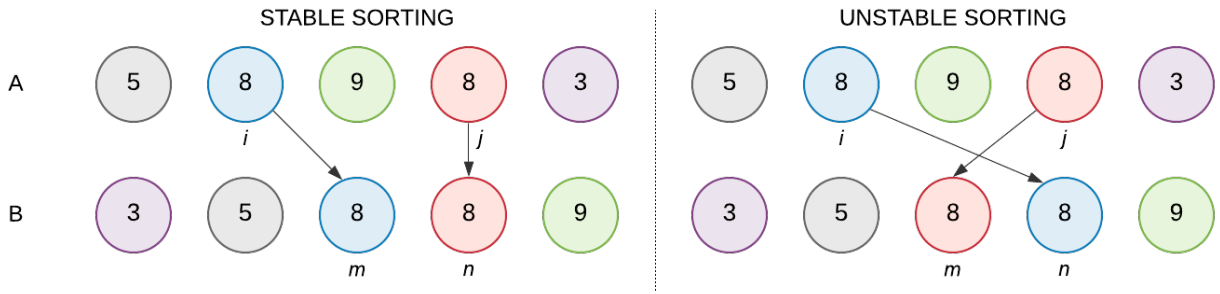
Data Structures:
Sorting: Bubble Sort, Heap Sort, Counting Sort, Radix Sort

YoungWoon Cha
(Slide credits to Won Kim)
Spring 2022

Some Sorting-Related Concepts

- Stable sort



- Indirect sort (indexing)

Stable Sort

- Sorting must keep the relative orders between the elements.

name	age	employer
Lee	30	IBM
Kim	25	LG
Lee	23	LG
Kim	40	Samsung
Chung	28	Samsung

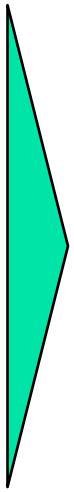
Given table
in employer order

Stable Sort (cont'd)

in employer order

name	age	employer
Lee	30	IBM
Kim	25	LG
Lee	23	LG
Kim	40	Samsung
Chung	28	Samsung

Primary key='name'



in name order

name	age	employer
Chung	28	Samsung
Kim	25	LG
Kim	40	Samsung
Lee	30	IBM
Lee	23	LG

stable

in name order

name	age	employer
Chung	28	Samsung
Kim	40	Samsung
Kim	25	LG
Lee	23	LG
Lee	30	IBM

unstable

Indirect Sort

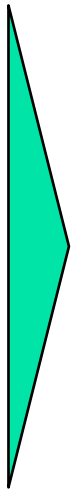
- Direct sort: move records
- Indirect sort: use an index

	name	age	employer
1	Lee	30	IBM
2	Kim	25	LG
3	Lee	23	LG
4	Kim	40	Samsung
5	Chung	28	Samsung

The Table in storage

1	5
2	2
3	4
4	1
5	3

index



name	age	employer
Kim	40	Samsung
Kim	25	LG
Chung	28	Samsung
Lee	23	LG
Lee	30	IBM

Virtual Table

Bubble Sort

Air Bubble in Water

Bubble Sort: Method

- (Not Recommended)
  - (Included just for comparison with Selection Sort)
- Similar to Selection Sort
- On the i^{th} Pass
  - Compare the j^{th} key with the $j+1^{th}$ key, and exchange them if the j^{th} key > $j+1^{th}$ key ($1 <= j <= n-i+1$)
  - Percolate ("bubble") up the largest key among the first n-i keys to the $n-i+1^{th}$ position.
- Examples:
  - https://www.youtube.com/watch?v=nmhjrI-aW5o
  - http://en.wikipedia.org/wiki/Bubble_sort

Bubble Sort: Example

sort the list:
13, 4, 9, 21, 37, 17, 22, 3, 8

pass 1: 4, 13, 9, 21, 37, 17, 22, 3, 8

4, 9, 13, 21, 37, 17, 22, 3, 8

4, 9, 13, 21, 17, 37, 22, 3, 8

4, 9, 13, 21, 17, 22, 37, 3, 8

4, 9, 13, 21, 17, 22, 3, 37, 8

4, 9, 13, 21, 17, 22, 3, 8, 37

Bubble Sort: Example (cont'd)

sort the list:

13, 4, 9, 21, 37, 17, 22, 3, 8

pass 2: 4, 9, 13, 21, 17, 22, 3, 8, 37

4, 9, 13, 17, 21, 3, 8, 22, 37

pass 3: 4, 9, 13, 17, 21, 3, 8, 22, 37

4, 9, 13, 17, 3, 8, 21, 22, 37

pass 7: 4, 3, 8, 9, 13, 17, 21, 22, 37

pass 8: 3, 4, 8, 9, 13, 17, 21, 22, 37

Bubble Sort: Performance Analysis

- $O(n^2)$ (Worst, Average)
  - n = the number of keys in the list
  - O(n) time to swap for each path.
- Best Case: O(n)
  - If the list is already sorted, no swap (O(1))
- Space Complexity: O(1) (in-place sort)
- Stable Sorting
  - No swap for duplicate elements.

Bubble Sort: Example code

```c
// C program for implementation of Bubble sort
#include <stdio.h>

void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

https://www.geeksforgeeks.org/bubble-sort/
https://www.programiz.com/dsa/bubble-sort#:~:text=Bubble%20sort%20is%20a%20sorting,is%20called%20a%20bubble%20sort.

Priority Queue

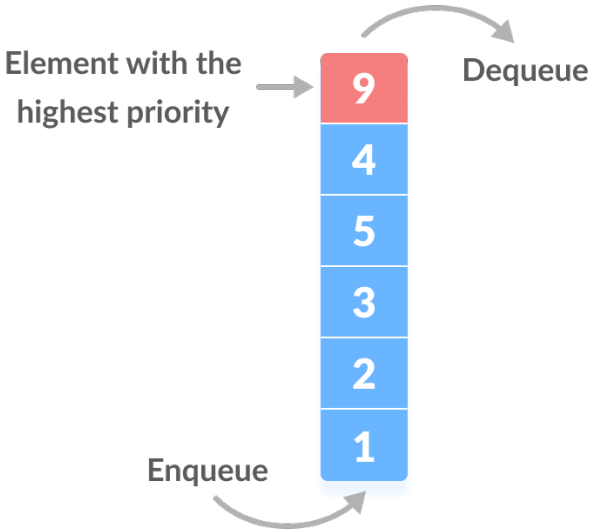
Priority Queue

- Not Priority Queues
  - Stack
    - LIFO (Last-In-First-Out)
    - Each inserted element has a higher priority
  - Queue
    - FIFO (First-In-First-Out)
    - Each inserted element has a lower priority

- Priority Queue
  - A queue in which every element is associated with a “priority”
  - Minimum or Maximum value has a higher priority.

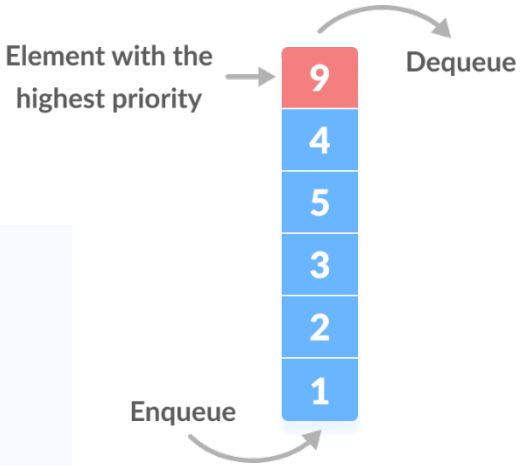
Priority Queue

- Implementations
  - (naive and inefficient)
    - unsorted array, unsorted linked list
    - sorted array, sorted linked list
  - (usually) binary heap



- Operations
  - insert (an element with a priority)
  - remove the highest priority element
    - Pop (get max, get front)
  - peek (just determine the highest priority element)

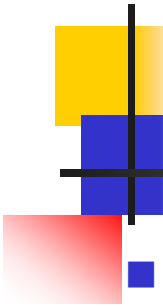
Priority Queue: Comparison



Operations	Peek	Insert	delete
Unsorted Array	O(n)	O(1)	O(n)
Unsorted Linked List	O(n)	O(1)	O(n)
Sorted Array	O(1)	O(n)	O(1)
Sorted Linked List	O(1)	O(n)	O(1)
Binary Heap	O(1)	O(log n)	O(log n)
Binary Search Tree	O(1)	O(log n)	O(log n)

Heap Sort

A Heap



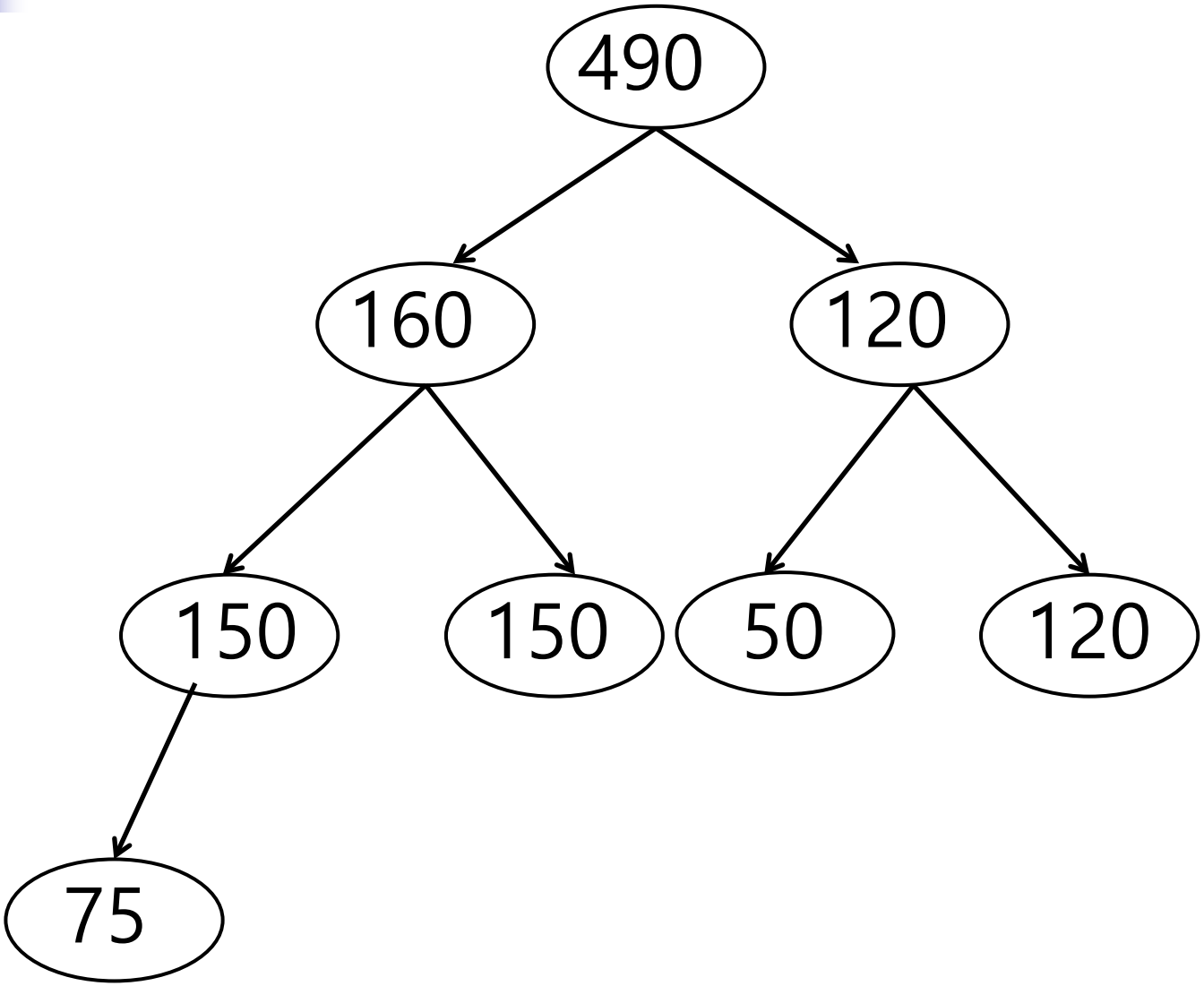
Heap Sort

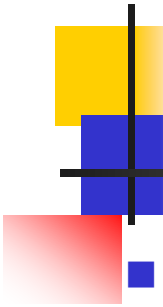
- Use of a Heap
  - Max Heap or Min Heap
- “Min-Max” Sort
- Supplementary Reading
  - http://en.wikipedia.org/wiki/Heapsort

(Binary) Max Heap

- "Complete Binary Tree + Max Tree"
- Max Tree
  - The key in each node is not smaller than any key in its left and right subtrees
  - Not a binary search tree
- Operations
  - Insert a new node
  - Delete the largest node
  - "No search"
- Insertion and deletion require restructuring to maintain the max heap properties.
- May be used to implement a priority queue or heap sort

(Binary) Max Heap: Example

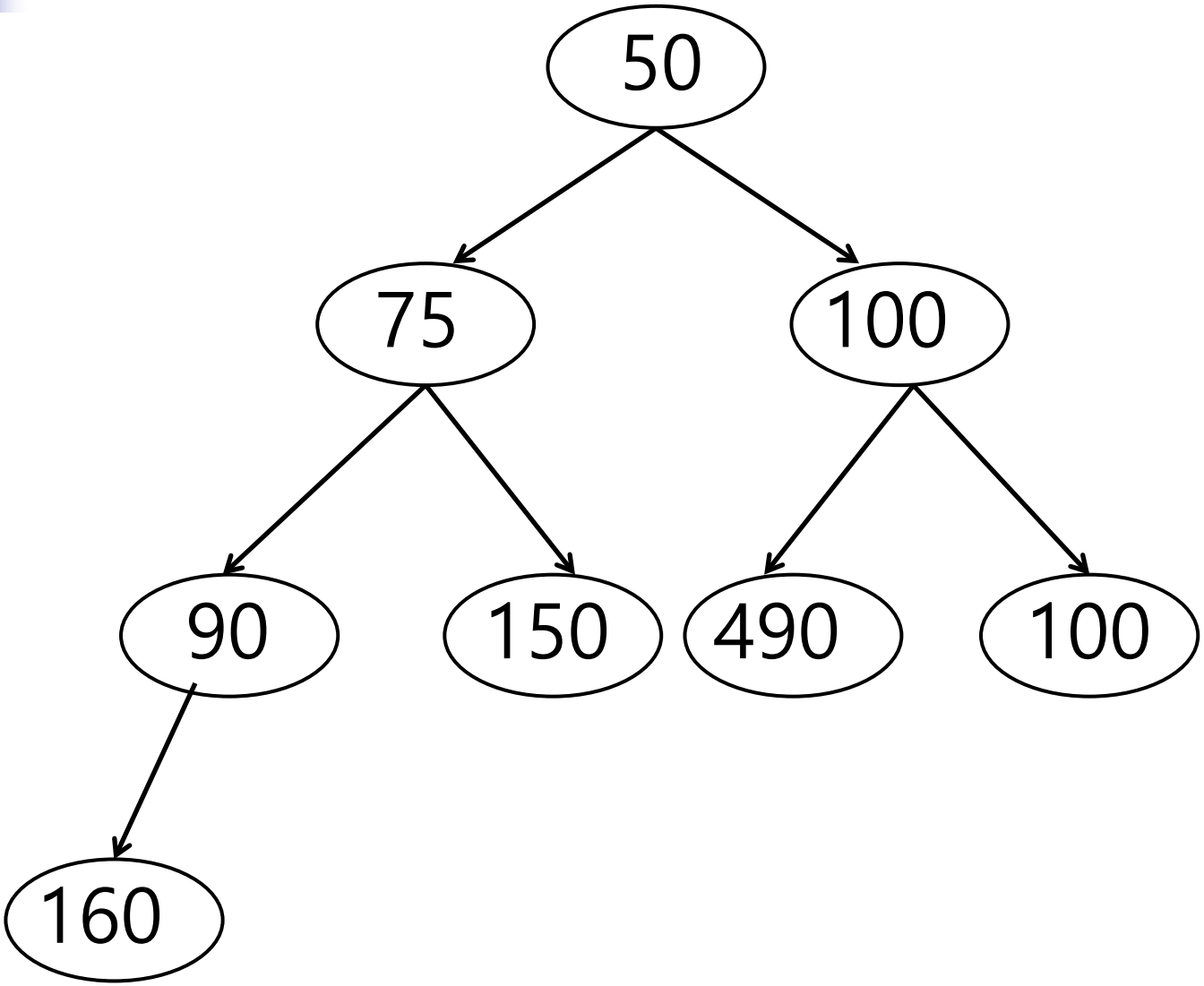




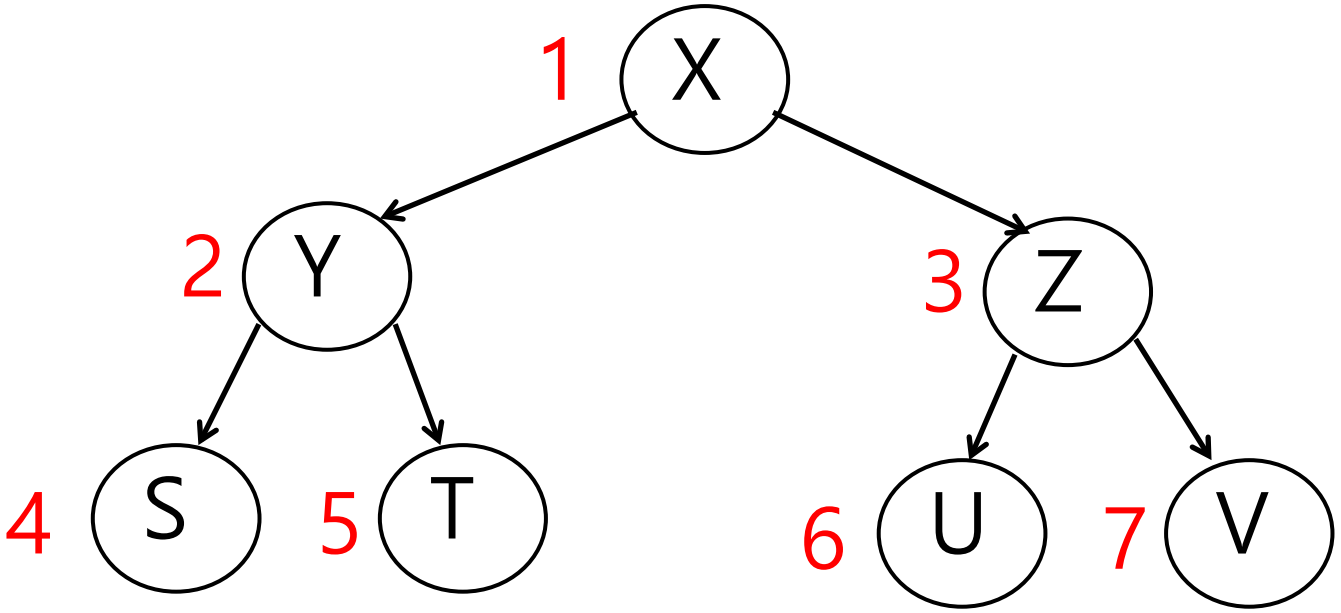
(Binary) Min Heap

- Complete Binary Tree + Min Tree
- Min Tree
  - The key in each node is not larger than any key in its left and right subtrees
- Operations
  - Insert a new node
  - Delete the smallest node

(Binary) Min Heap: Example



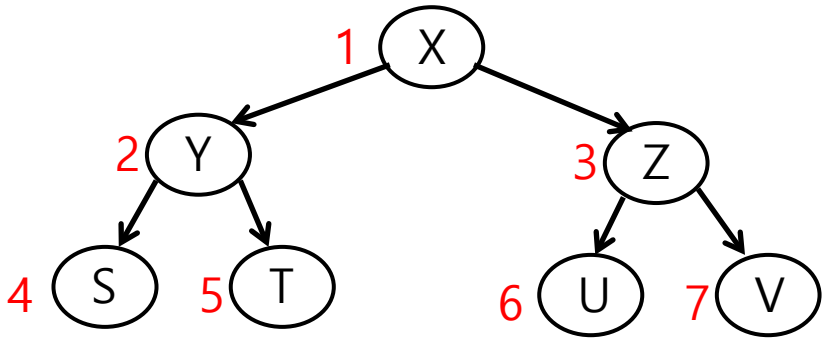
Implementing a Heap Using an Array



1	X
2	Y
3	Z
4	S
5	T
6	U
7	V

Computing the Locations of Nodes

- **For a node with array index i on a full binary tree with n nodes**
  - **Parent of i = ⌈ i/2⌉**
  - **Left child of i = 2i**
  - **Right child of i = 2i + 1**



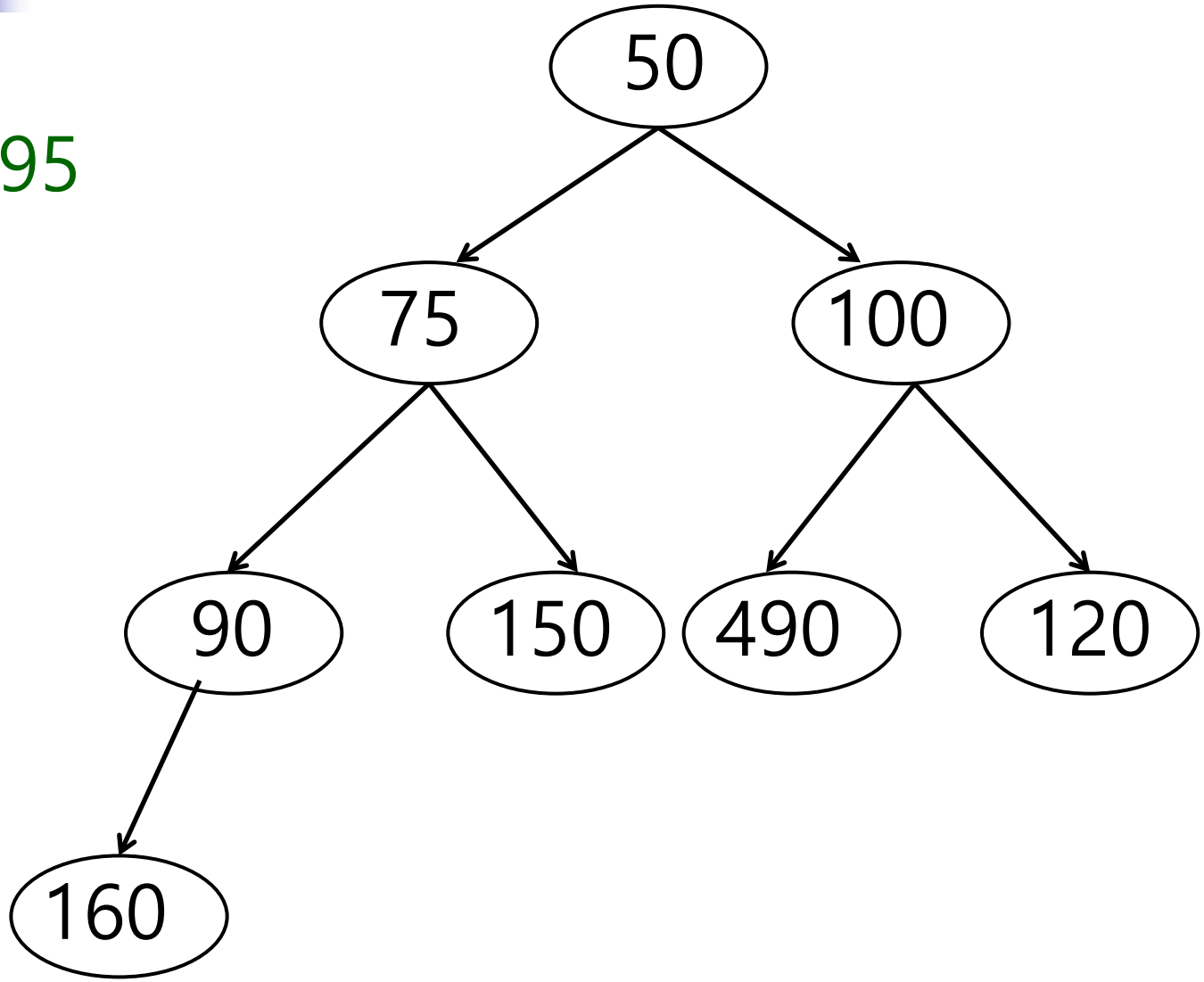
1	X
2	Y
3	Z
4	S
5	T
6	U
7	V

Heap Sort: Method

- "Complete Binary Tree + Max Tree"
- Operations
  - Insert a new node
  - Delete the largest node
  - Insertion and deletion require restructuring to maintain the max heap properties.
- Examples:
  - https://youtu.be/MtQL_ll5KhQ

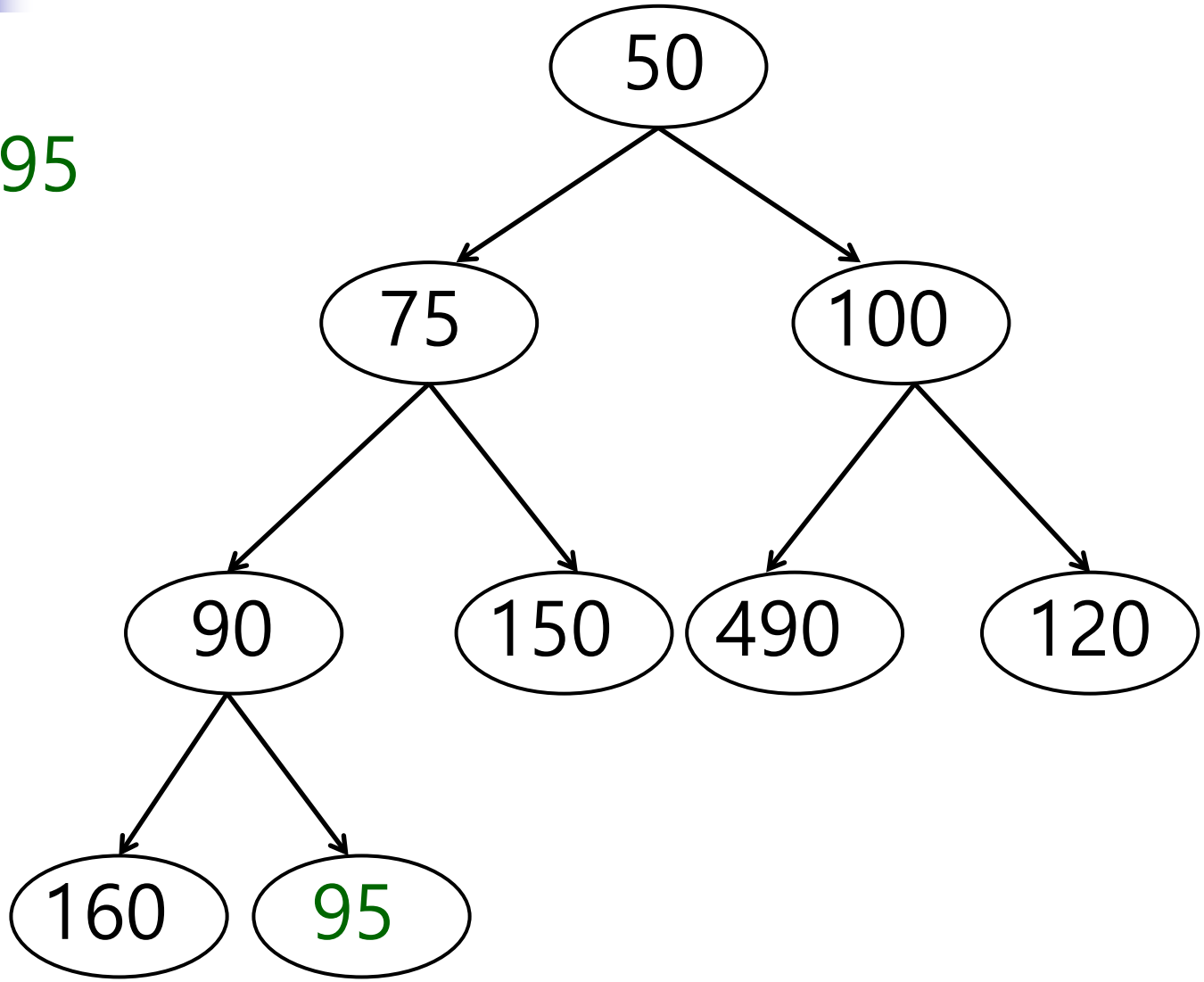
Inserting into a Min Heap: Example 1

95



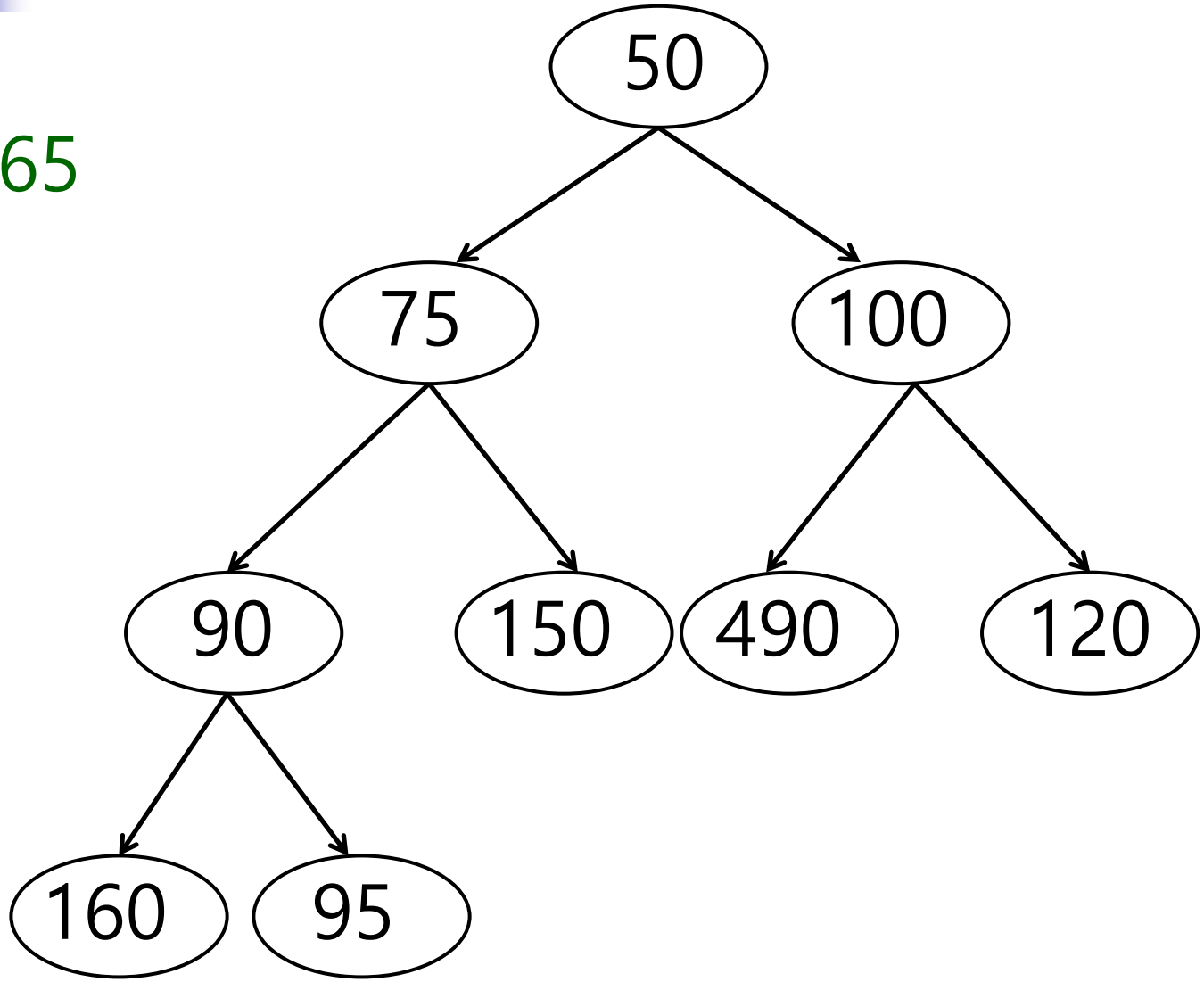
Inserting into a Min Heap: Example 1 (cont'd)

95



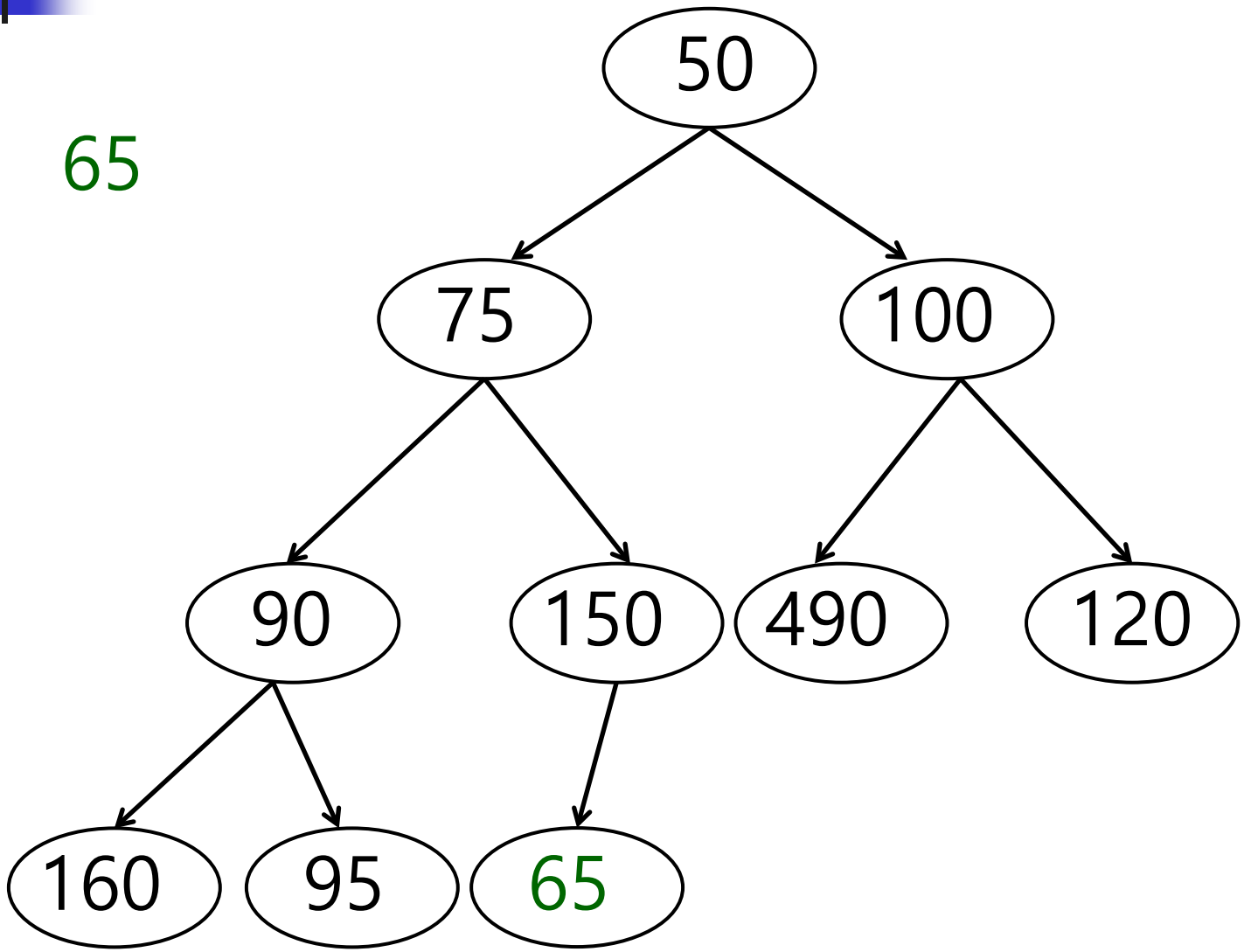
Inserting into a Min Heap: Example 2

65



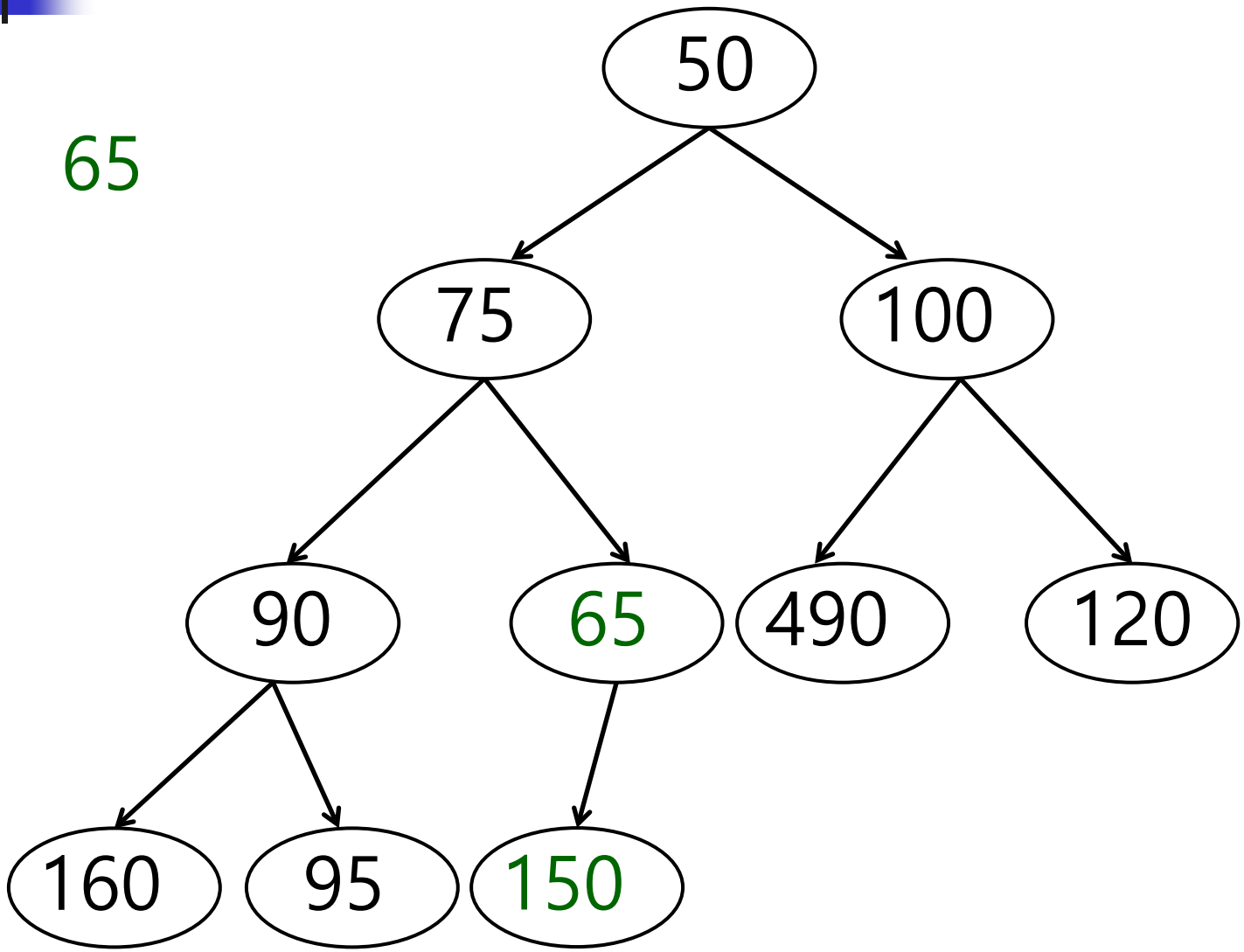
Inserting into a Min Heap: Example 2 (cont'd)

65

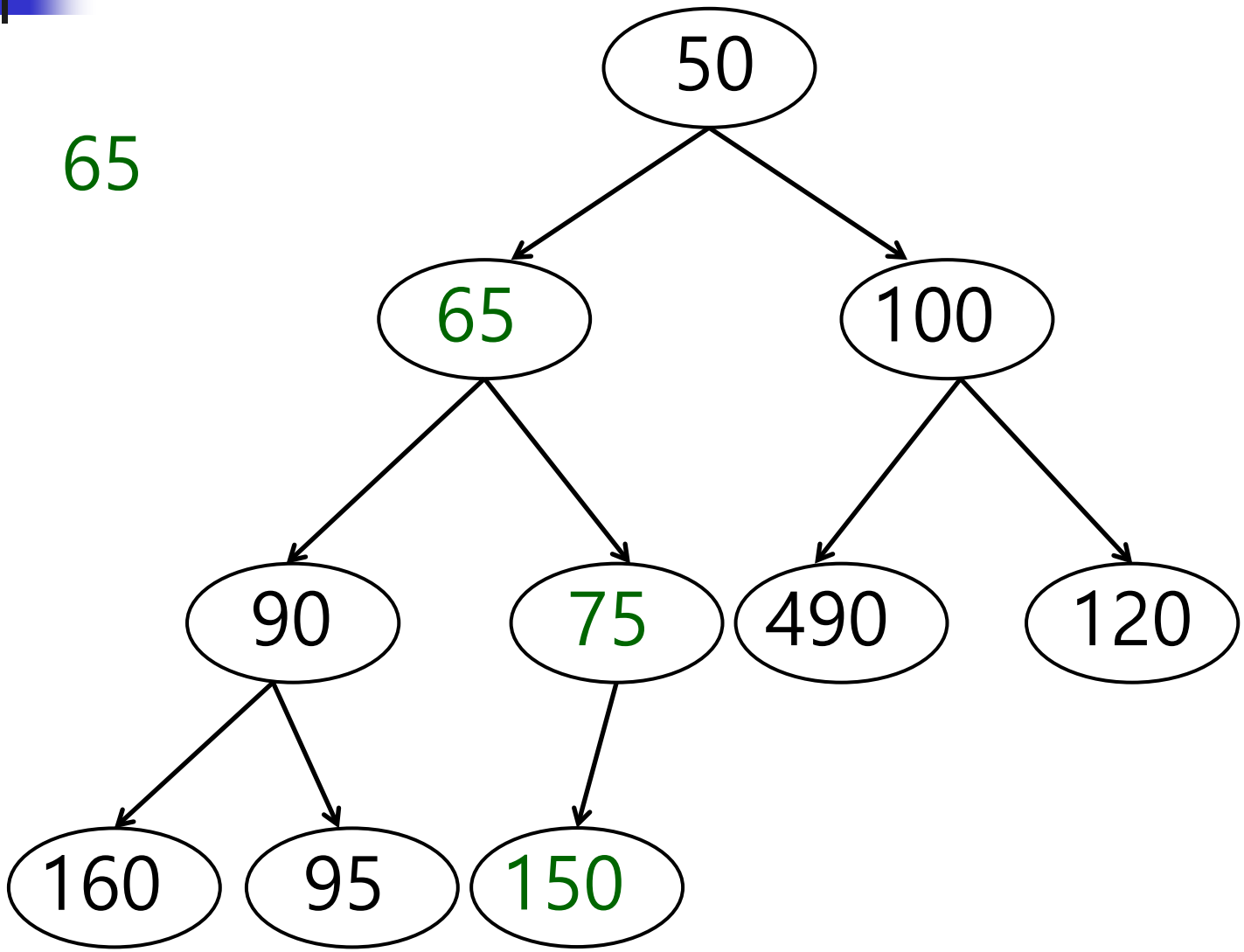


Inserting into a Min Heap: Example 2 (cont'd)

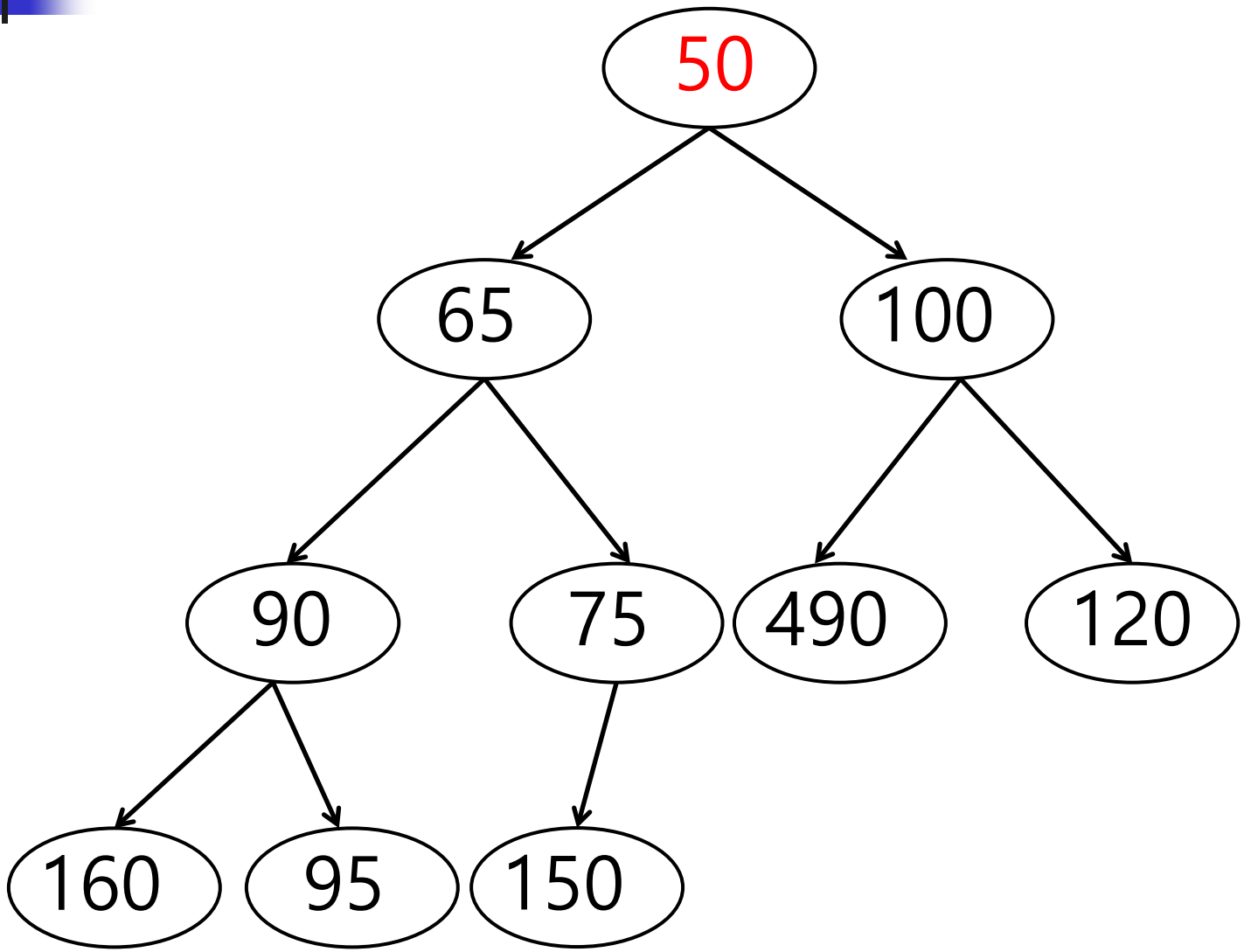
65



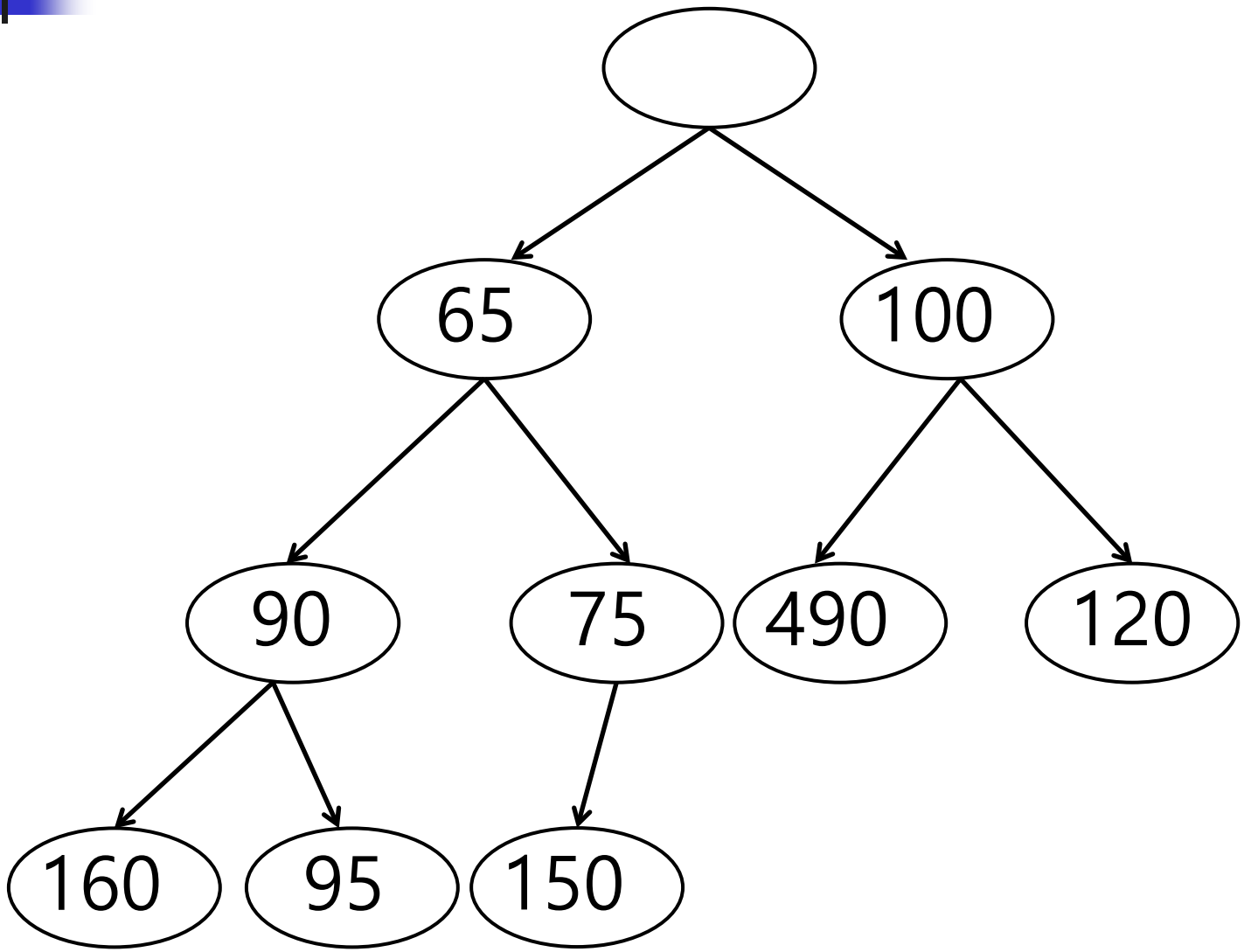
Inserting into a Min Heap: Example 2 (cont'd)



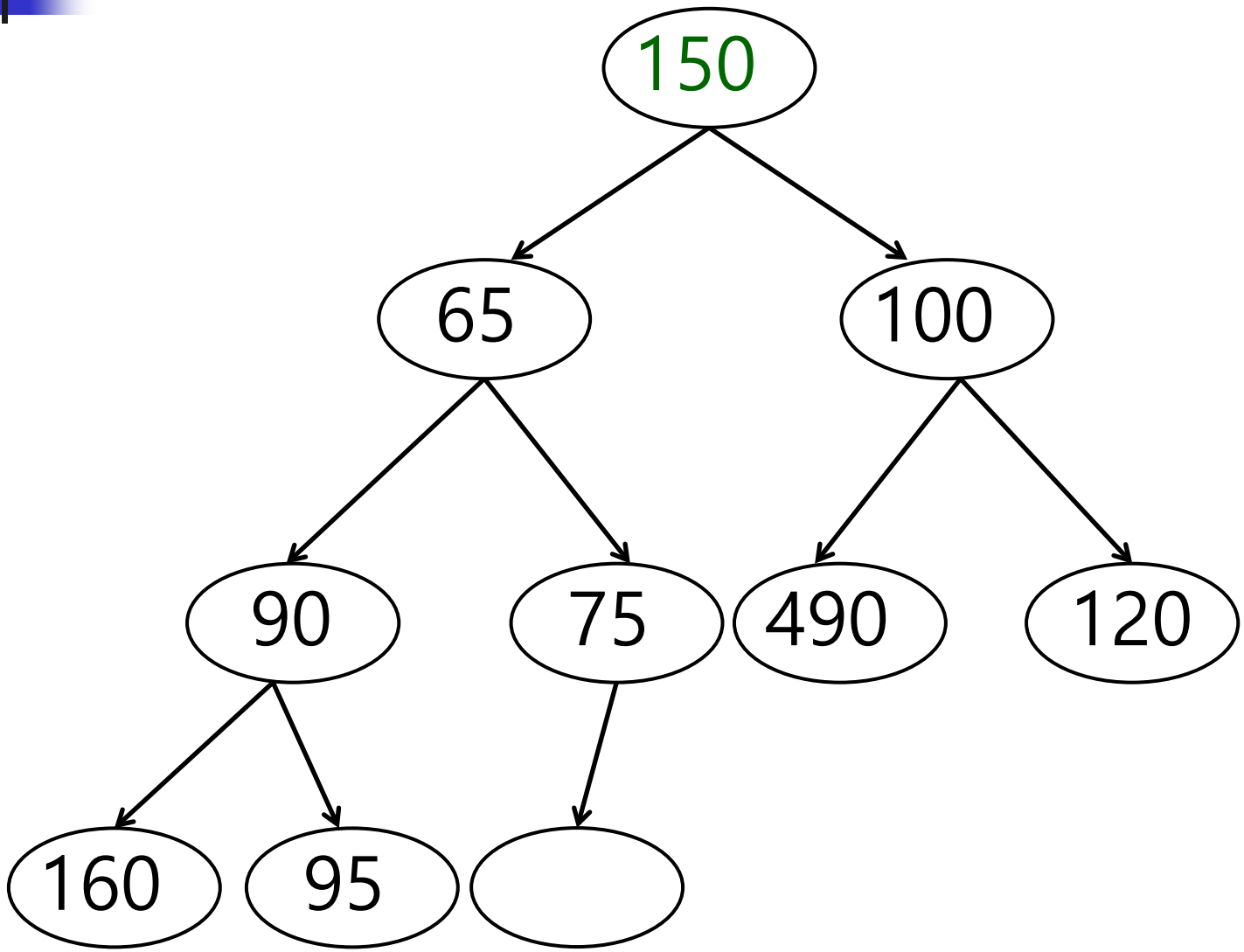
Deleting from a Min Heap: Example



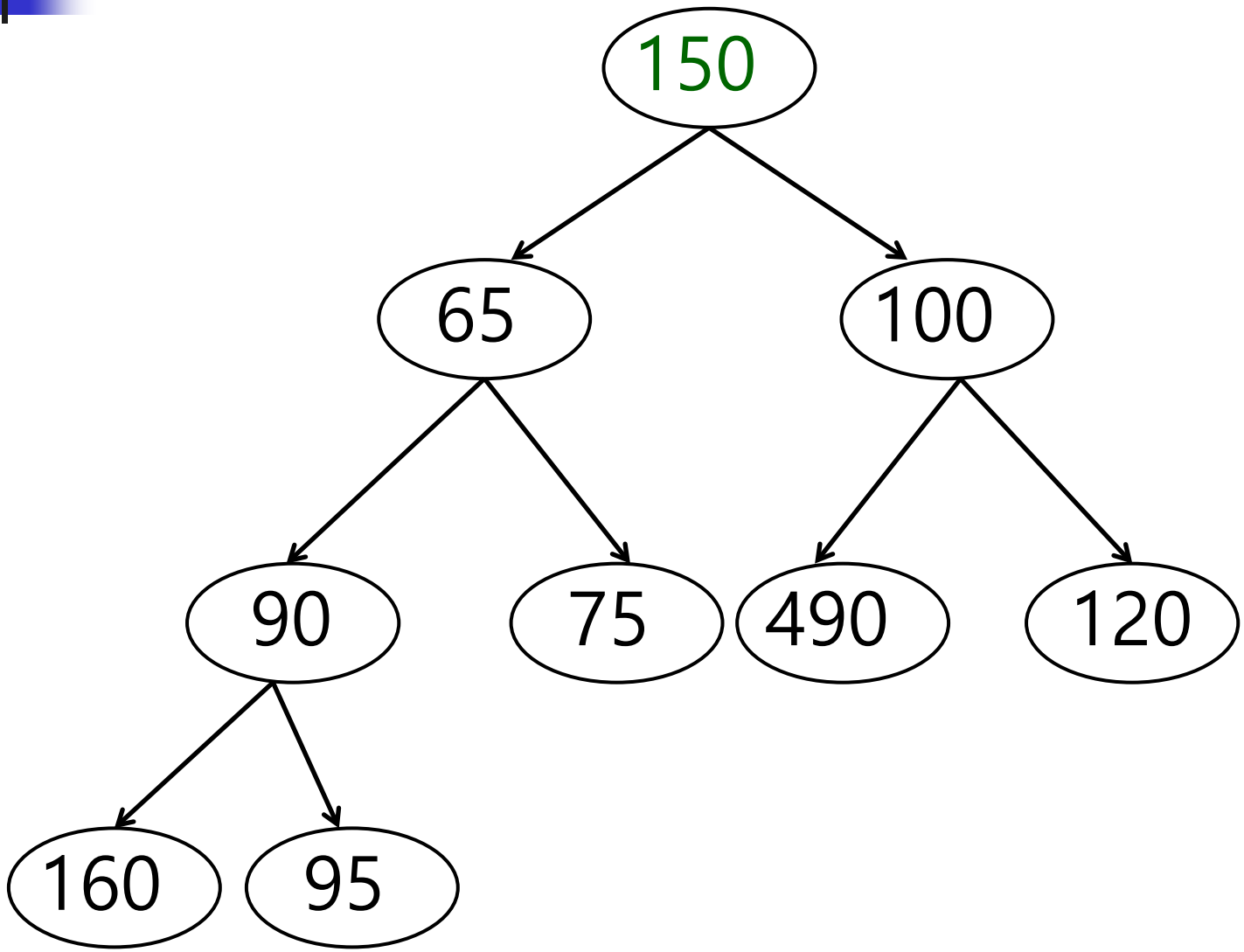
Deleting from a Min Heap: Example (cont'd)



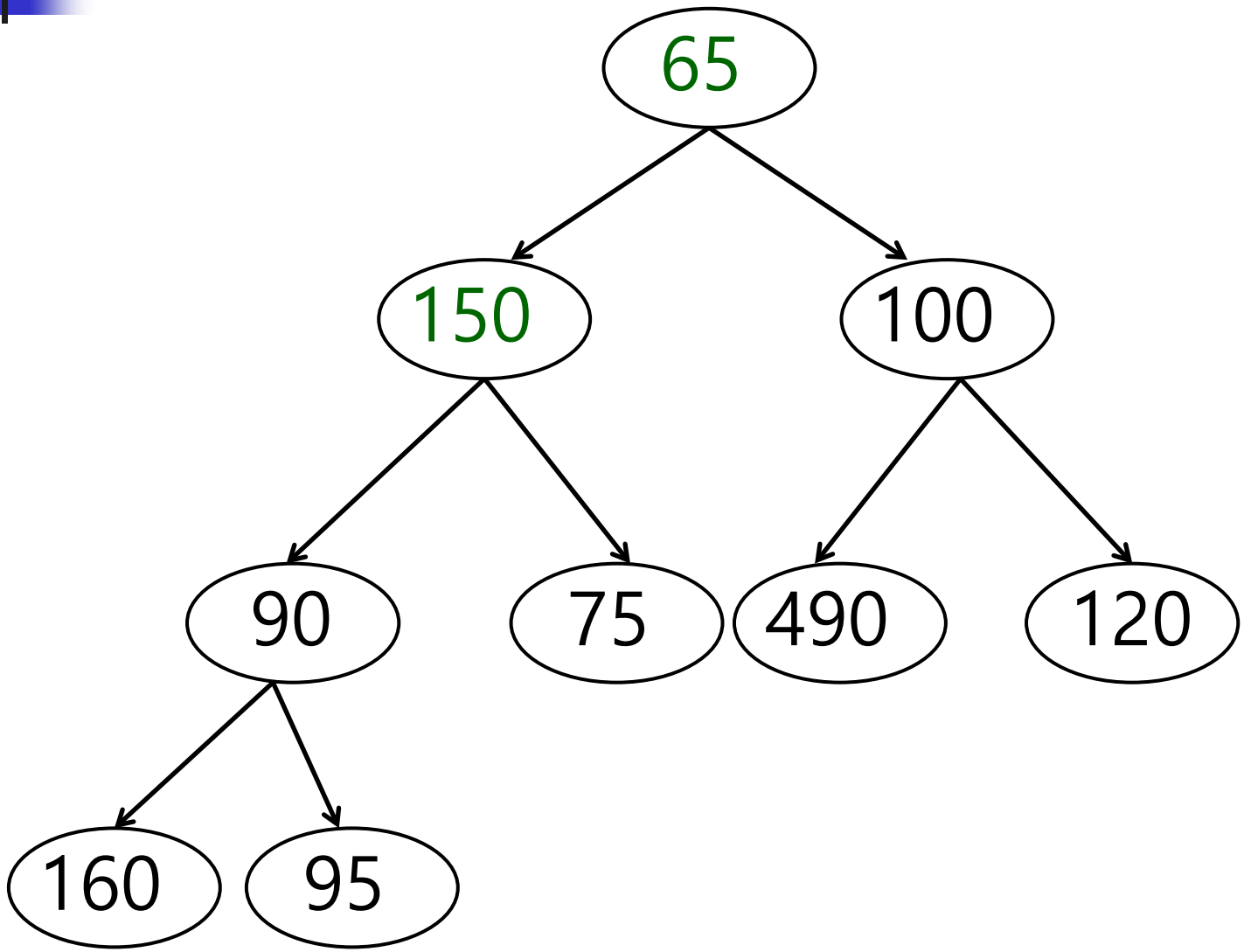
Deleting from a Min Heap: Example (cont'd)



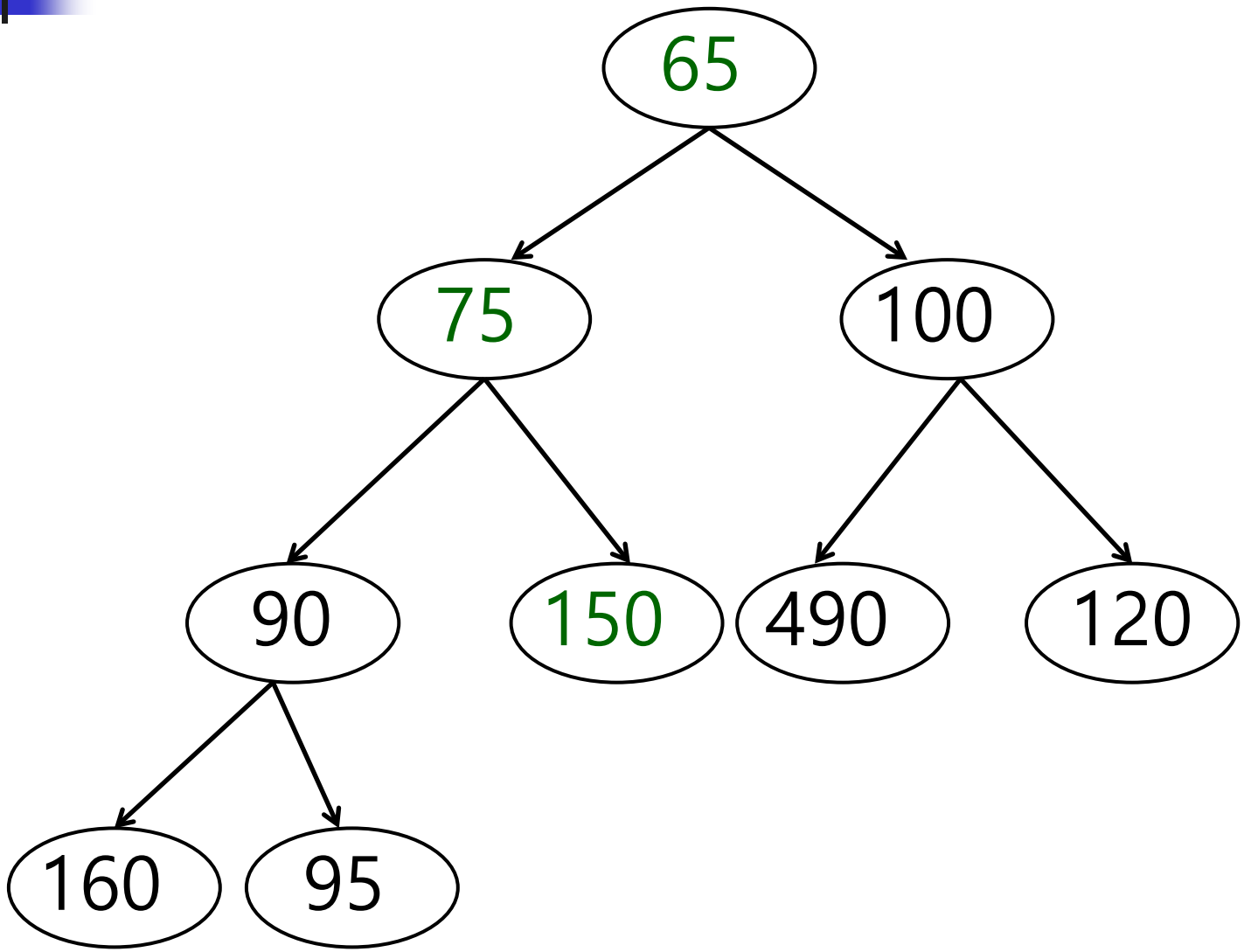
Deleting from a Min Heap: Example (cont'd)



Deleting from a Min Heap: Example (cont'd)

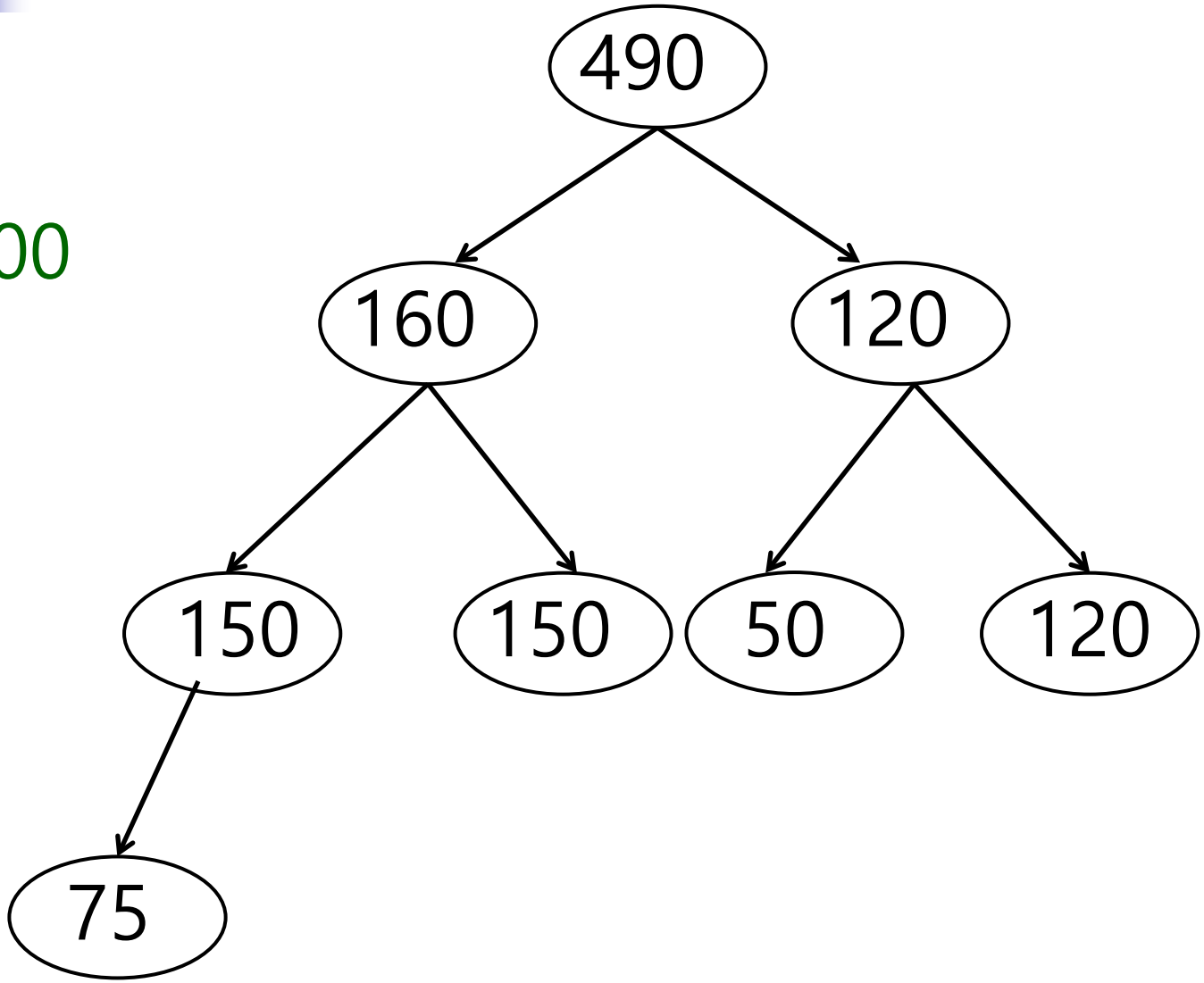


Deleting from a Min Heap: Example (cont'd)

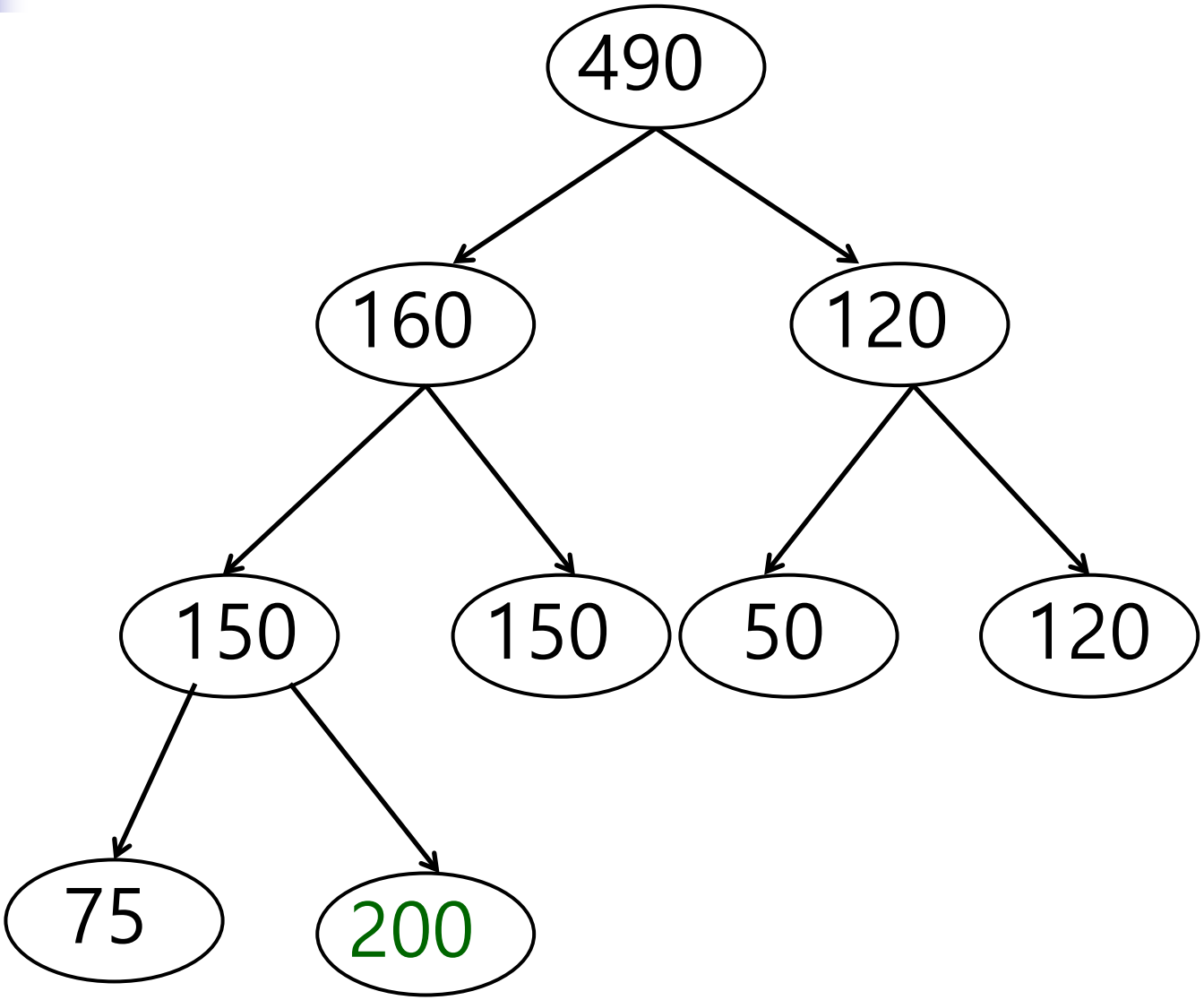


Inserting into a Max Heap: Example

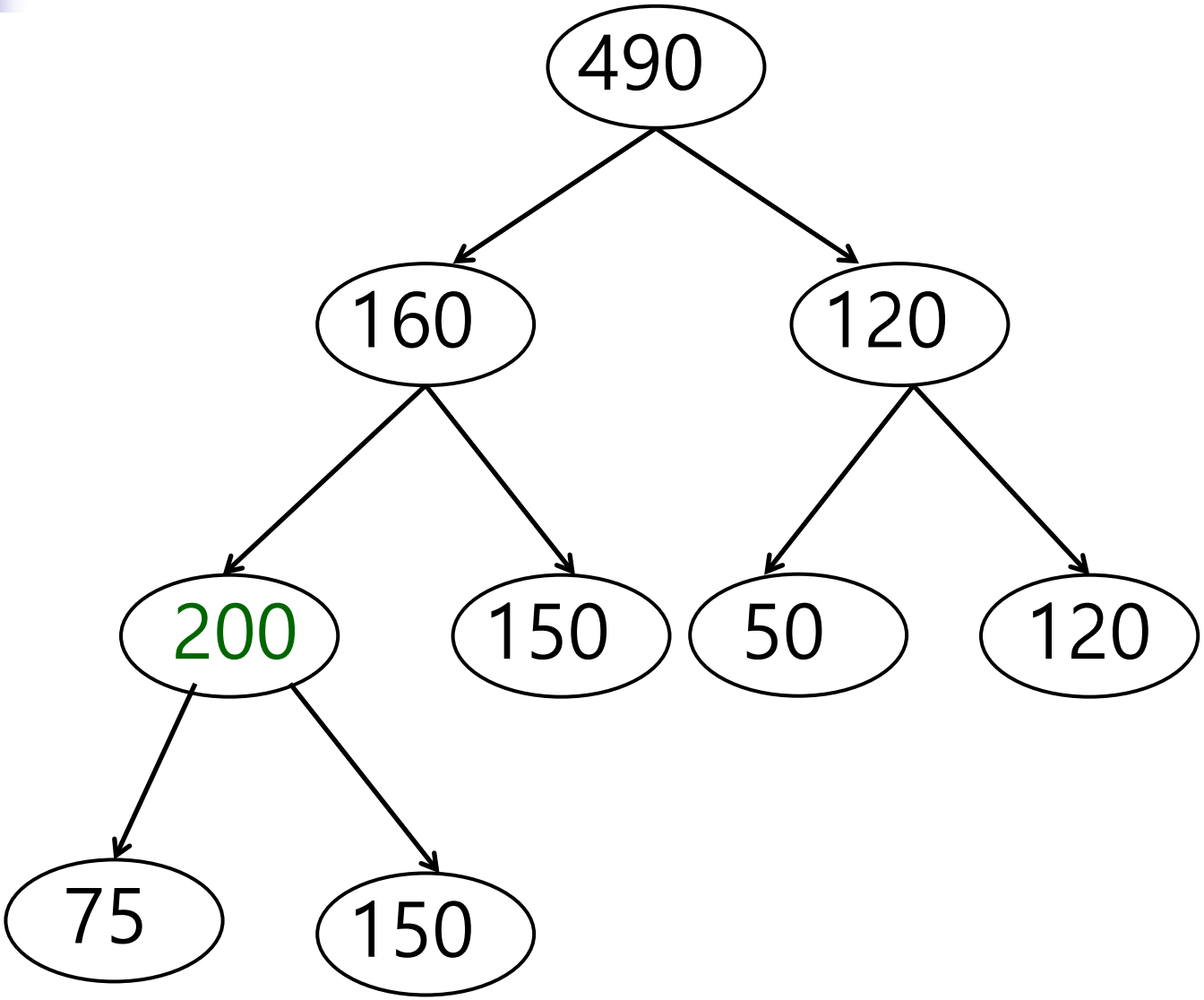
200



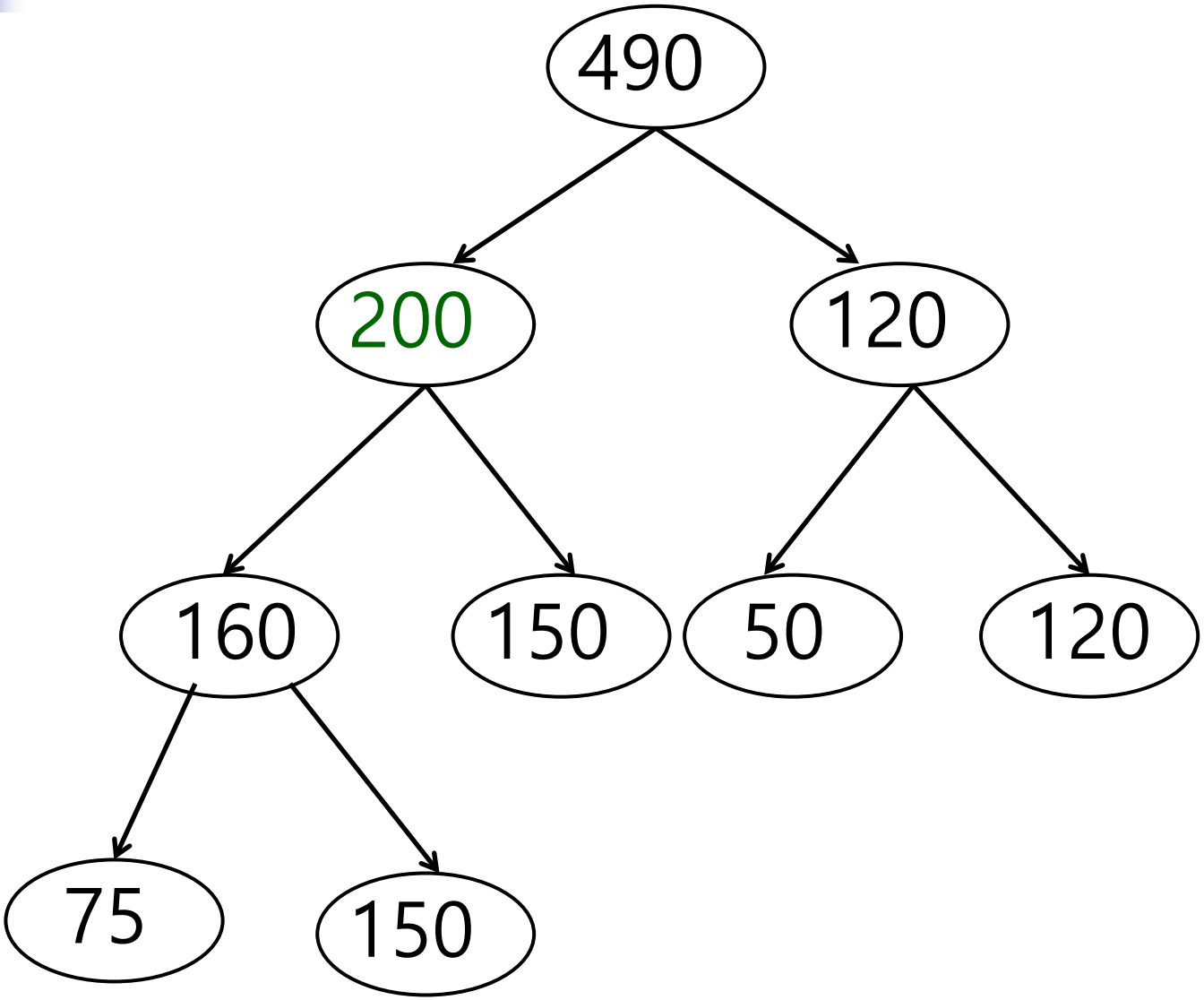
Inserting into a Max Heap: Example



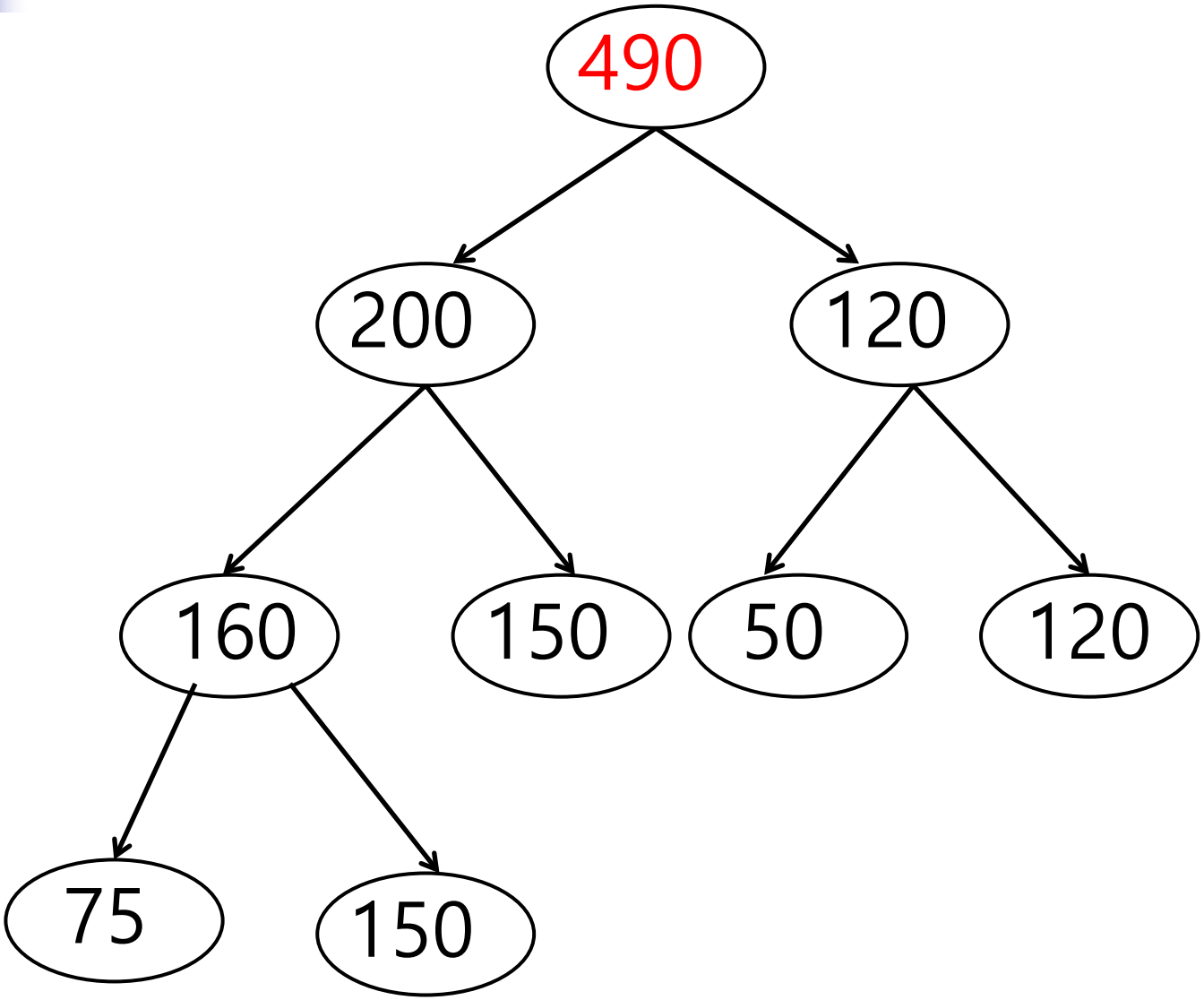
Inserting into a Max Heap: Example (contd.)



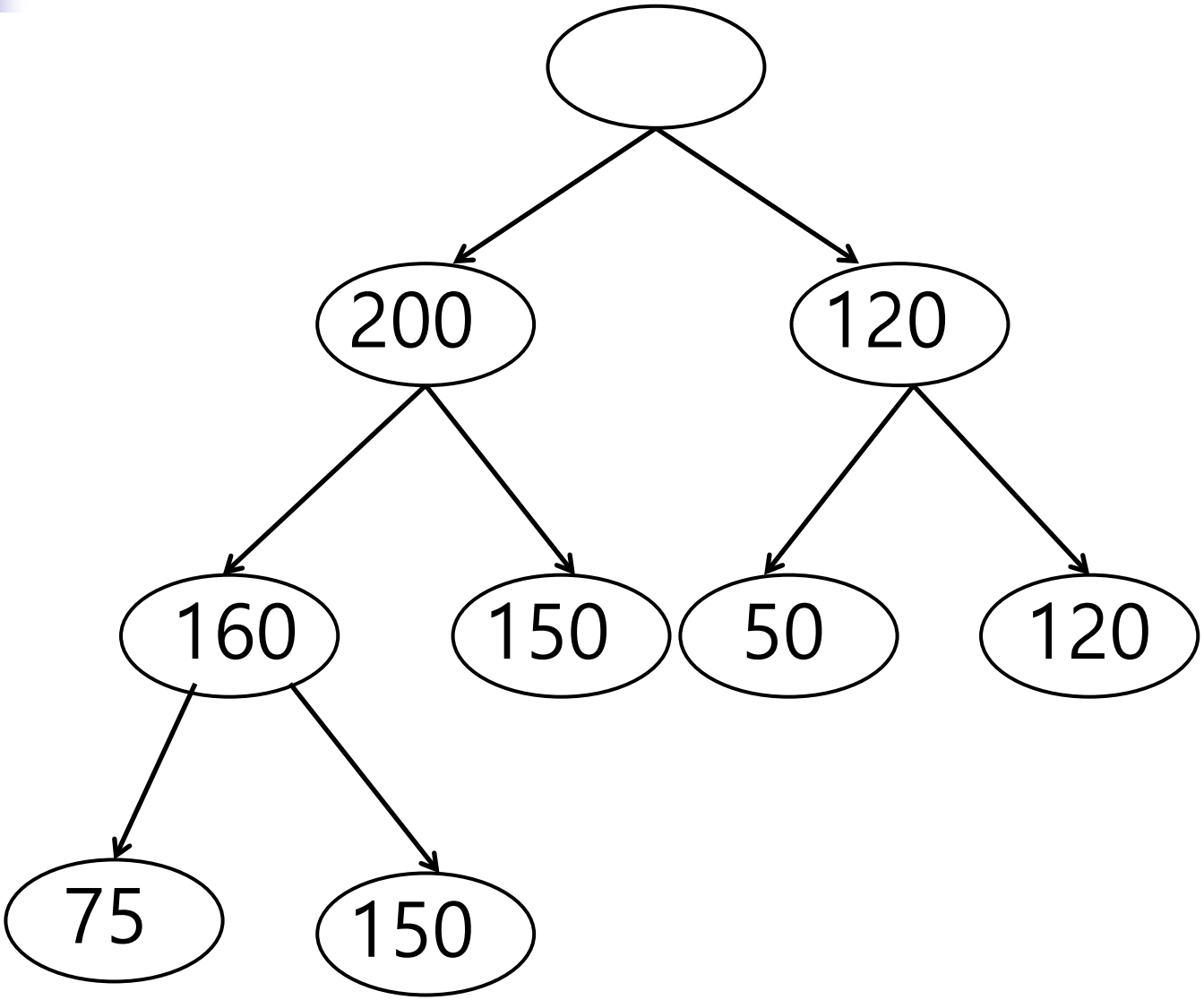
Inserting into a Max Heap: Example (contd.)



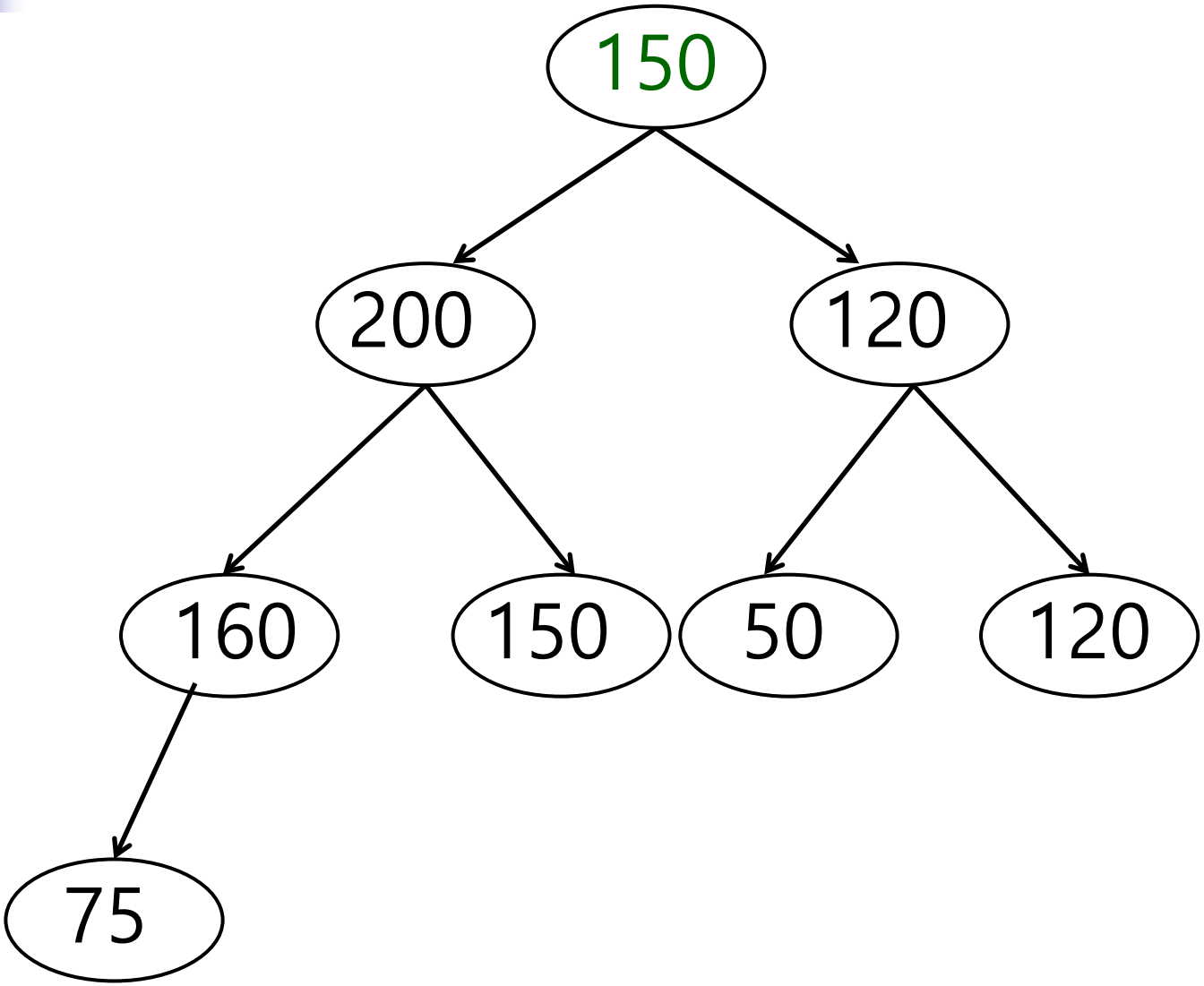
Deleting from a Max Heap: Example



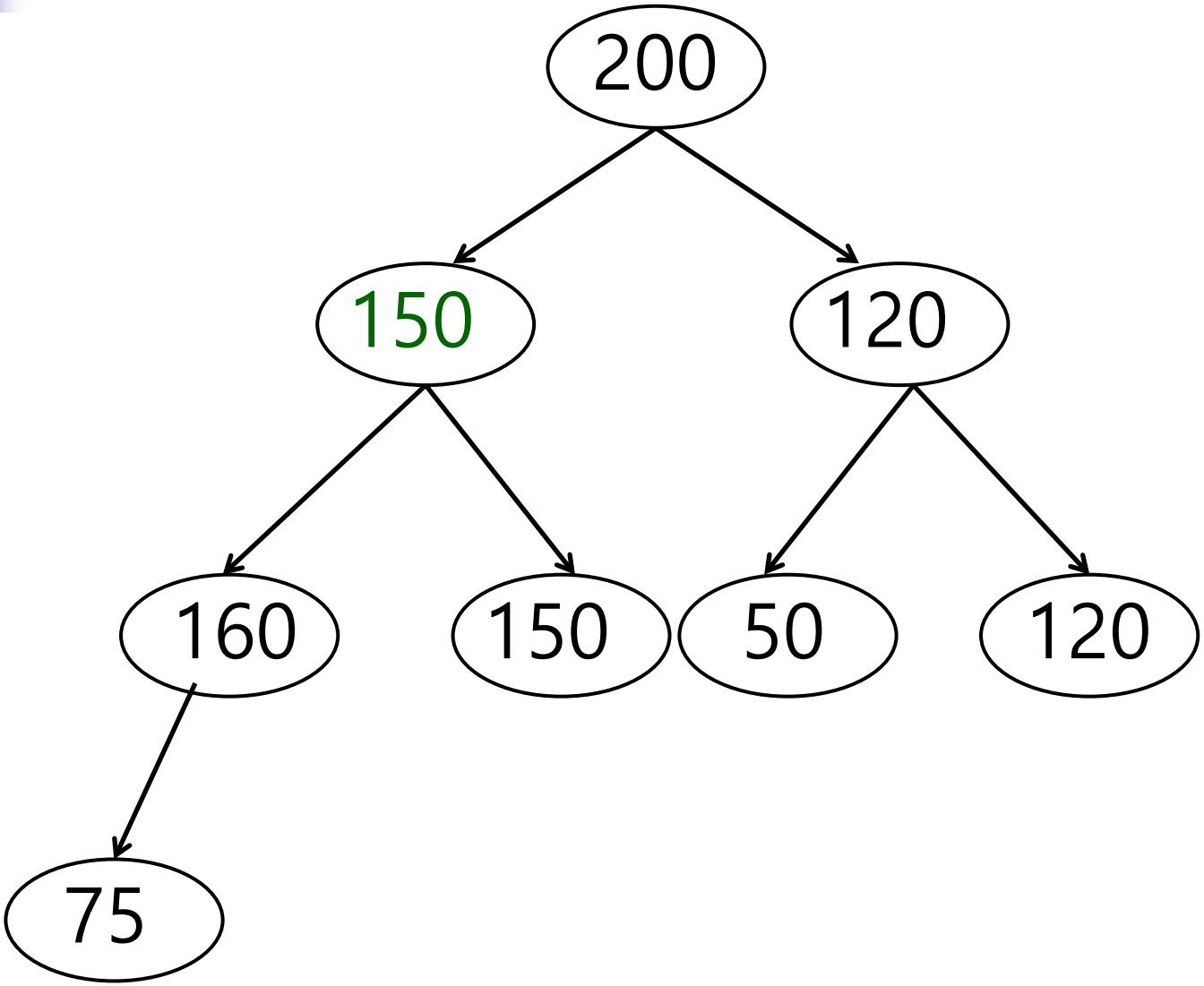
Deleting from a Max Heap: Example



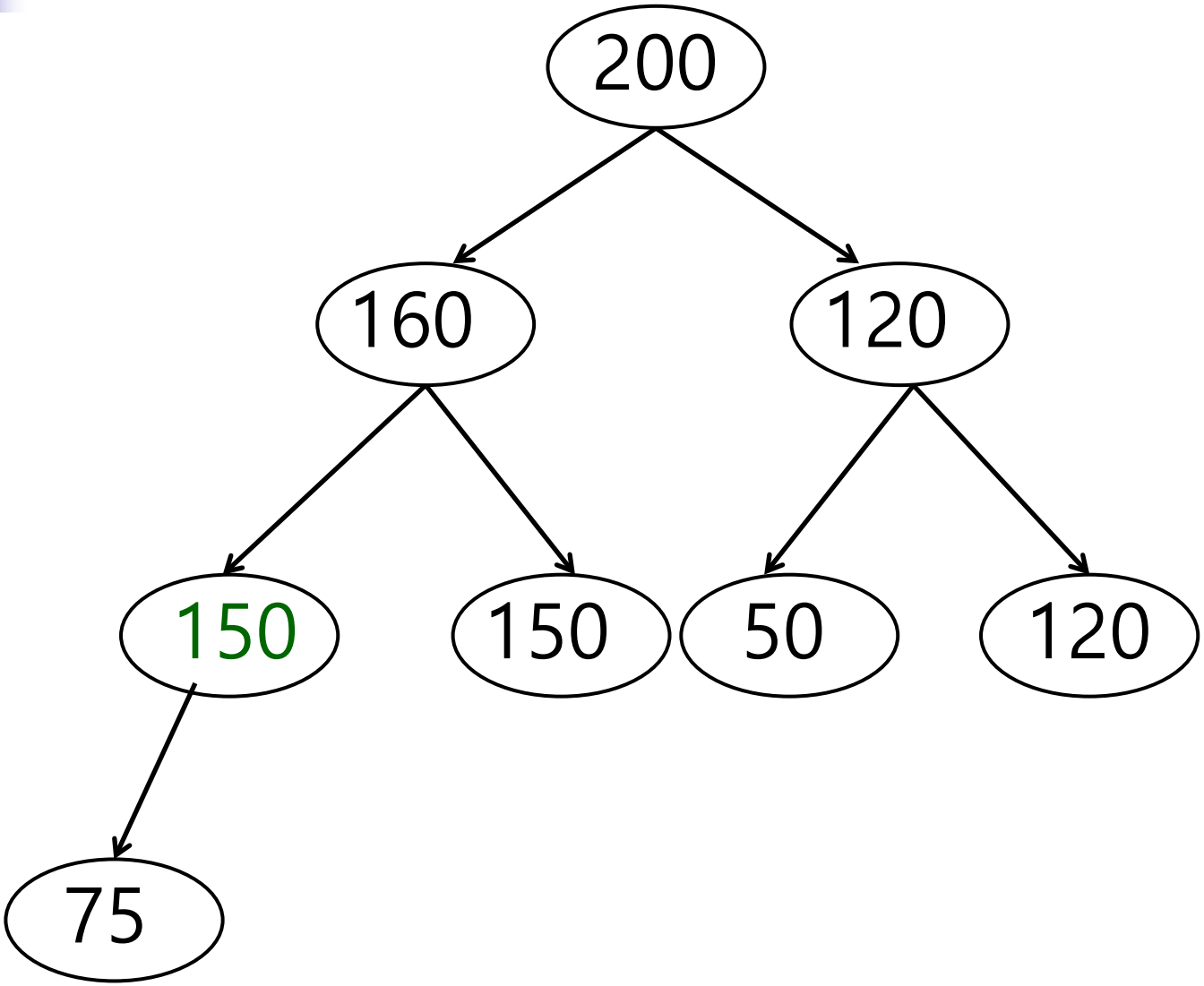
Deleting from a Max Heap: Example



Deleting from a Max Heap: Example



Deleting from a Max Heap: Example



Heap Sort (Ascending Order)

- Insert All Keys into a Min Heap.
- Keep deleting the root key and inserting the deleted key into a separate list.
  - Each deletion triggers the min heap to be restructured to maintain the min heap properties.

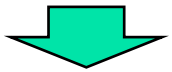
Heap Sort (Descending Order)

- Insert All Keys into a Max Heap.
- Keep deleting the root key and inserting the deleted key into a separate list.
  - Each deletion triggers the max heap to be restructured to maintain the max heap properties.

Heap Sort: Performance and Properties

- O(n $\log_2$ n) (Best, Worst, Average)
  - n: number of keys in the list
  - The height of a complete binary tree is always log n
  - $\log_2$ n comparisons for each restructuring
  - Build Heap (O(n $\log_2$ n)) + Delete All keys (O(n $\log_2$ n))
  - Best: O(n): if all the keys are the same.
- Space Complexity: O(1) (in-place sort)
  - Array-based implementation
- Unstable sorting

21, 20a, 20b, 12, 11, 8, 7



21, 20b, 20a, 12, 11, 8, 7

Heap Sort: Example code

```c
// Heap Sort in C

#include <stdio.h>

// Function to swap the the position of two elements
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

void heapify(int arr[], int n, int i) {
  // Find largest among root, left child and right child
  int largest = i;
  int left = 2 * i + 1;
  int right = 2 * i + 2;

  if (left < n && arr[left] > arr[largest])
    largest = left;

  if (right < n && arr[right] > arr[largest])
    largest = right;

  // Swap and continue heapifying if root is not largest
  if (largest != i) {
    swap(&arr[i], &arr[largest]);
    heapify(arr, n, largest);
  }
}
```

```c
// Main function to do heap sort
void heapSort(int arr[], int n) {
  // Build max heap
  for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);

  // Heap sort
  for (int i = n - 1; i >= 0; i--) {
    swap(&arr[0], &arr[i]);

    // Heapify root element to get highest element at root again
    heapify(arr, i, 0);
  }
}

// Print an array
void printArray(int arr[], int n) {
  for (int i = 0; i < n; ++i)
    printf("%d ", arr[i]);
  printf("\n");
}

// Driver code
int main() {
  int arr[] = {1, 12, 9, 5, 6, 10};
  int n = sizeof(arr) / sizeof(arr[0]);

  heapSort(arr, n);

  printf("Sorted array is \n");
  printArray(arr, n);
}
```

https://www.programiz.com/dsa/heap-sort
https://www.geeksforgeeks.org/heap-sort/

Exercise 1

- (1) Create a min heap with the following keys
  - 15 86 32 55 27 32 73 15 94 44
- (2) Do a heap sort (in ascending order)

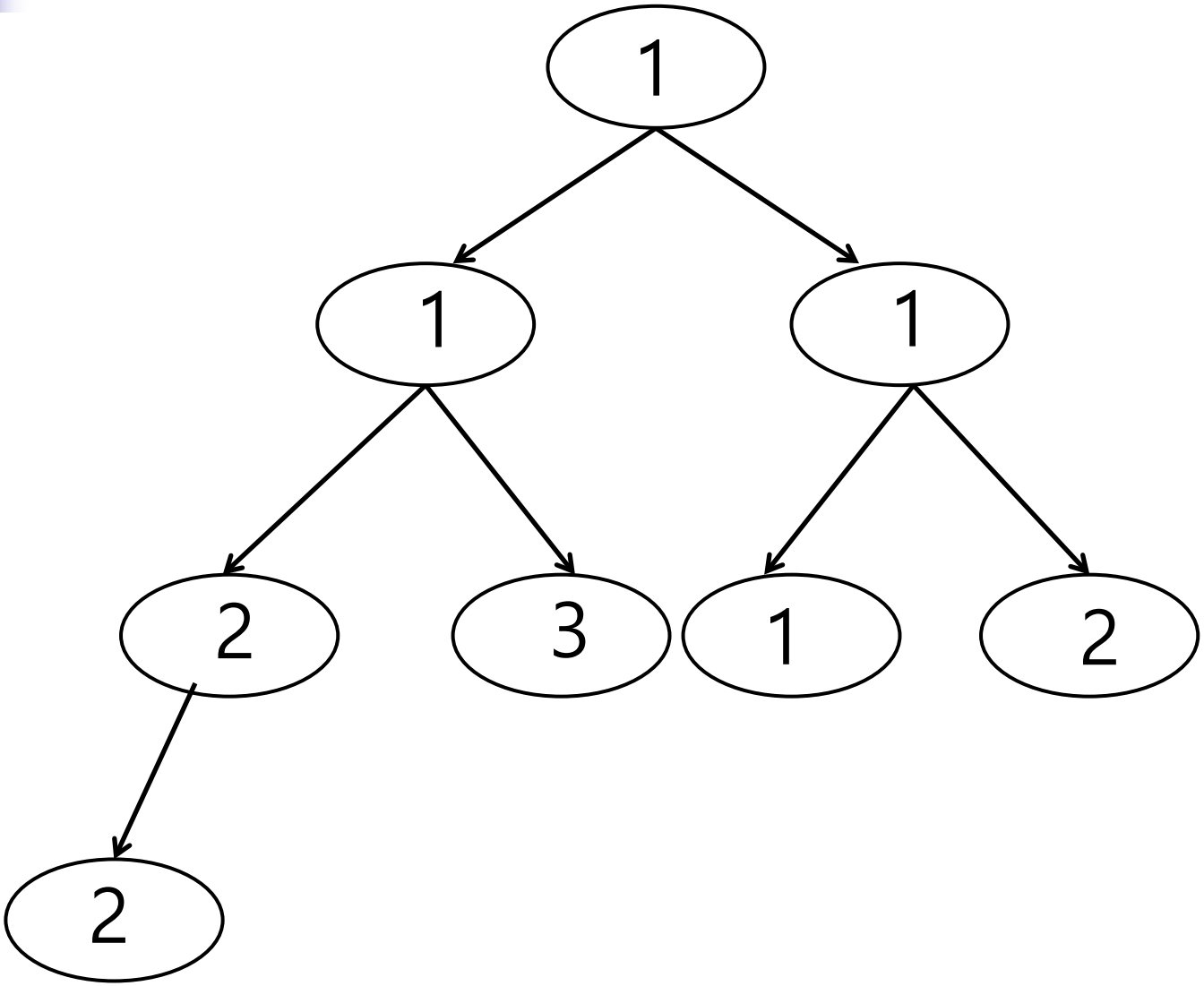
Exercise 2

- (1) Create a max heap with the following keys
  - 15 86 32 55 27 32 73 15 94 44
- (2) Do a heap sort (in descending order)

Summary Definition of a Heap

- A heap is a complete binary tree where data are automatically maintained in a priority order.

Min Heap as a Priority Queue
(Each key is one of the permissible priorities)



Counting Sort

Counting Sort

- Non-comparation based sorting
  - Sorts the elements by counting the number of occurrences of each unique element in the array.
  - The count is stored in an auxiliary array.
  - The sorting is done by mapping the count as an index of the auxiliary array.
- Example:
  - https://youtu.be/7zuGmKfUt7s
  - Note that there is NO Comparison between elements ! (hence, no swap)
- Sample code:
  - https://www.geeksforgeeks.org/counting-sort/

Counting Sort: Analysis

- O(n+k): Linear time algorithm! (Best, Worst, Average)
  - n the number of elements in the input array.
  - k is the range of input.
- Space Complexity: O(n+k) (Not in-place sort)
- Stable Sort

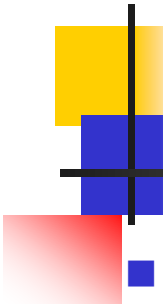
- Counting sort O(n+k) is better than Merge sort O(n log n)?

Counting Sort: Analysis

- O(n+k): Linear time algorithm! (Best, Worst, Average)
  - n: number of keys to sort
  - k is the range of input.
- Space Complexity: O(n+k) (Not in-place sort)
- Stable Sort

- Some drawbacks.. (The space complexity is the problem!)
  - For negative numbers?
    - No array indices for negative numbers. (X work)
  - For floating point numbers?
    - Requires too much memory space. (k is too big)
  - e.g. [1, 2, 3, 10^4] (n=4)
    - O(n log n) = 4*2 = 8
    - O(n+k) = 4 + $10^4 \approx 10^4 \approx n^5$
- It must be used only when k < log(n).

Radix Sort



Radix Sort

- “Radix” = “Character”
- Sort by numeric or alphabetic symbols in lexicographical order
- Least Significant Digit (LSD) Radix Sort
- Most Significant Digit (MSD) Radix Sort
- Supplementary Reading
  - http://en.wikipedia.org/wiki/Radix_sort

LSD Radix Sort

- k Passes over n Keys
- 1st Pass
  - Create a list for each distinct LSD digit in lexicographic order, and throw the keys of the same LSD digit into the list in their original order.
  - Make a sequence of the lists created above.
- ith Pass (i = 2 through k)
  - Repeat the above for the ith LSD digit
- Examples:
  - https://www.youtube.com/watch?v=nu4gDuFabIM
  - https://www.youtube.com/watch?v=xuU-DS_5Z4g

LSD Radix Sort: Example

sort the list:
170, 90, 2, 802, 24, 45, 75, 66

1st pass:
(170, 90), (2, 802), (24), (45, 75), (66)

2nd pass:
(2), (802), (24), (45), (66), (170, 75), (90)

3rd pass:
(2), (24), (45), (66), (75), (90), (170), (802)

LSD Radix Sort: Example

sort the list:

170, 90, 2, 802, 24, 45, 75, 66

(170, 090, 002, 802, 024, 045, 075, 066)

1st pass:

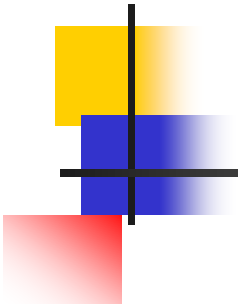
(170, 090), (002, 802), (024), (045, 075), (066)

2nd pass:

(002), (802), (024), (045), (066), (170, 075), (090)

3rd pass:

(002), (024), (045), (066), (075), (090), (170), (802)



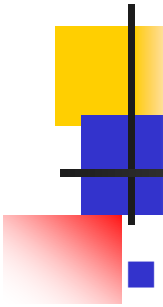
Exercise

sort the list:
170000, 10

Exercise

sort the list:

17000, 190, 2, 3802, 24, 45, 75, 66



MSD Radix Sort

- Recursively sort for each digit, starting from the most significant digit.
- Usually MSD is used to sort strings of variable length, unlike LSD.

MSD Radix Sort: Example

sort the list:

170, 90, 2, 802, 24, 25, 75, 66

(170, 090, 002, 802, 024, 025, 075, 066)

1st pass:

(090, 002, 024, 025, 075, 066), (170), (802)

2nd pass:

(002), (024,025), (066), (075), (090), (170), (802)

3rd pass:

(002), (024), (025), (066), (075), (090), (170), (802)

MSD Radix Sort: Exercise

sort the list:

170, 4, 160, 90, 4, 45, 24, 890, 892, 802, 2, 45

Radix Sort: Example code

```cpp
// C++ implementation of Radix Sort
#include <iostream>
using namespace std;

// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[10] = { 0 };

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains actual
    //  position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

```cpp
// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

// A utility function to print an array
void print(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver Code
int main()
{
    int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };
    int n = sizeof(arr) / sizeof(arr[0]);

      // Function Call
      radixsort(arr, n);
    print(arr, n);
    return 0;
}
```

https://www.geeksforgeeks.org/radix-sort/
https://www.programiz.com/dsa/radix-sort#:~:text=Radix%20sort%20is%20a%20sorting,value%20of%20the%20unit%20place

Radix Sort: Performance and Qualities

- O(d(n+k)), O(d n)
  - n: number of keys to sort
  - k is the range of input.
  - d: number of digits per key

  - Non-comparative sort.
  - Linear time sorting algorithm.
  - Radix sort uses counting sort as a subroutine.
- Space Complexity: O(n+k), O(n+d) (Not in-place sort)
  - Performance depends on the size of the keys and choice of the radix (digit, alphabet,…)
- Stable sort

Radix Sort: Performance and Qualities

- Problems
  - Same problem with counting sort.
  - Takes much more space for the radix-sorted lists
    - e.g. [1, 2, 3, 10^{10}]
    - It must be used only when k < log(n).

- Hence, Radix sort is less flexible than other comparison-based sort algorithms.

Summery

Summary Observations

- Every sorting algorithm has its uses and problems.
  - There is no such thing as “the best sorting algorithm for every situation”.
- For a large internal list, a combination of a few algorithms may be best.
  - (e.g.) insertion sort, quick sort, merge sort
    - Quick sort can use insertion sort for small sublists.
    - Merge sort can use quick sort for medium-size sublists.

Sorting in Widely Used Systems

- Efficient quicksort implementations generally outperform merge sort for sorting RAM-based arrays.
- Merge sort is a stable sort and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list.
- The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) impossible.
- The Linux kernel uses merge sort for linked lists. Python uses Timsort, a tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7 on Android platform and in GNU Octave (a free alternative to Matlab).
- As of Perl 5.8, merge sort is its default sorting algorithm (it was quicksort in previous versions of Perl).
- In Java, the Arrays.sort() methods use merge sort or a tuned quicksort depending on the data types; switch to insertion sort when fewer than seven array elements are being sorted.

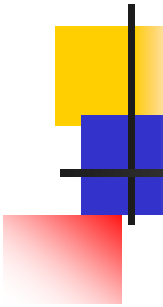
Assignment 9

HW-9(P1): Descending Order Heap Sort (Max Heap)

31, 11, 3, 20, 19, 24, 17, 3, 31

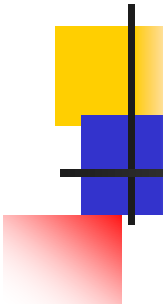
HW-9(P2): Ascending Order Heap Sort (Min Heap)

31, 11, 3, 20, 19, 24, 17, 3, 31



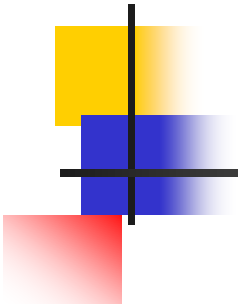
HW-9(P3): LSD Radix Sort

310, 11, 3, 20, 11119, 24, 17, 3, 31



HW-9(P4): MSD Radix Sort

310, 11, 3, 20, 11119, 24, 17, 3, 31



End of Lecture