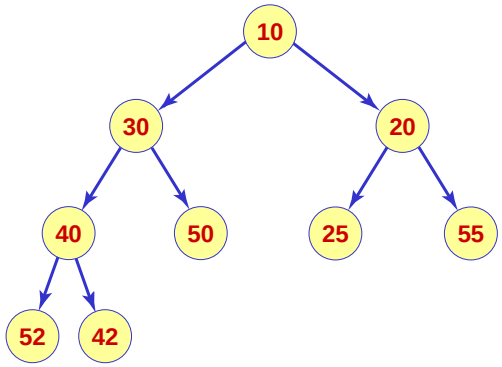
Non-Linear Data Structures

Heaps

Heap Definition

- A heap is a complete binary tree with the property that the value in each node is less than the values in its children nodes.
- The relation, less than, could also be less than or equal, greater than, and greater than or equal.

Heap Example



Heap Operations

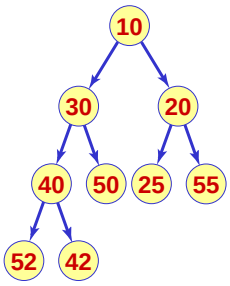
- There is only one important operation on a heap, that is, how to form a heap from a sequence of values.

- HeapCreate:

  Given a complete binary tree with n elements, constructs a heap with n elements.

Heap Implementation

- Heaps can be implemented using a linked structure or arrays.
- Heaps are more efficient and useful when implemented using arrays.
- Example: Remember the complete tree properties?!



Index:	1	2	3	4	5	6	7	8	9
A:	10	30	20	40	50	25	55	52	42

The heap on the left represented as an array

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

- Given an array of n elements, make it a heap:
- The input array is already a complete tree! Why?
- The last n – n/2 elements of the array are leaf nodes, and therefore satisfy the heap conditions. Why?

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

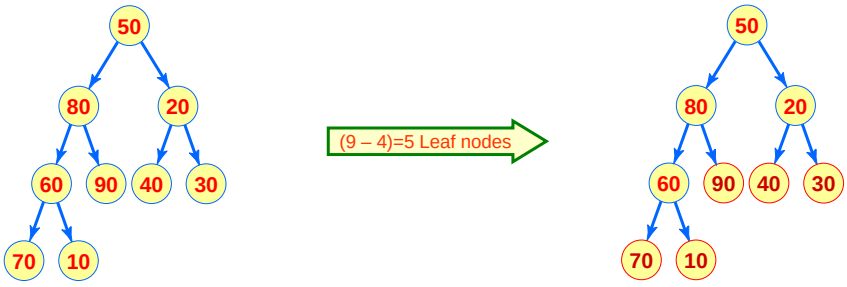
- The following algorithm works back to the root, one node at a time, inserting it into the heap:

1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node. } $O(\log_2 n)$
2. If the path length is not zero: } $O(\log_2 n)$
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done. } $O(\frac{1}{2}n)$

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

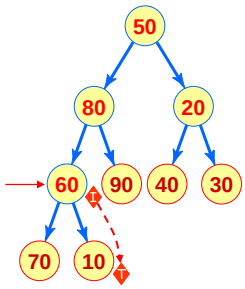
- Example:
  Given the following array, A, convert it into a heap

Index:	1	2	3	4	5	6	7	8	9
A:	50	80	20	60	90	40	30	70	10



Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.

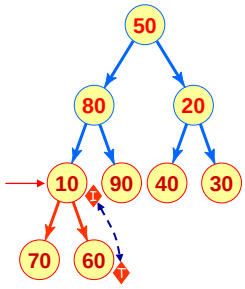


Index:	1	2	3	4	5	6	7	8	9
A:	50	80	20	60	90	40	30	70	10

I T

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.

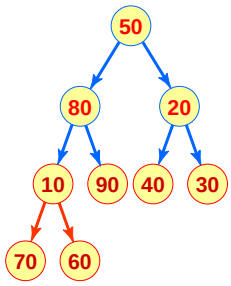


Index:	1	2	3	4	5	6	7	8	9
A:	50	80	20	10	90	40	30	70	60

I T

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

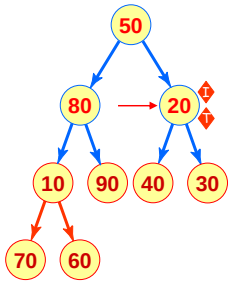
1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.



Index:	1	2	3	4	5	6	7	8	9
A:	50	80	20	10	90	40	30	70	60

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

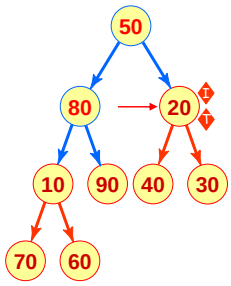
1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.



Index:	1	2	3	4	5	6	7	8	9
A:	50	80	20	10	90	40	30	70	60
			I T						

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

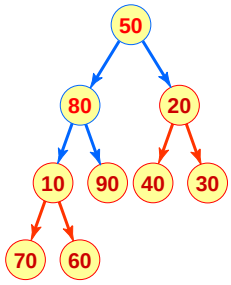
1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.



Index:	1	2	3	4	5	6	7	8	9
A:	50	80	20	10	90	40	30	70	60

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

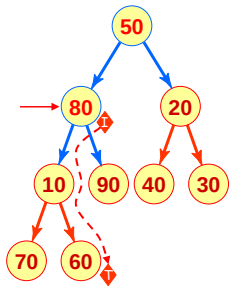
1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.



Index:	1	2	3	4	5	6	7	8	9
A:	50	80	20	10	90	40	30	70	60

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.

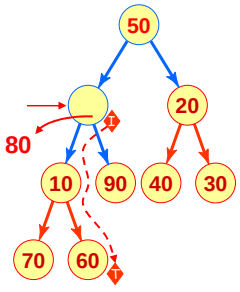


Index:	1	2	3	4	5	6	7	8	9
A:	50	80	20	10	90	40	30	70	60

I T

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.

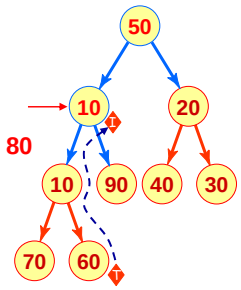


Index:	1	2	3	4	5	6	7	8	9
A:	50		20	10	90	40	30	70	60

I T

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.

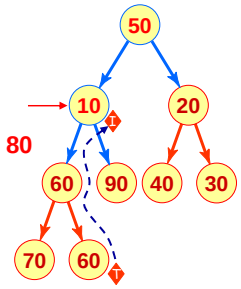


Index:	1	2	3	4	5	6	7	8	9
A:	50	10	20	10	90	40	30	70	60

I T

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.

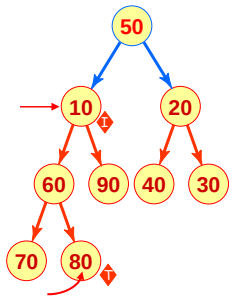


Index:	1	2	3	4	5	6	7	8	9
A:	50	10	20	60	90	40	30	70	60

I T

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.

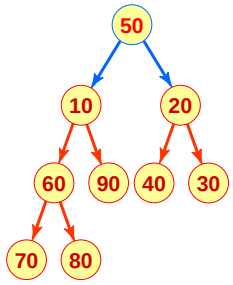


Index:	1	2	3	4	5	6	7	8	9
A:	50	10	20	60	90	40	30	70	80

I T

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

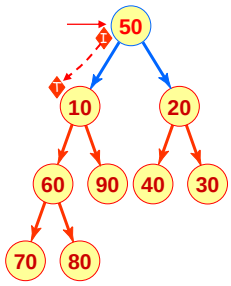
1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.



Index:	1	2	3	4	5	6	7	8	9
A:	50	10	20	60	90	40	30	70	80

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

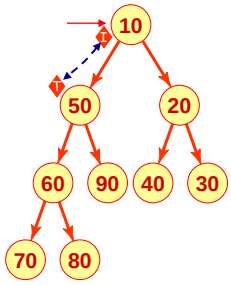
1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.



Index:	1	2	3	4	5	6	7	8	9
A:	50	10	20	60	90	40	30	70	80

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

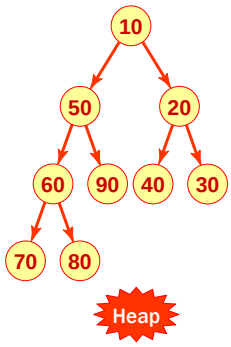
1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.



Index:	1	2	3	4	5	6	7	8	9
A:	10	50	20	60	90	40	30	70	80

Mapping Operation: Heap_create() ... $O(\frac{1}{2}n \log_2 n)$

1. Find a path from the node, I, to be inserted to a leaf node such that the path terminates at a node, T, that has no child smaller than I, always choosing the smaller child of a node.
2. If the path length is not zero:
   - Save the element at node I.
   - Copy the contents of each node in the path into its predecessor's position until node T is done.
   - Insert the contents of node I into node T.
3. Repeat Sifting (steps 1 and 2) until the root is done.



Index:	1	2	3	4	5	6	7	8	9
A:	10	50	20	60	90	40	30	70	80

Heap Properties

- The root node in a "less-than" heap is guaranteed to have the minimum value of all the nodes in the heap.
- The implementation of a heap using arrays is efficient in both space and time.
- The sifting operation is $O(\log_2 n)$.

Heap Applications

Heaps have two important applications:

1. Implementing Priority Queues.
2. Heap Sort.

Heap Application: Implementing Priority Queues

- Remember that each element in a priority queue is associated with a priority value, according to which it is served.
- The serve operation returns the element which have the maximum priority value.
- If the priority queue is arranged as a heap, then the element of maximum priority is at the root of the heap.

Heap Application: Operations of Priority Queues

The Enqueue Algorithm:

1. Add the new element at the end of the array (the heap). The heap may loose its heap property because of this addition. — O(1)
2. Sift up the new element to bring the array back into a heap. — $O(\log_2 n)$

The Serve Algorithm:

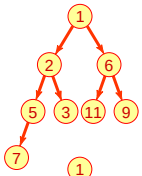
1. Exchange the first element in the array (the root) with the last one. Then remove the last element from the array. The heap may loose its heap property because of this operation. — O(1)
2. Sift down the new root to bring the array back into a heap. — $O(\log_2 n)$

Heap Application: Priority Queues: Enqueue Example

Example:

Given the array A, shown representing a priority queue as a heap.

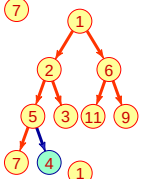
A:	1	2	6	5	3	11	9	7



Insert an element with priority = 4:

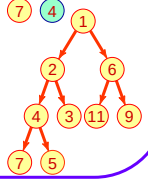
Step1: Add

A:	1	2	6	5	3	11	9	7	4



Step2: Sift

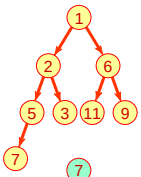
A:	1	2	6	4	3	11	9	7	5



Heap Application: Priority Queues: Serve Example

Example:

Given the array A, shown representing a priority queue as a heap.

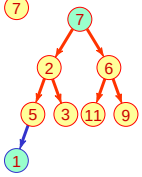


A:	1	2	6	5	3	11	9	7

Serve the highest priority element:

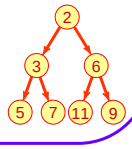
Step1: Exch.

A:	7	2	6	5	3	11	9	1



Step2: Delete & Sift

A:	2	3	6	5	7	11	9



Heap Application: Heap Sort

Heap Sort Algorithm:

Given a linear list, A, of n items, produce a sorted list of them.

1. Convert the list in A into a heap of n items. $O(\frac{1}{2} n \log n)$
2. Repeat the following steps while the heap contains more than one item: $O(n)$
   a. Exchange the first and the last items of the heap. $O(1)$

      The array is now partitioned into two parts:

      i. The last item is in the sorted part.

      ii. The remaining (n -1) items of the heap are in the heap part.

   b. Reform the heap by sifting down the new root in the heap part only. $O(\log n)$

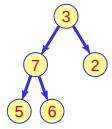
Heap Application: Heap Sort Example

Example:

Given the array A, shown representing a list of items. Sort the list in descending order.

A:

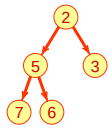
3	7	2	5	6



Make it a heap:

A:

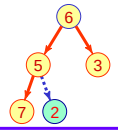
2	5	3	7	6



Exchange first and last items:

A:

6	5	3	7	2

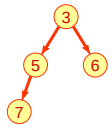


Heap Application: Heap Sort Example (cont.)

Sift the root down:

A:

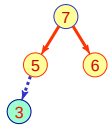
3	5	6	7	2



Exchange first and last items:

A:

7	5	6	3	2



Sift the root down:

A:

5	7	6	3	2

Heap Application:
Heap Sort Example (cont.)

Exchange first and last items:

A:

6	7	5	3	2

Sift the root down:

A:

6	7	5	3	2

Exchange first and last items:

A:

7	6	5	3	2

The List is Sorted.

A:

7	6	5	3	2