed.
- We will then discuss updatable heaps in which priorities can change dynamically.

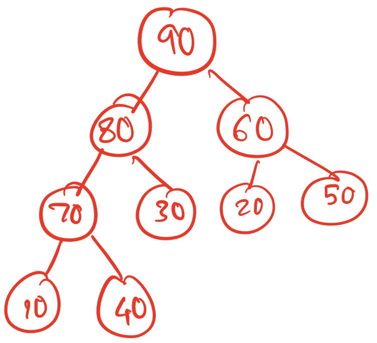
DALHOUSIE UNIVERSITY

Definition of a heap

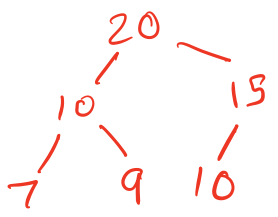
- ***A heap is a complete binary tree in which the key at any node x is greater than or equal to the keys at all nodes in the subtrees rooted at x.***
- ***This means that the largest key is always at the root – easy to get!***
- Recall the definition of a complete binary tree – every level except the last must have the maximum number of nodes. Last level must be filled from left to right.
- ***This ensures that the heap is balanced (it is a good tree!)***
- Duplicate entries are allowed in a heap (for e.g., two items may have the same priority).

DALHOUSIE UNIVERSITY

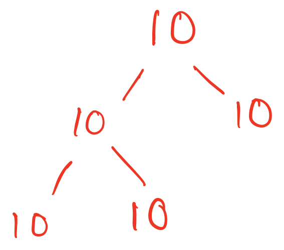
HEAP EXAMPLES



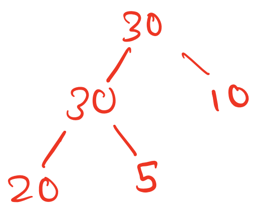
Heap



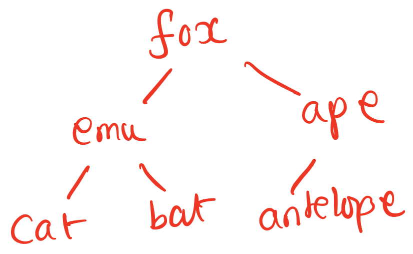
Heap



Heap

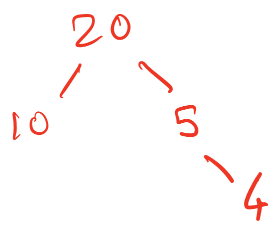


Heap

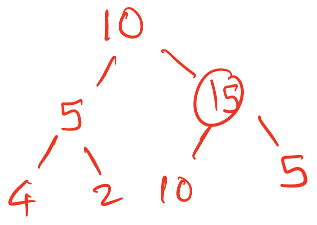


Heap

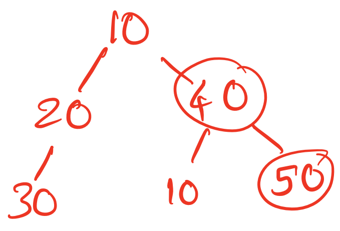
Assumption:
Higher the alphabet
= Higher the priority



Not a heap
Violates structure



Not a heap
Violates order



Not a heap
Violates Structure
& order

HEAP OPERATIONS

1. INSERT A NEW KEY INTO A HEAP

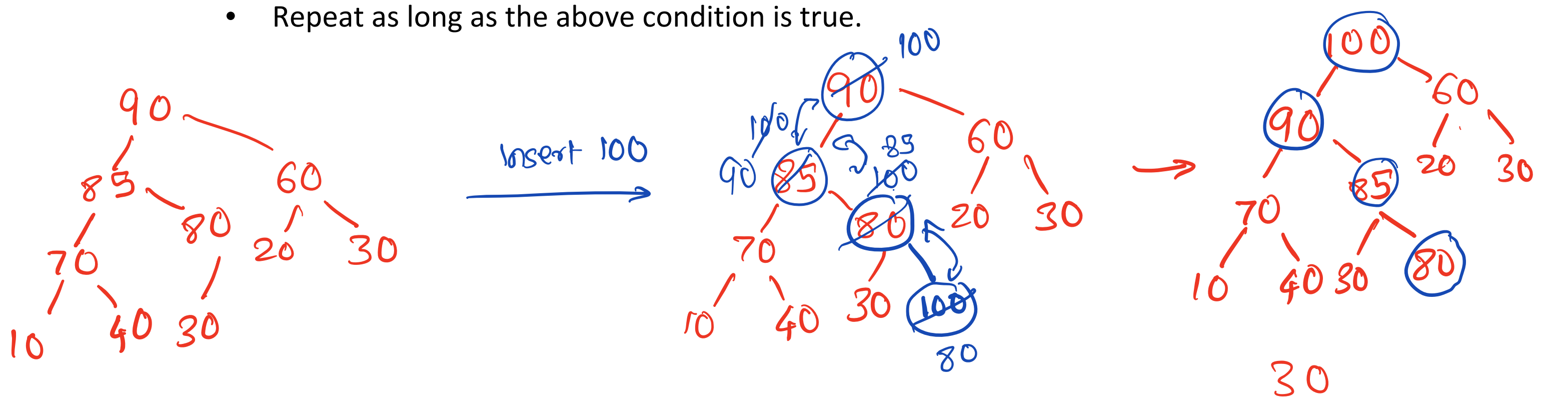
Inserting a new key must maintain both the heap structure and heap order properties.

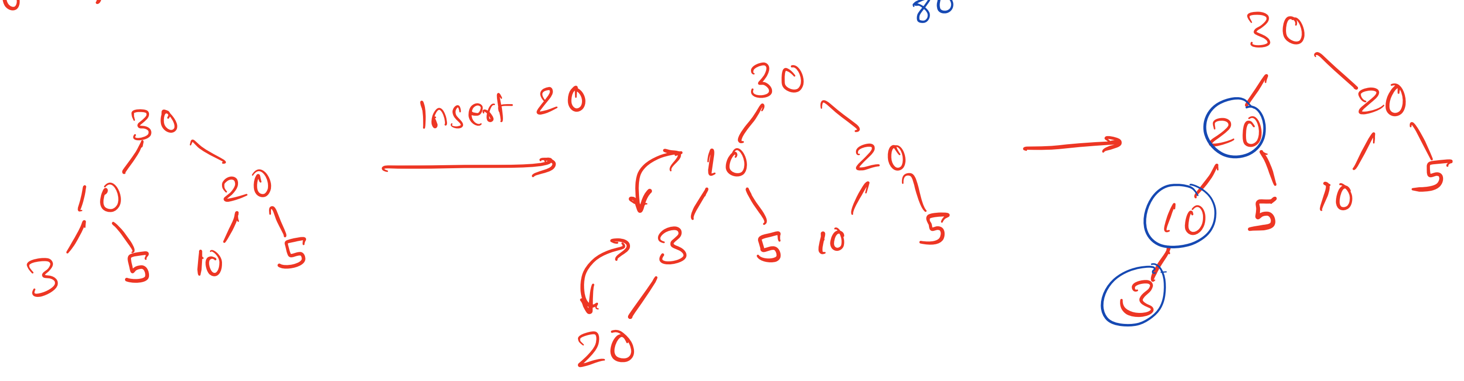
Algorithm:

- First insert the key so that the heap structure is maintained.
- This means it must continue to be a complete binary tree.
- This in turn means that the new key will be a leaf node.
- This might violate the order property.
- *Sift the new key up the tree* until the order property is restored.

Algorithm for sifting up:

- If new key is > key of its parent, exchange current node and parent node.
- Repeat as long as the above condition is true.





SIFTING UP!

2. DELETE

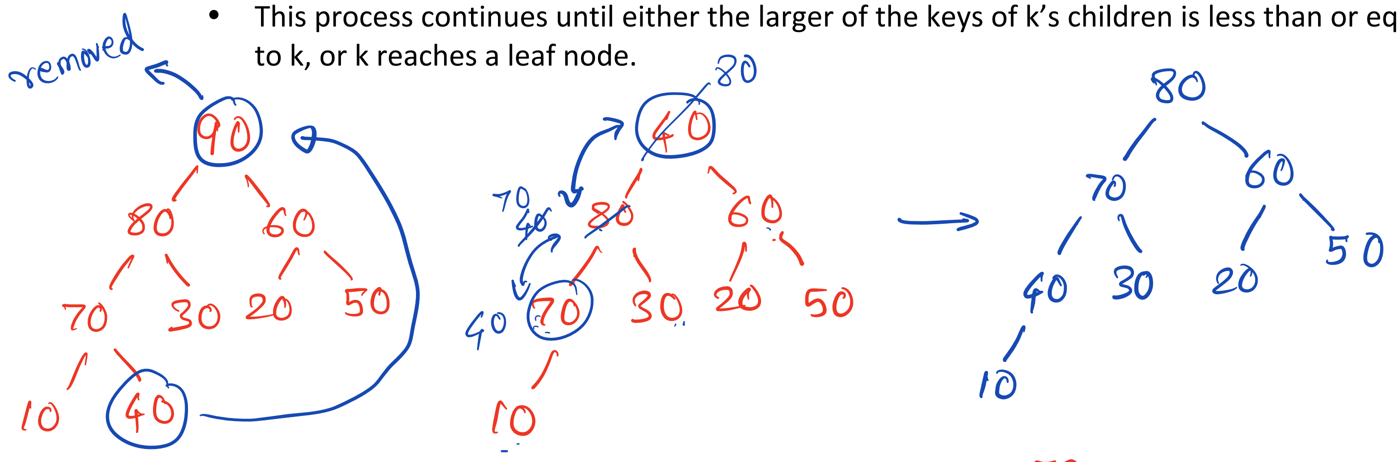
Deletion removes the entry at the top of the heap (also called DeleteMax).
This leaves a vacant spot at the root, hence the heap has to be restored.

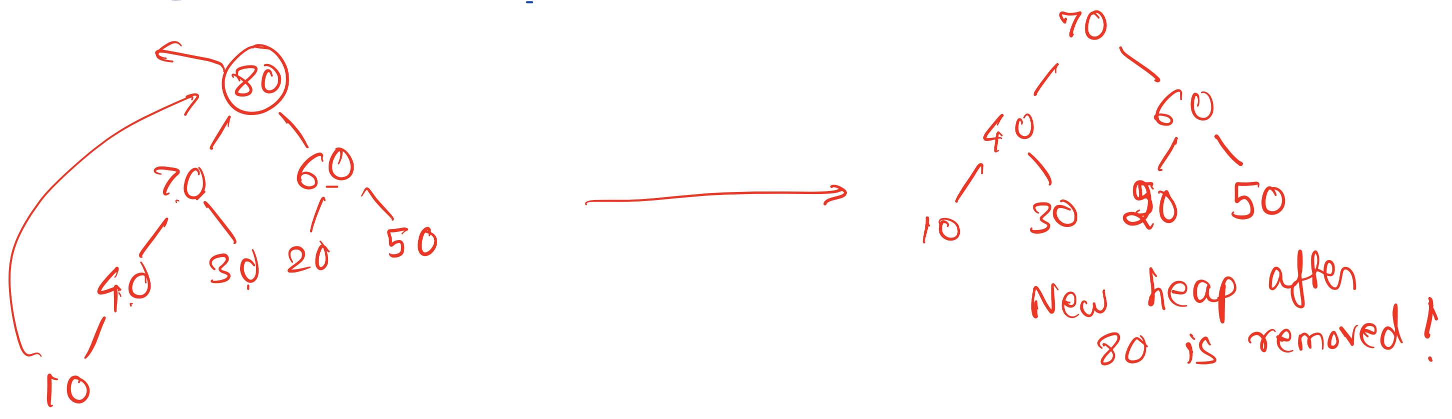
Algorithm:

- Remove the item from the root.
- Replace it with the rightmost node in the last level (this restores the structure property).
- *Sift it down the tree* until the order property is restored.

  Algorithm for sifting down:

  - Let k be the new key in the root node (after it is replaced).
  - The children of k are first compared with each other to determine the larger key.
  - If k is smaller, it is exchanged with the larger key.
  - This process continues until either the larger of the keys of k's children is less than or equal to k, or k reaches a leaf node.



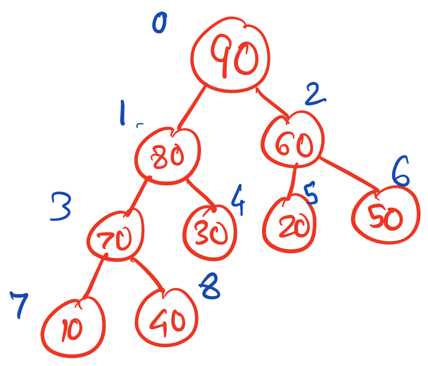


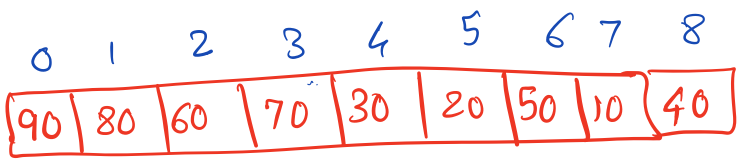
SIFTING DOWN!

HEAP CLASS

What data structure is ideal for storing and processing heap elements? ARRAY LIST!

We will never build any binary tree, but just use simple array list manipulations.





Parent of a node at index i = Node at index (i-1)/2

Children of a node at index i = Nodes at indices 2*i+1 and 2*i+2

Attributes

ArrayList<T> heapList;

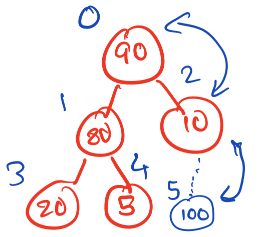
Constructor

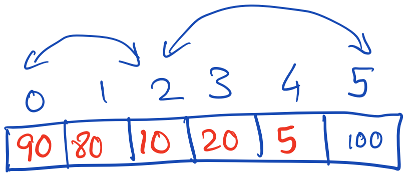
Heap()	Creates an empty heap

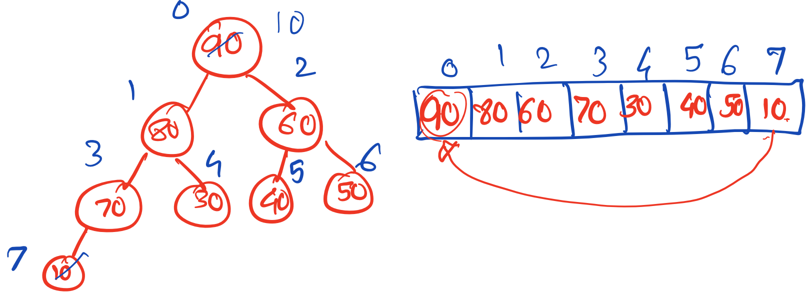
Methods

Name	**What it does**	**Header**	**Price tag (complexity)**
add	Adds an item to the heap	void add (T item)	$O(\log_2 n)$
deleteMax	Deletes the item with the maximum key	T deleteMax ( )	$O(\log_2 n)$
size	Returns the size of the heap	int size( )	$O(1)$
clear	Clears the heap	void clear( )	$O(1)$
isEmpty	Checks if the heap is empty	boolean isEmpty( )	$O(1)$
enumerate	Prints the keys in level order	void enumerate ( )	$O(n)$

```
import java.util.ArrayList;
public class Heap<T extends Comparable<T>>{
        private ArrayList<T> heapList;

        public Heap(){
                heapList = new ArrayList<T>();
        }

        public int size(){
                return heapList.size();
        }
        public boolean isEmpty(){
                return (size()==0);
        }
        public void clear(){
                heapList.clear();
        }

        public void enumerate(){
                System.out.println(heapList);
        }

        public void add(T item)
        {
heapList.add (item);
int index = heapList.size()-1;
int pindex = (index-1)/2;
T parent = heapList.get(pindex); //get the value in the parent node
 while (index >0   && & item.compareTo (parent) >0 )
 {
     heapList.set (index, parent);     // switch the values
     heapList.set (pindex, item);
      index = pindex;                 // get the next
      pindex = (index - 1)/2;            set of values
      parent = heapList.get(pindex);
   }
}
```







```
public T deleteMax()
{
            if (isEmpty())
            {
                        System.out.println("Heap is empty");
                        return null;
            }
            else
            {
                        T ret = heapList.get(0); //root has the item to be returned

                        T item = heapList.remove(heapList.size()-1); //remove the last item

                        if (heapList.size()==0)                          //if the heap now has no items
                                    return ret;

                        heapList.set(0,item); //set it as the root
                        int index, lIndex, rIndex, maxIndex;
                        T maxChild;
                        boolean found=false;

                        index =0;
                        lIndex = index*2+1;
                        rIndex = index*2+2;

                        while(!found)
                        {
                                    //case 1: node has two children
                        if (lIndex < size() && rIndex() < size())
                        { if (heapList.get(lIndex).compareTo(heapList.get(rIndex))
                                                                                  > 0)
                          {    maxChild = heapList.get(lIndex);
                               maxIndex = lIndex;
                          }
                          else
                          {    maxChid = heapList.get(rIndex);
                               maxIndex = rIndex;
                          }
                          // sift down if necessary
                          if (item.compareTo(maxChild) < 0)
                          {    heapList.set(maxIndex, item);
                               heapList.set(index, maxChild);
                               index = maxIndex;
                          }
                          else
                               found = true;
                        }
```

```
                    //case 2: node has only left child
                    else if (lIndex<size())
                    {

                              if (item.compareTo(heapList.get(lIndex))<0)
                              {
                                        heapList.set(lIndex,item);
                                        heapList.set(index,heapList.get(lIndex));
                                        index = lIndex;
                              }
                              else
                                        found = true;
                    }
                    //case 3: only right child – this case does not occur since it is a complete binary tree

                    //case 4: no children
                    else
                              found = true;


                    lIndex = index*2+1;
                    rIndex = index*2+2;
          }


          return ret;
     }
  }
}
```

```
import java.util.Scanner;
public class HeapDemo
{
	public static void main(String[] args)
	{
		Heap<Integer> myHeap = new Heap<Integer>();
		Scanner keyboard = new Scanner(System.in);
		System.out.println("Enter positive integers into the heap (-1 when done): ");
		Integer num = keyboard.nextInt();
		while (num!=-1)
		{
			myHeap.add(num);
			num = keyboard.nextInt();
		}
		myHeap.enumerate();


		System.out.println("Adding nodes 20, 67, 14, 2");
		myHeap.add(20);
		myHeap.enumerate();

		myHeap.add(67);
		myHeap.enumerate();

		myHeap.add(14);
		myHeap.enumerate();

		myHeap.add(2);
		myHeap.enumerate();

		System.out.println("Removing nodes on a priority basis");

		while(!myHeap.isEmpty())
		{
			System.out.println(myHeap.deleteMax());
			myHeap.enumerate();
		}
	}
}
```

HEAP APPLICATION: HEAP SORT

Sorting is easy with a heap!

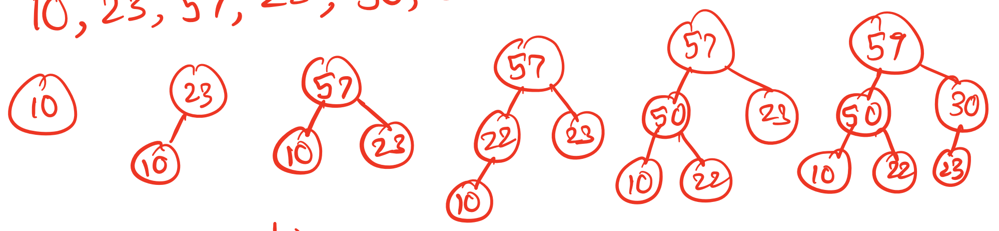
Given n keys

- **Initialize an empty heap.**
- **Read the given set of keys and store them in the heap.** → $n \log_2 n$
- **Repeatedly remove the elements (deleteMax).** → $n \log_2 n$

$\} \; \underline{O(n \log_2 n)}$

Given the keys: 10, 23, 57, 22, 50, 30

Build the heap:



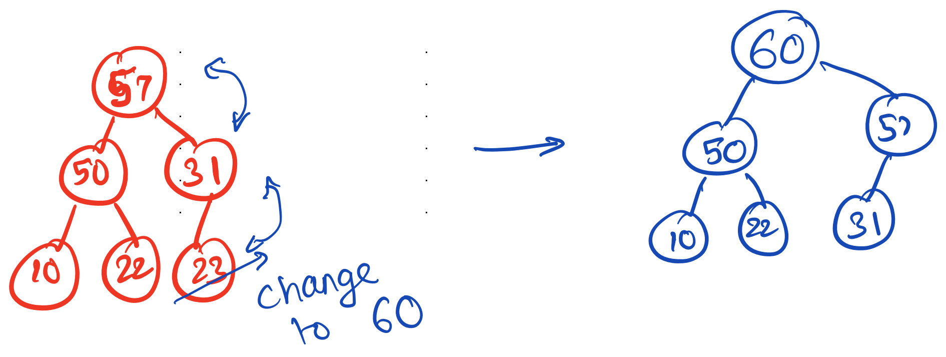
Destroy the heap : keep repeating deleteMax until heap is empty.

57 50 30 23 22 10 ⟶ Sorted!

Note: A min Heap is where the smallest key is always on top.
Can sort in ascending order!

UPDATABLE HEAP:

An updatable heap is a heap in which the priorities can change dynamically.
This means that the nodes in the heap may have to sift up or down when the key values change.
An example application of an updatable heap is the ER waiting room where patients' priorities can change while they are waiting.


change to 60