CS 2124: DATA STRUCTURES
Spring 2024

8th Lecture

Topics: **Heaps**

Default Canvas Grading Scheme

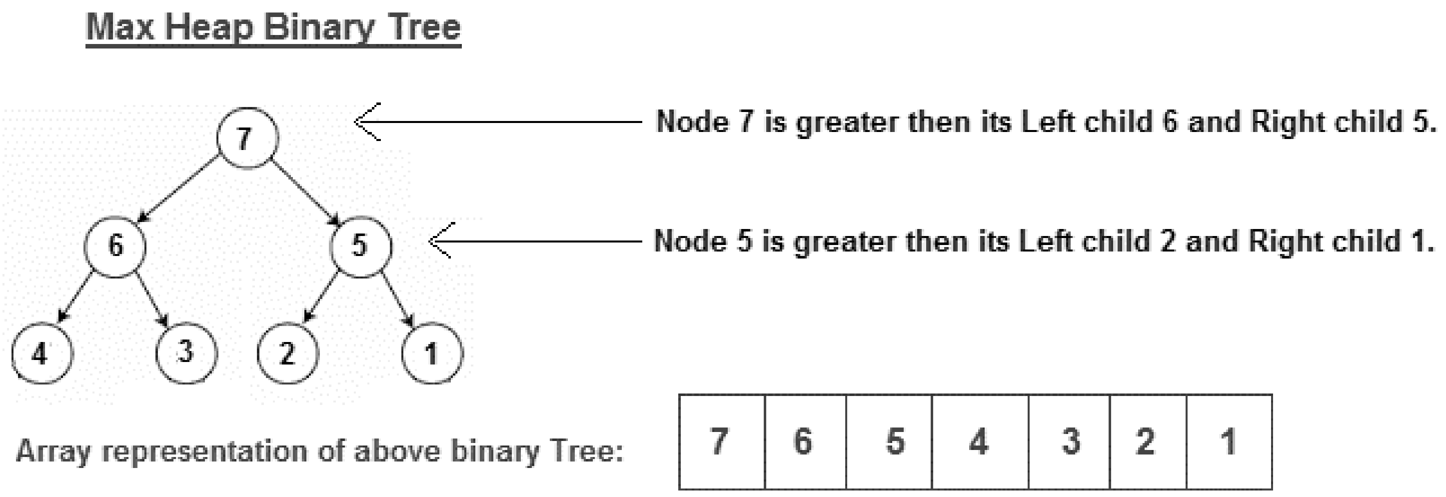
Letter Grade	Range
A	100% to 94%
A-	< 94% to 90%
B+	< 90% to 87%
B	< 87% to 84%
B-	< 84% to 80%
C+	< 80% to 77%
C	< 77% to 74%
C-	< 74% to 70%
D+	< 70% to 67%
D	< 67% to 64%
D-	< 64% to 61%
F	< 61% to 0%

Heap (Applications)

- **Priority Queues:** (Usually Heap Property) Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(log N) time.
- **Order statistics:** The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.
- **Sorting:**
  - Max-heap are use for heapsorting

Input 35 33 42 10 14 19 27 44 26 31

Heap (Applications - Sorting)



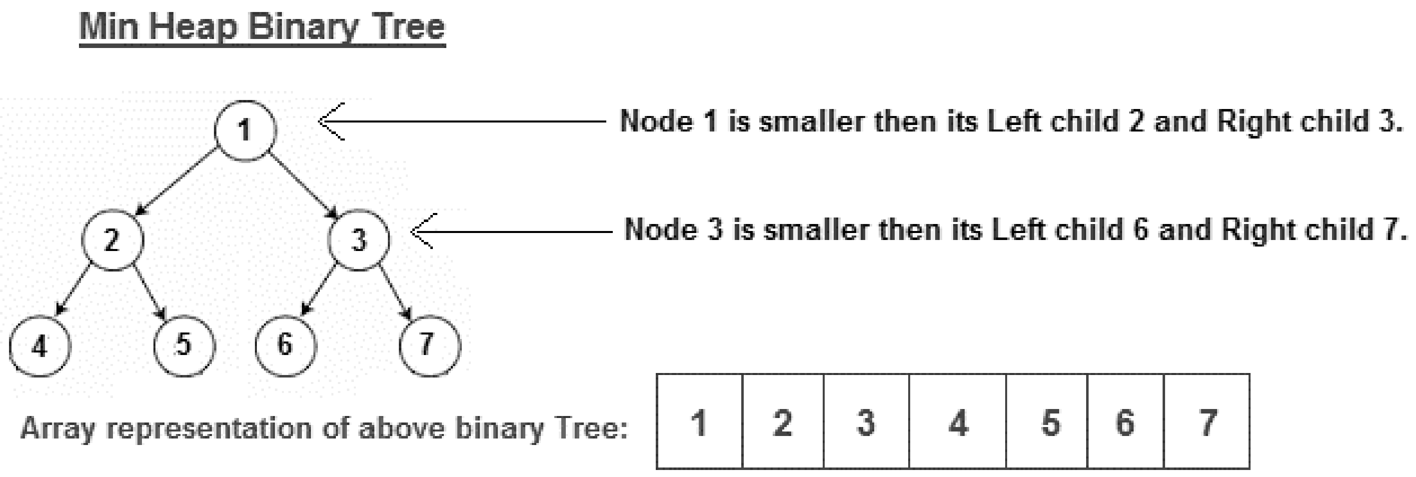
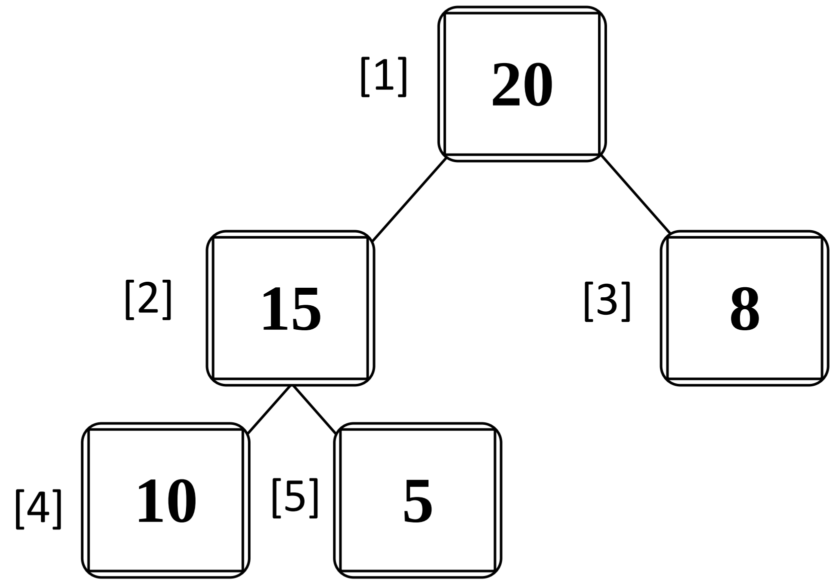


Image Source: [Link]

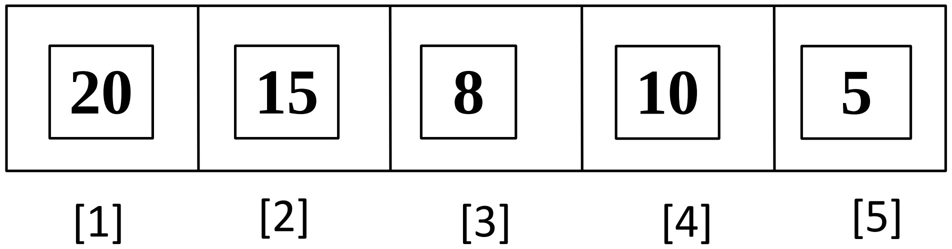
Max-Priority Queue

```c
int main() {
  int A[tree_array_size];
  insert(A, 20);
  insert(A, 15);
  insert(A, 8);
  insert(A, 10);
  insert(A, 5);

  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));

  print_heap(A);

  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
```

Is it a Max Heap Tree?



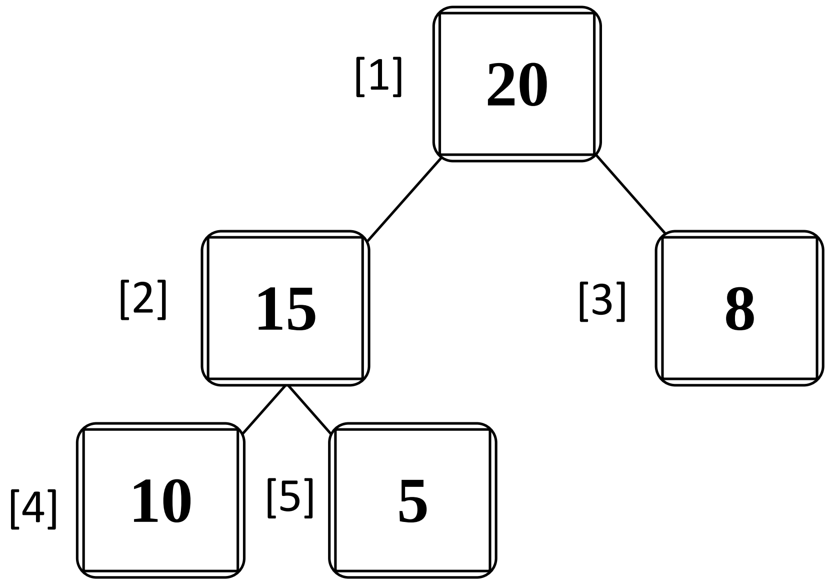

```c
void increase_key(int A[], int index, int key) {
  A[index] = key;
  while((index>1) && (A[get_parent(A, index)] < A[index])) {
    swap(&A[index], &A[get_parent(A, index)]);
    index = get_parent(A, index);
  }
}
```



Source

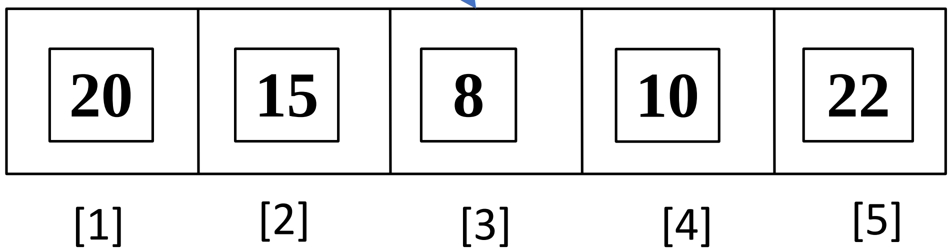
Max-Priority Queue

Is it a Max Heap Tree?




```c
int main() {
  int A[tree_array_size];
  insert(A, 20);
  insert(A, 15);
  insert(A, 8);
  insert(A, 10);
  insert(A, 5);

  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));

  print_heap(A);

  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
```

```c
void increase_key(int A[], int index, int key) {
  A[index] = key;
  while((index>1) && (A[get_parent(A, index)] < A[index])) {
    swap(&A[index], &A[get_parent(A, index)]);
    index = get_parent(A, index);
  }
}
```

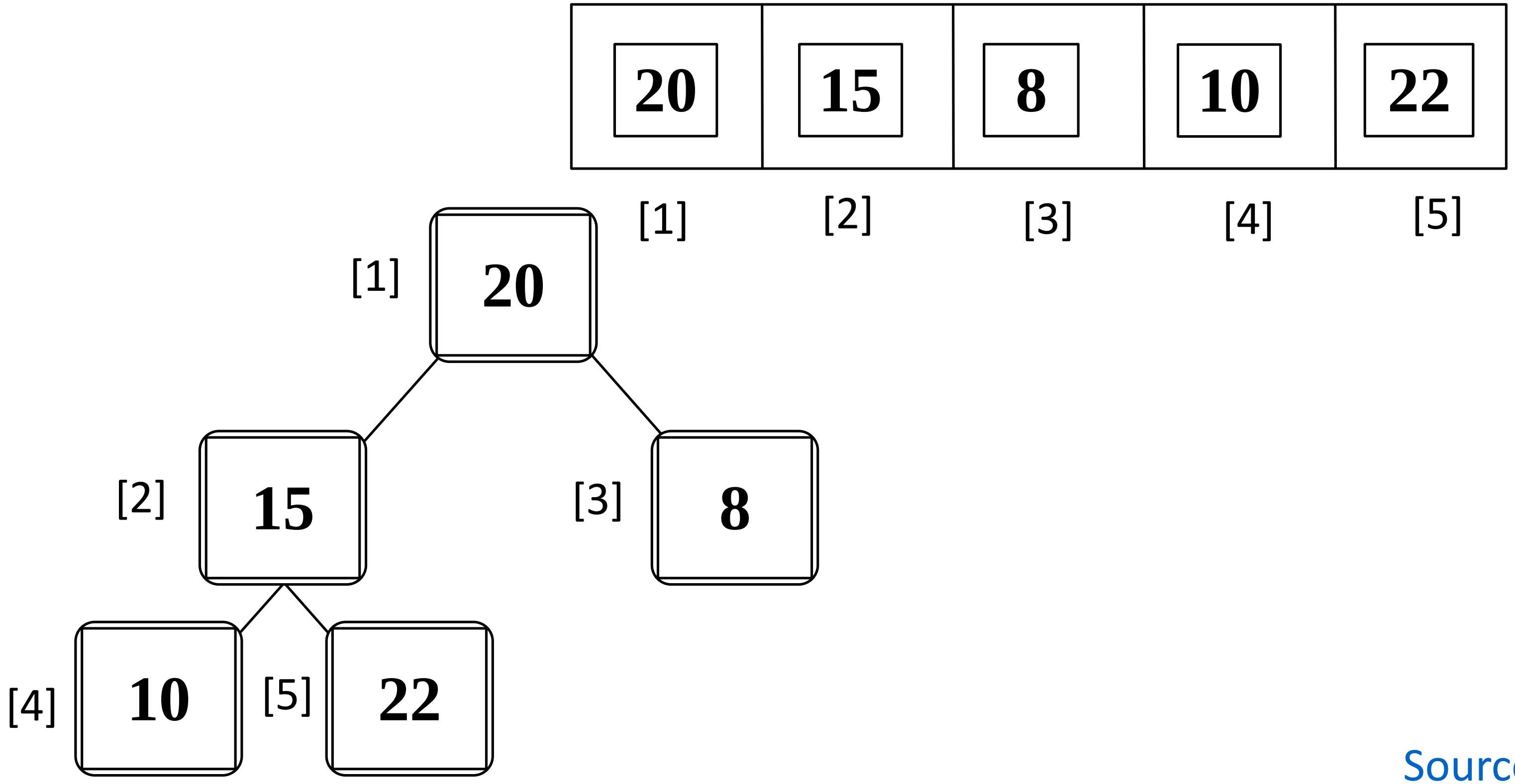


Source

Max-Priority Queue

```
int main() {
  int A[tree_array_size];
  insert(A, 20);
  insert(A, 15);
  insert(A, 8);
  insert(A, 10);
  insert(A, 5);

  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));

  print_heap(A);

  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
```

```
void increase_key(int A[], int index, int key) {
  A[index] = key;
  while((index>1) && (A[get_parent(A, index)] < A[index])) {
    swap(&A[index], &A[get_parent(A, index)]);
    index = get_parent(A, index);
  }
}
```



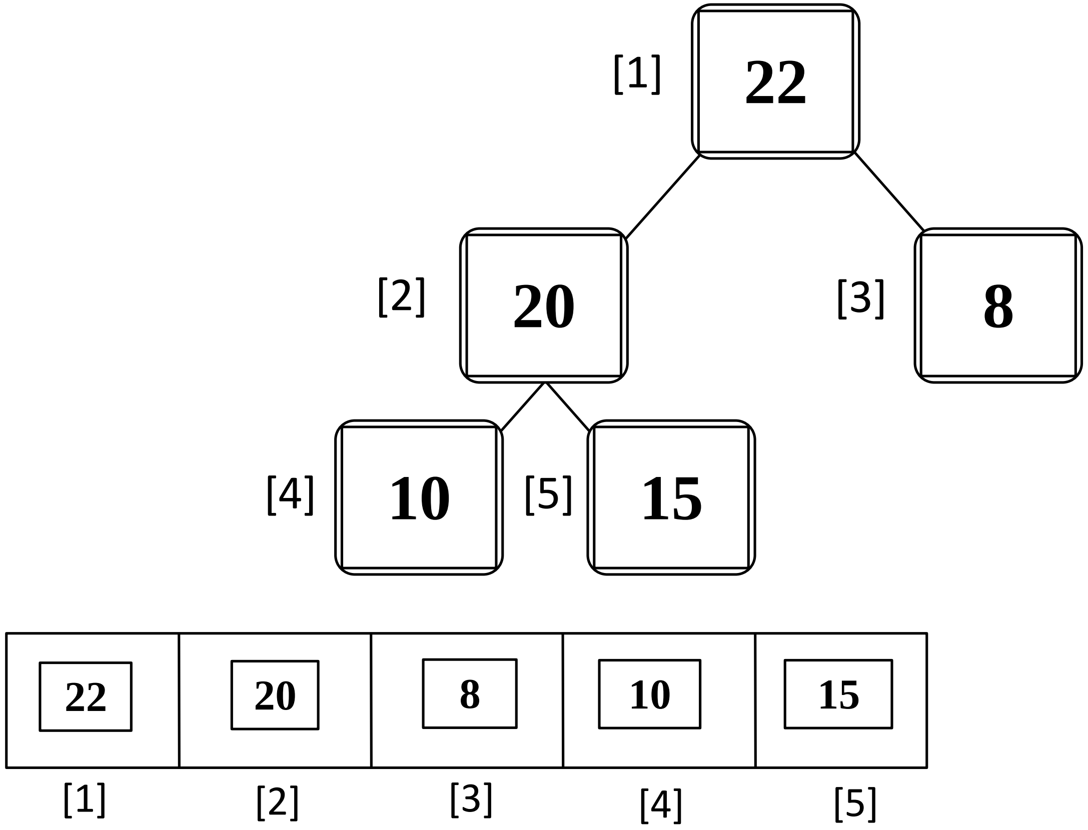
Source

Max-Priority Queue

```c
void increase_key(int A[], int index, int key) {
  A[index] = key;
  while((index>1) && (A[get_parent(A, index)] < A[index])) {
    swap(&A[index], &A[get_parent(A, index)]);
    index = get_parent(A, index);
  }
}
```

```c
int main() {
  int A[tree_array_size];
  insert(A, 20);
  insert(A, 15);
  insert(A, 8);
  insert(A, 10);
  insert(A, 5);

  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));

  print_heap(A);

  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
```

Is it a Max Heap Tree?

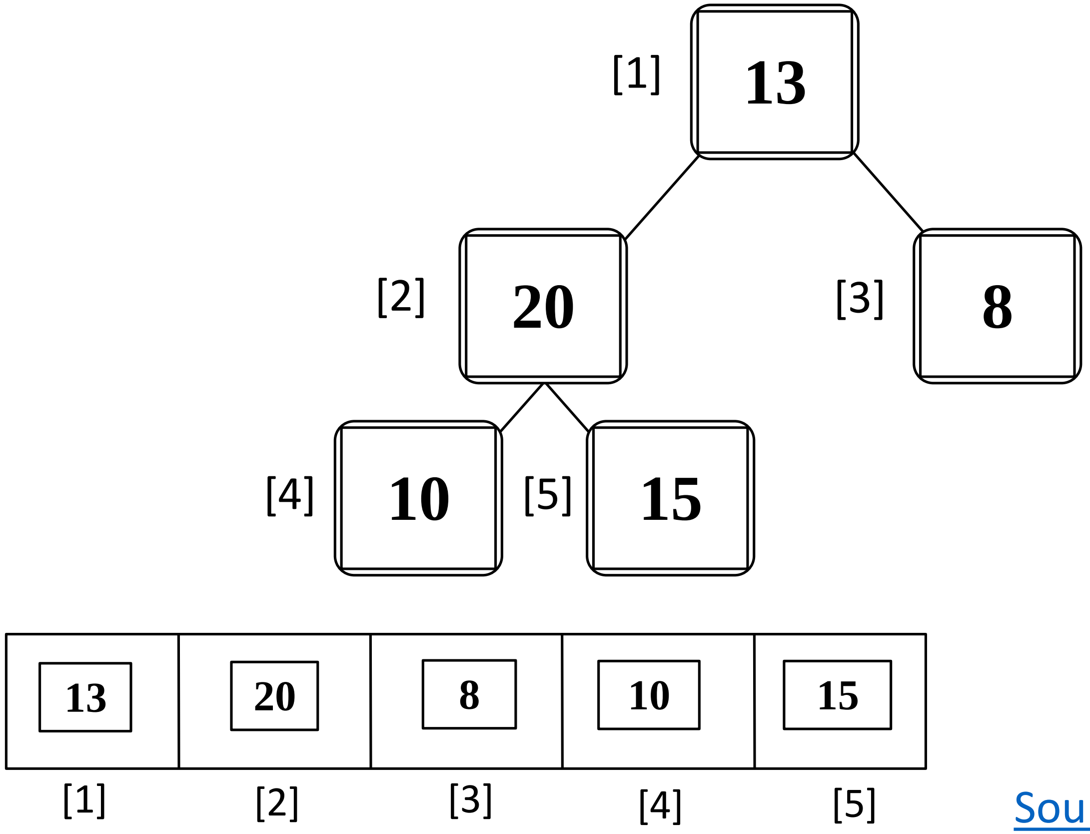


Source

Max-Priority Queue

```c
int main() {
  int A[tree_array_size];
  insert(A, 20);
  insert(A, 15);
  insert(A, 8);
  insert(A, 10);
  insert(A, 5);

  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));

  print_heap(A);

  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
```

```c
void decrease_key(int A[], int index, int key) {
  A[index] = key;
  max_heapify(A, index);
}
```



Source

Max-Priority Queue

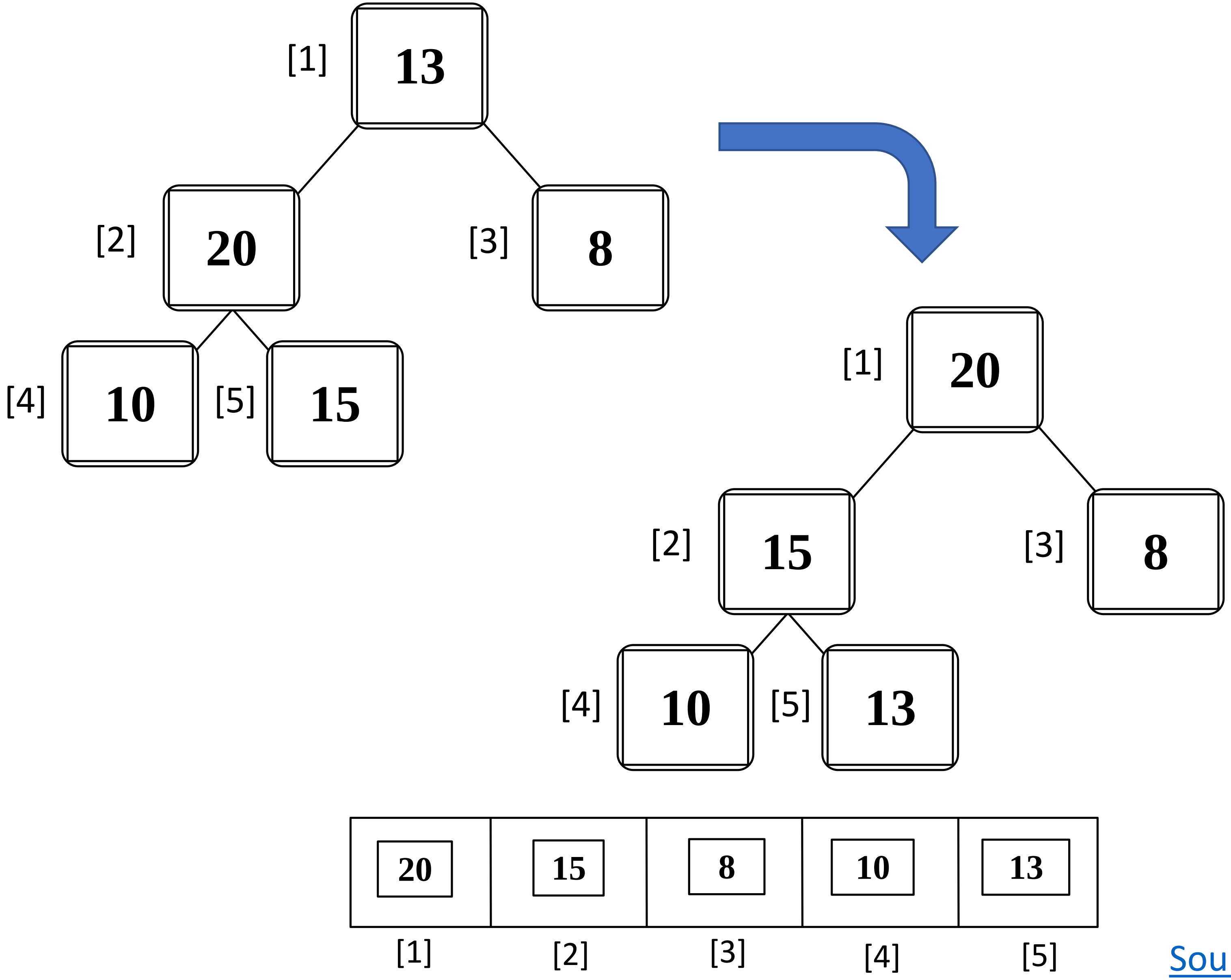
```c
int main() {
  int A[tree_array_size];
  insert(A, 20);
  insert(A, 15);
  insert(A, 8);
  insert(A, 10);
  insert(A, 5);

  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));

  print_heap(A);

  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
```

```c
void decrease_key(int A[], int index, int key) {
  A[index] = key;
  max_heapify(A, index);
}
```

```c
void max_heapify(int A[], int index) {
  int left_child_index = get_left_child(A, index);
  int right_child_index = get_right_child(A, index);
  // finding largest among index, left child and right child
  int largest = index;
  if ((left_child_index <= heap_size) && (left_child_index>0)) {
    if (A[left_child_index] > A[largest]) {
      largest = left_child_index;
    } }
  if ((right_child_index <= heap_size && (right_child_index>0))) {
    if (A[right_child_index] > A[largest]) {
      largest = right_child_index;
    } }
  // largest is not the node, node is not a heap
  if (largest != index) {
    swap(&A[index], &A[largest]);
    max_heapify(A, largest);
  } }
```

Max-Priority Queue

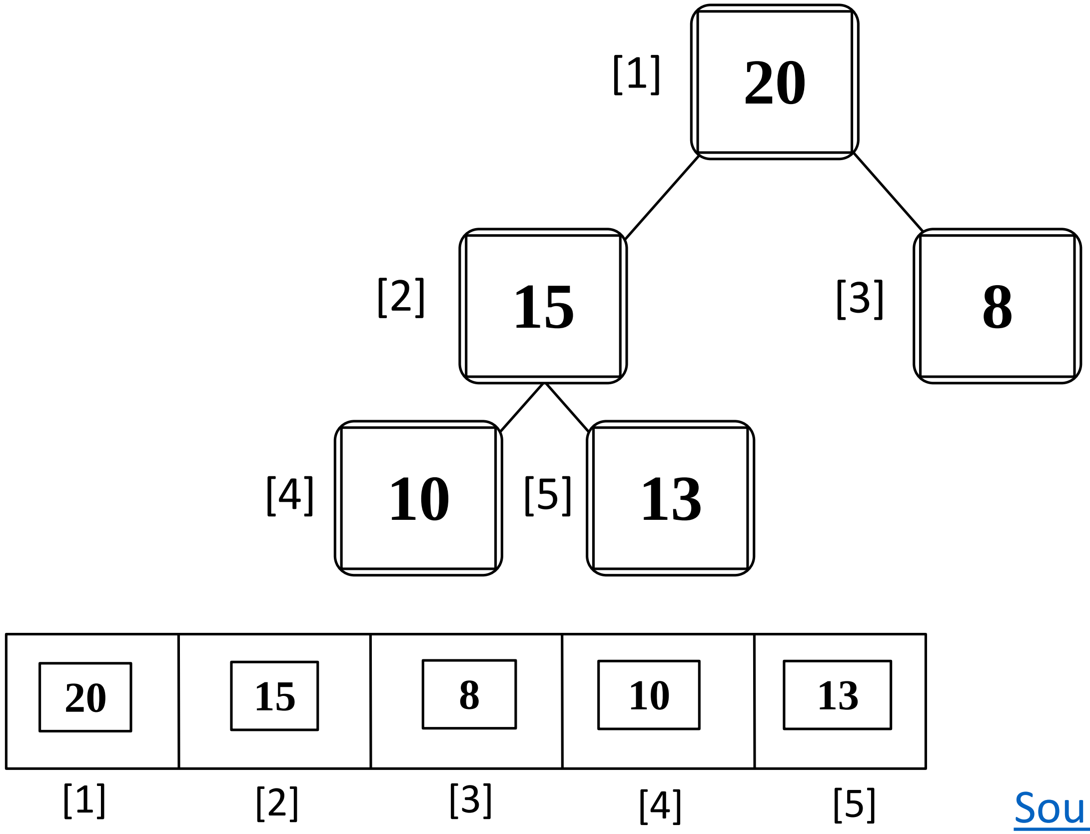
```
int main() {
  int A[tree_array_size];
  insert(A, 20);
  insert(A, 15);
  insert(A, 8);
  insert(A, 10);
  insert(A, 5);

  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));

  print_heap(A);

  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
```



Source

Max-Priority Queue

```c
int main() {
  int A[tree_array_size];
  insert(A, 20);
  insert(A, 15);
  insert(A, 8);
  insert(A, 10);
  insert(A, 5);

  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));

  print_heap(A);

  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
```



Source

Max-Priority Queue

```c
int main() {
  int A[tree_array_size];
  insert(A, 20);
  insert(A, 15);
  insert(A, 8);
  insert(A, 10);
  insert(A, 5);

  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));

  print_heap(A);

  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A));
```

```c
int maximum(int A[]) {
  return A[1];
}
```

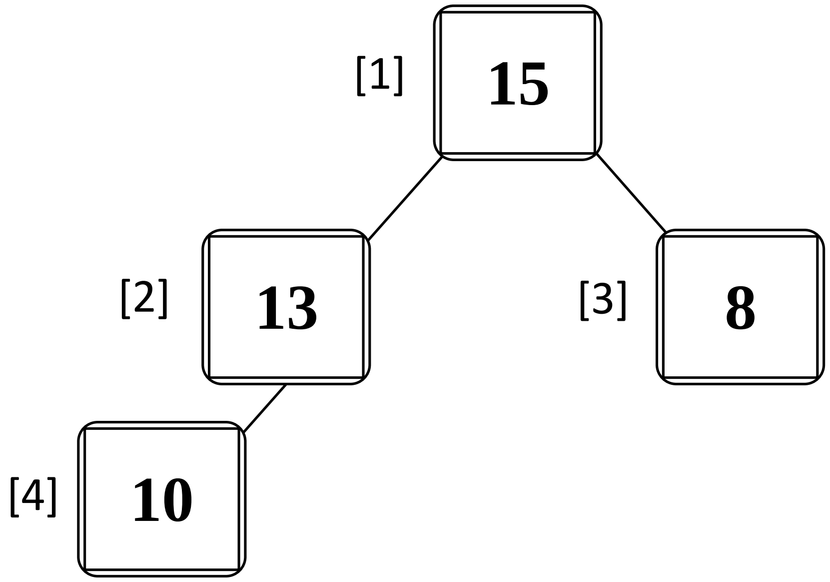
```c
int extract_max(int A[]) {
  int maxm = A[1];
  A[1] = A[heap_size];
  heap_size--;
  max_heapify(A, 1);
  return maxm;
}
```

Source

Max-Priority Queue

```
int main() {
    int A[tree_array_size];
    insert(A, 20);
    insert(A, 15);
    insert(A, 8);
    insert(A, 10);
    insert(A, 5);

    print_heap(A);

    increase_key(A, 5, 22);
    print_heap(A);

    decrease_key(A, 1, 13);
    print_heap(A);

    printf("%d\n\n", maximum(A));
    printf("%d\n\n", extract_max(A));

    print_heap(A);

    printf("%d\n", extract_max(A));
    printf("%d\n", extract_max(A));
    printf("%d\n", extract_max(A));
    printf("%d\n", extract_max(A));
```

```
int extract_max(int A[]) {
    int maxm = A[1];
    A[1] = A[heap_size];
    heap_size--;
    max_heapify(A, 1);
    return maxm;
}
```



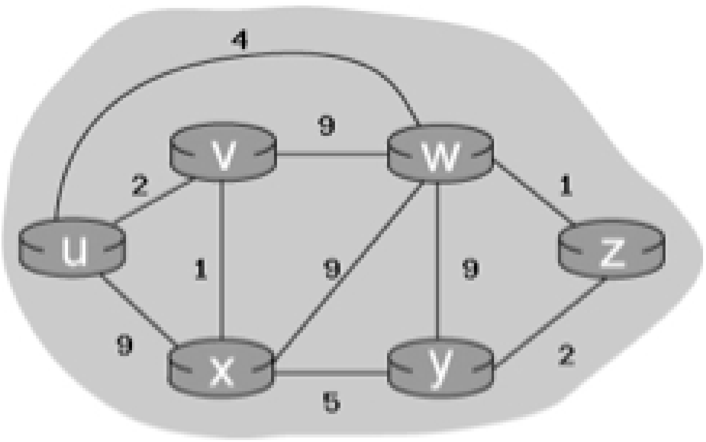
Source

Heap (Applications - Case)

- I used a heap many years ago to optimize a program for Bell Canada.
- The program took in forecasts of future demand for data transfer between nodes in a large network that spanned the country.
- The program could be configured in terms of the how to choose routes for the data transfer, with the objective of minimizing cost of the required equipment overall.
- As a simple example, imagine allowing each node to transfer directly to the destination node vs transmitting to a hub which would eventually route the data to it's destination.
    - **Glenn Reid** CEO - RJB Technology Inc.1999–present

Heap tress can be use for Djikstra's Algorithm i.e. It is used to find the shortest path between two nodes in a graph.

Heap (Applications - Case)

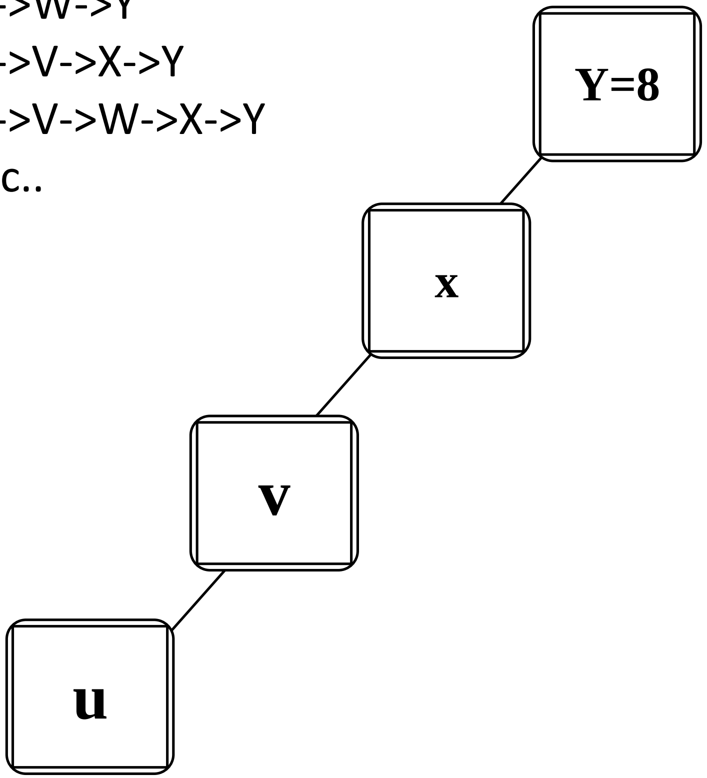


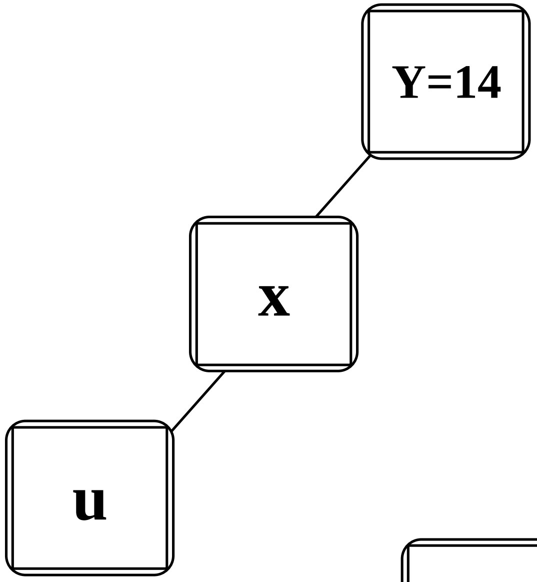
Source: u
Destination: y

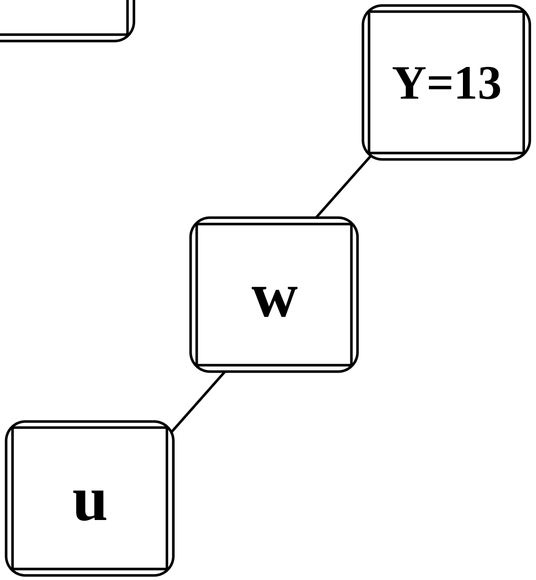
U -> Y Paths:

- U->X->Y
- U->W->Y
- U->V->X->Y
- U->V->W->X->Y
- Etc..

Nprime	D(u),p(u)	D(v),p(v)	D(w),p(w)	D(x),p(x)	D(y),p(y)	D(z),p(z)
w						
w						
w						
w						
w						
w						



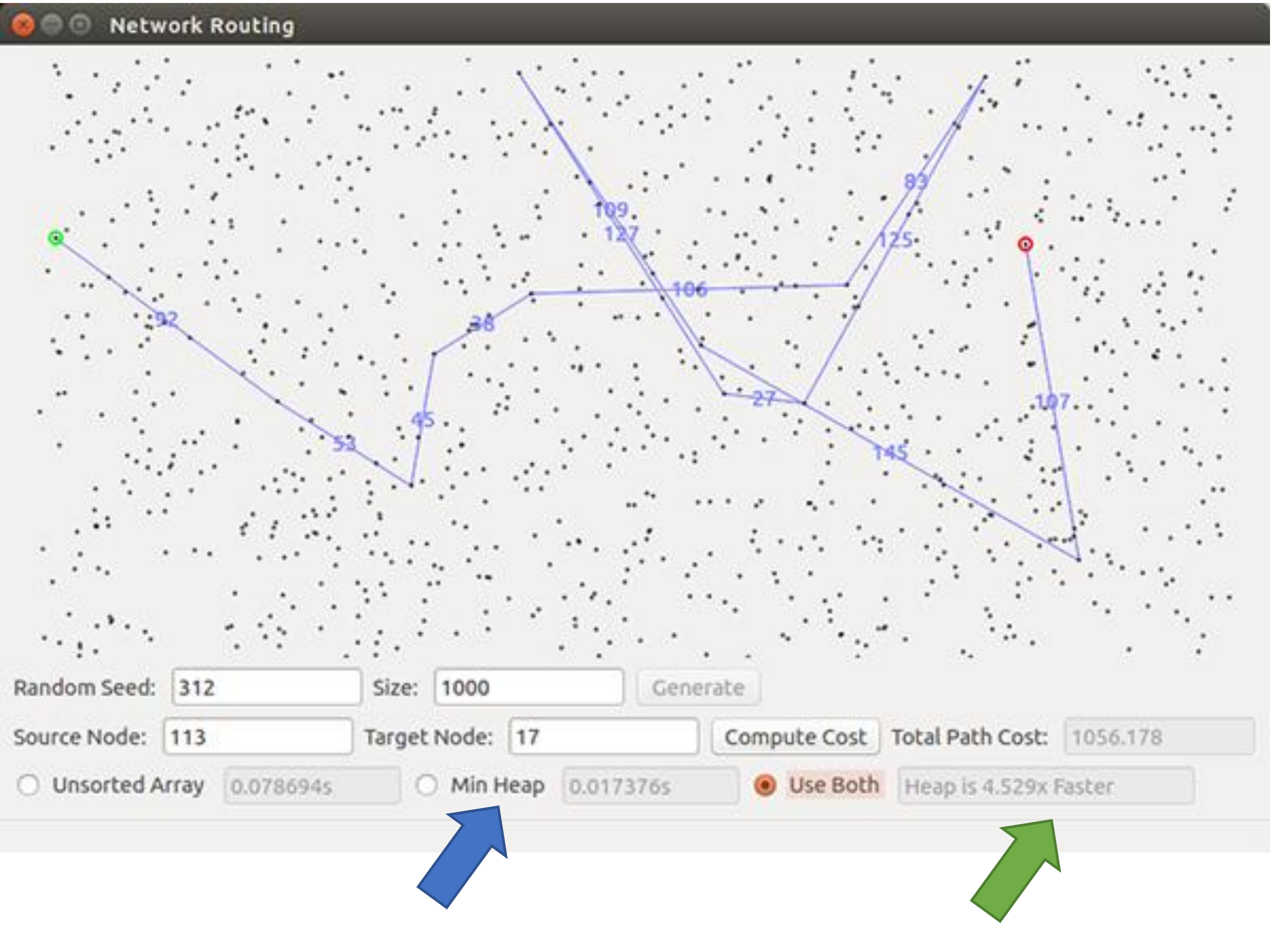


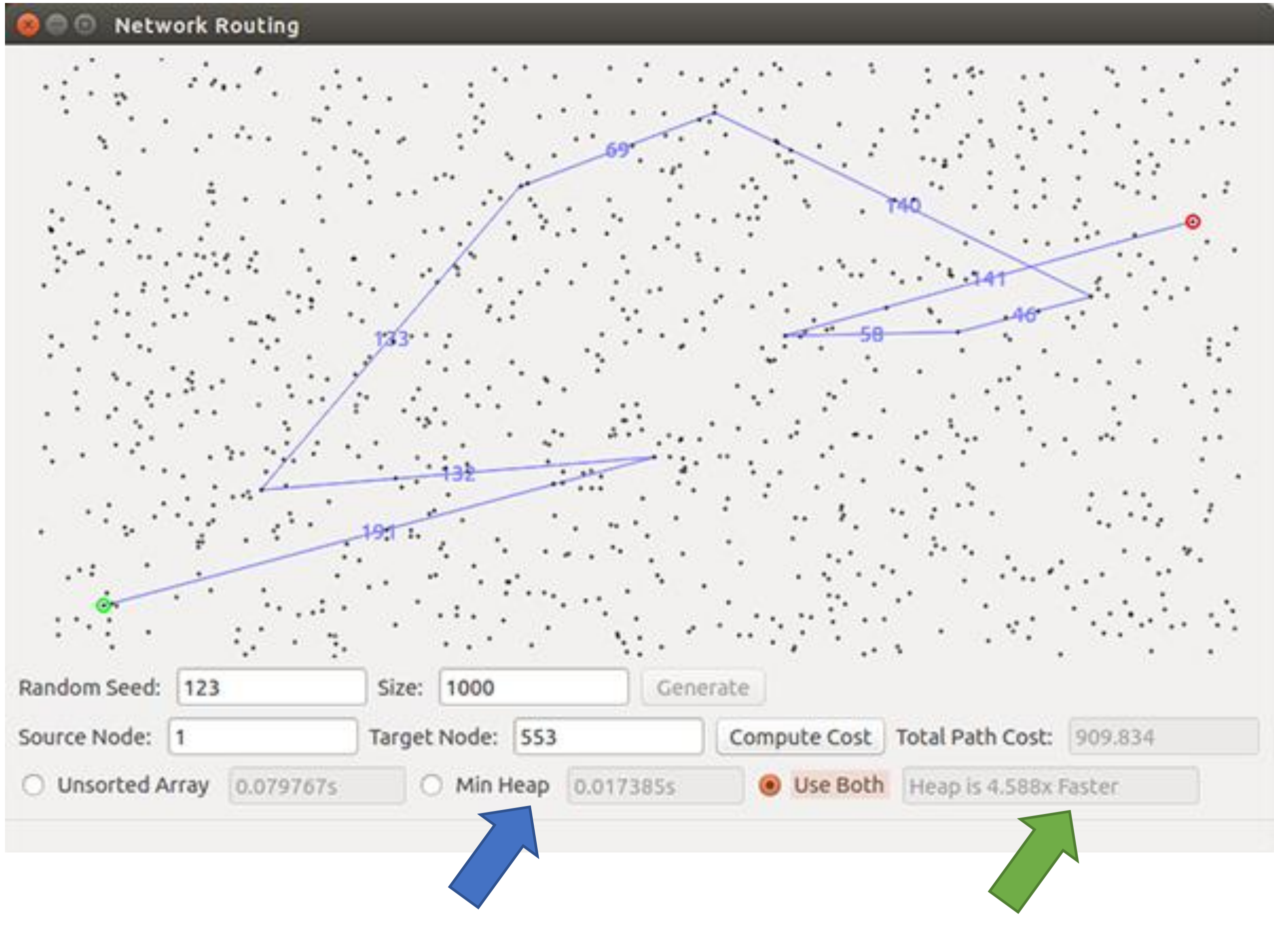


Network Routing (Source: Link)

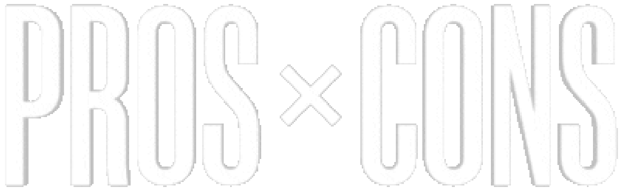
- Overview
  - In this project you will implement Dijkstra’s algorithm to find paths through a graph representing a network routing problem.
- Goals
  - Understand Dijkstra’s algorithm in the context of a real world problem (Lesson 12: Dijkstra).
  - Implement a priority queue with worst-case logarithmic operations.
  - Compare two different priority queue data structures for implementing Dijkstra’s and empirically verify their differences.
  - Understand the importance of proper data structures/implementations to gain the full efficiency potential of algorithms.

Network Routing (Source: Link)





Heap (Advantages and Disadvantages)



- **Advantages** of Heaps:
  - Fast access to maximum/minimum element (O(1))
  - Efficient Insertion and Deletion operations (O(log n))
  - Flexible size
  - Can be efficiently implemented as an array
  - Suitable for real-time applications
- **Disadvantages** of Heaps:
  - Not suitable for searching for an element other than maximum/minimum (O(n) in worst case)
  - Extra memory overhead to maintain heap structure
  - Slower than other data structures like arrays and linked lists for non-priority queue operations.

Building Huffman Tree (Variable Bit) using Heap

- Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

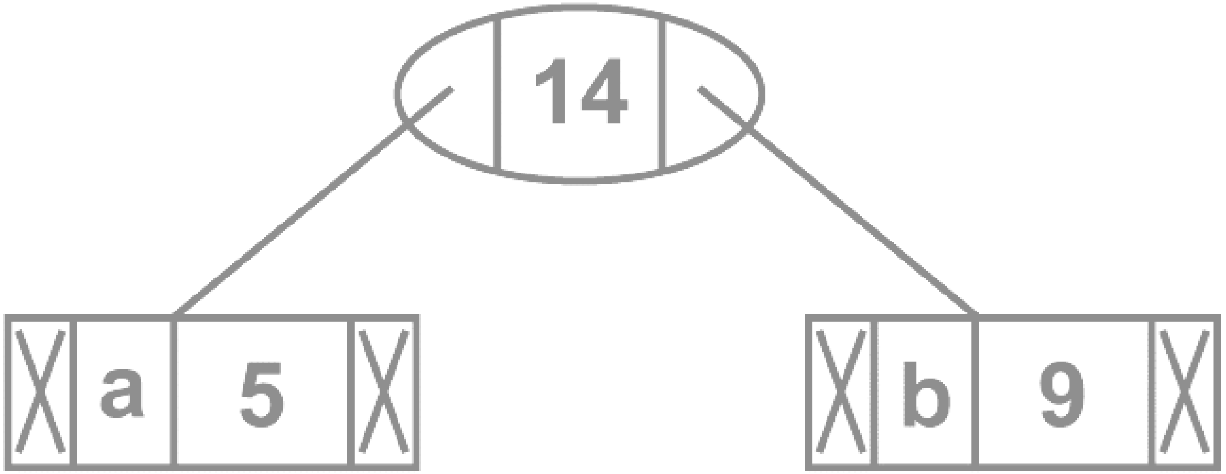
1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue.)
   A. The value of frequency field is used to compare two nodes in min heap.
   B. Initially, the least frequent character is at root
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies.
   A. Make the first extracted node as its left child and the other extracted node as its right child.
   B. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node.
   A. The remaining node is the root node and the tree is complete.

Building Huffman Tree using Heap

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

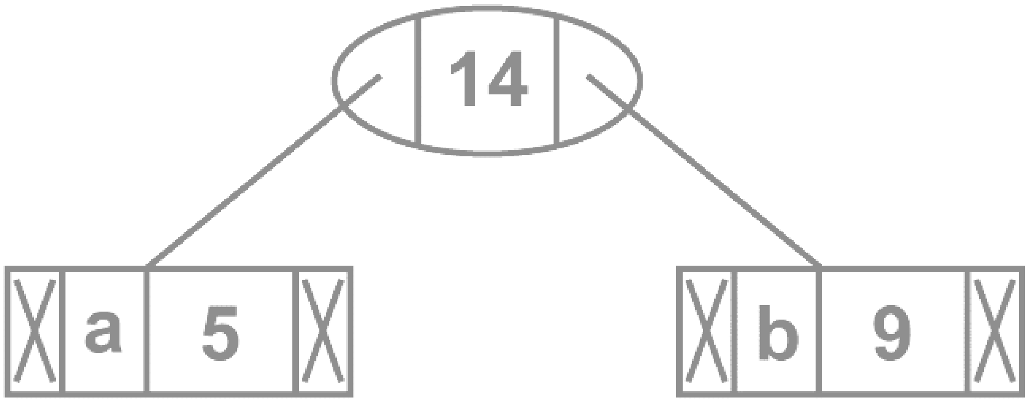


Building Huffman Tree using Heap

Character	Frequency
c	12
d	13
Int-Node	14
e	16
f	45

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25

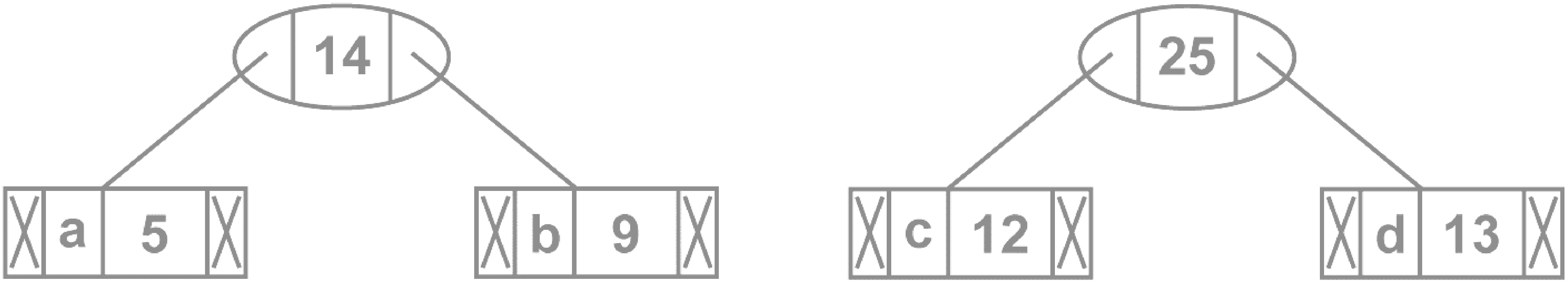


Building Huffman Tree using Heap

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

Character	Frequency
c	12
d	13
Int-Node	14
e	16
f	45

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25

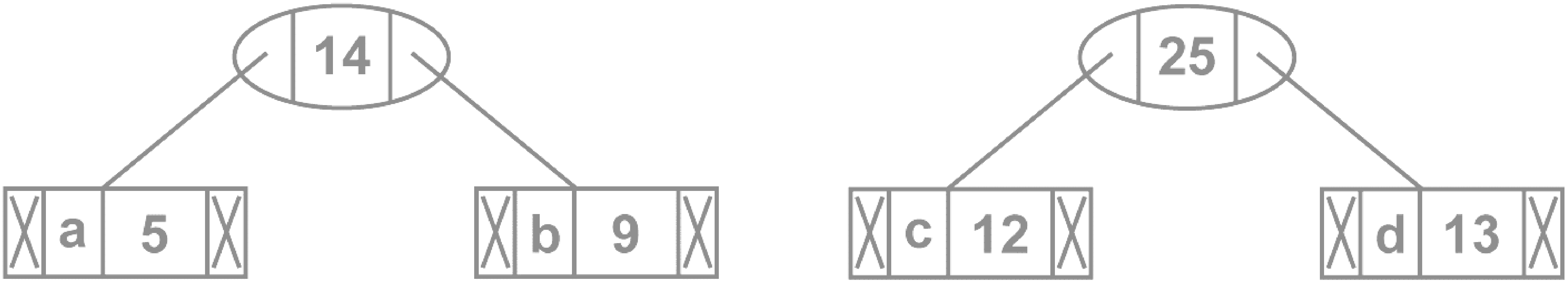


Building Huffman Tree using Heap

Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30

Character	Frequency
Int-Node	14
e	16
Int-Node	25
f	45

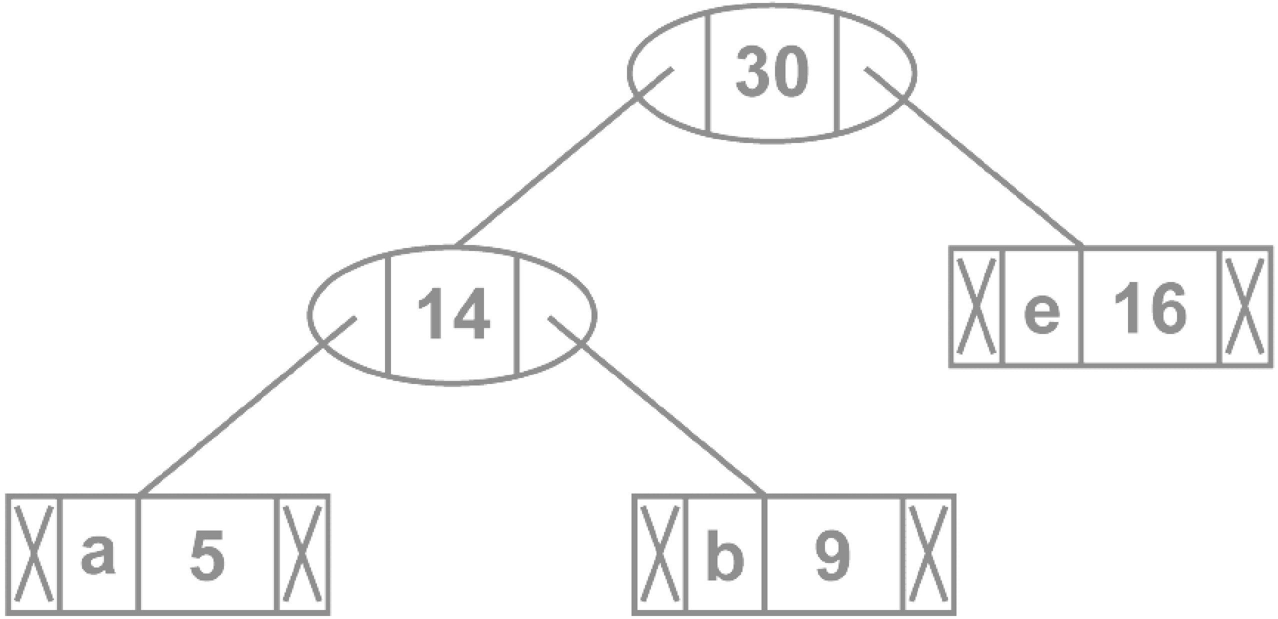


Building Huffman Tree using Heap

Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30

Character	Frequency
Int-Node	14
e	16
Int-Node	25
f	45

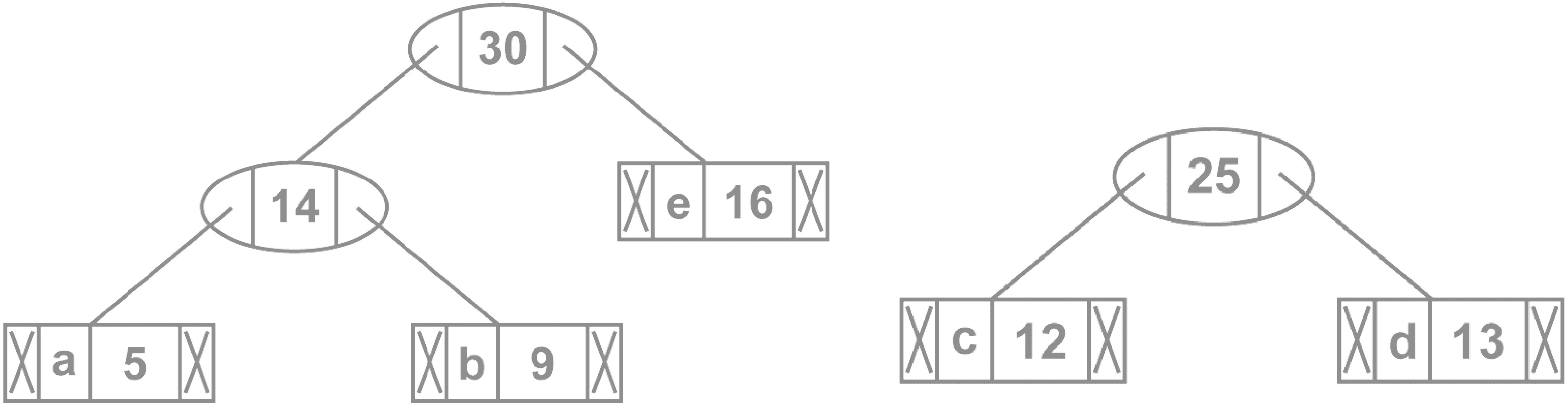


Building Huffman Tree using Heap

Now min heap contains 3 nodes.

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55

Character	Frequency
Int-Node	25
Int-Node	30
f	45

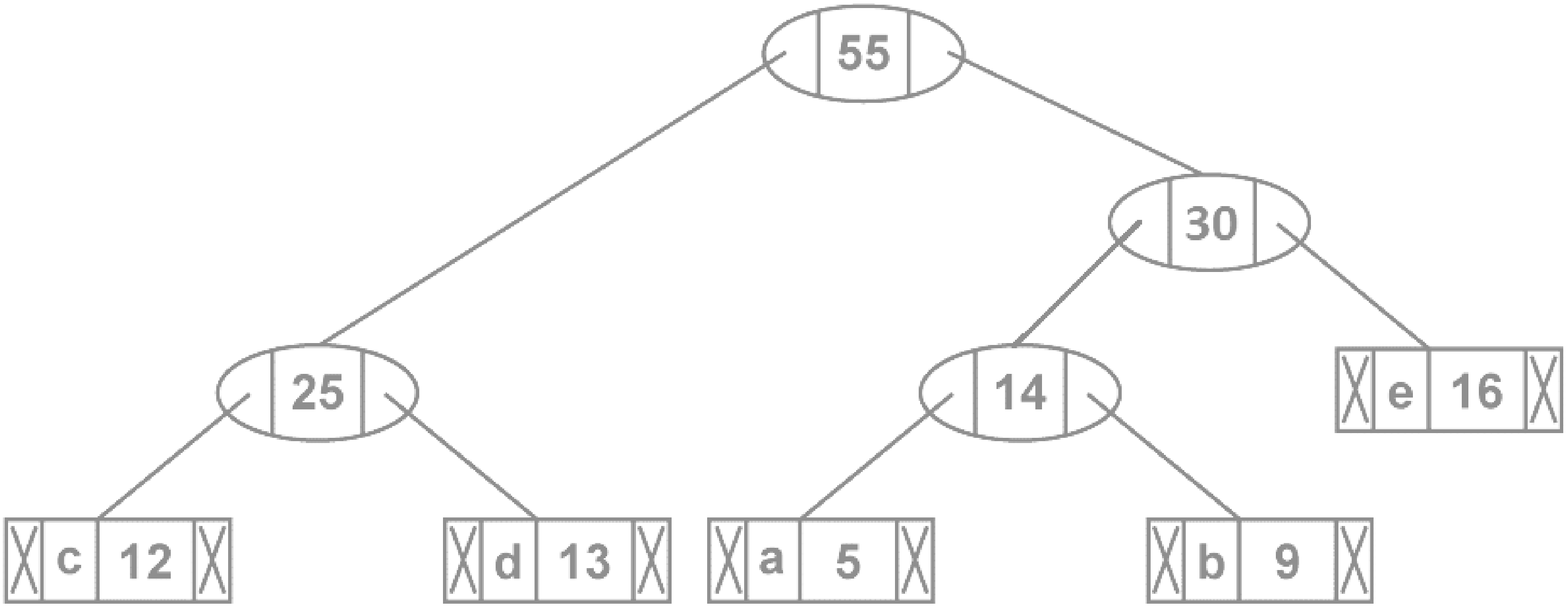


Building Huffman Tree using Heap

Character	Frequency
Int-Node	25
Int-Node	30
f	45

Now min heap contains 3 nodes.

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55

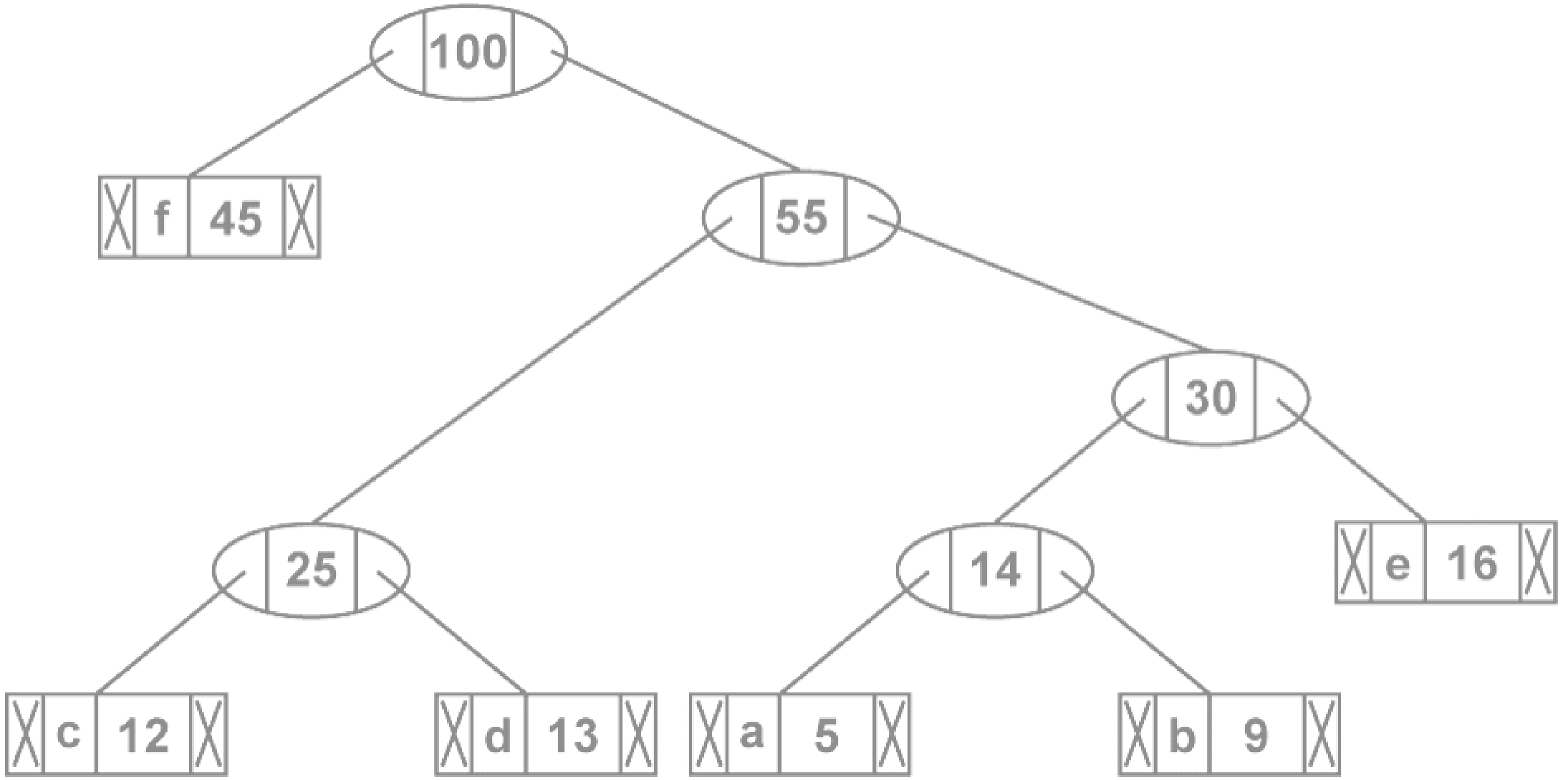


Building Huffman Tree using Heap

Now min heap contains 2 nodes.

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100

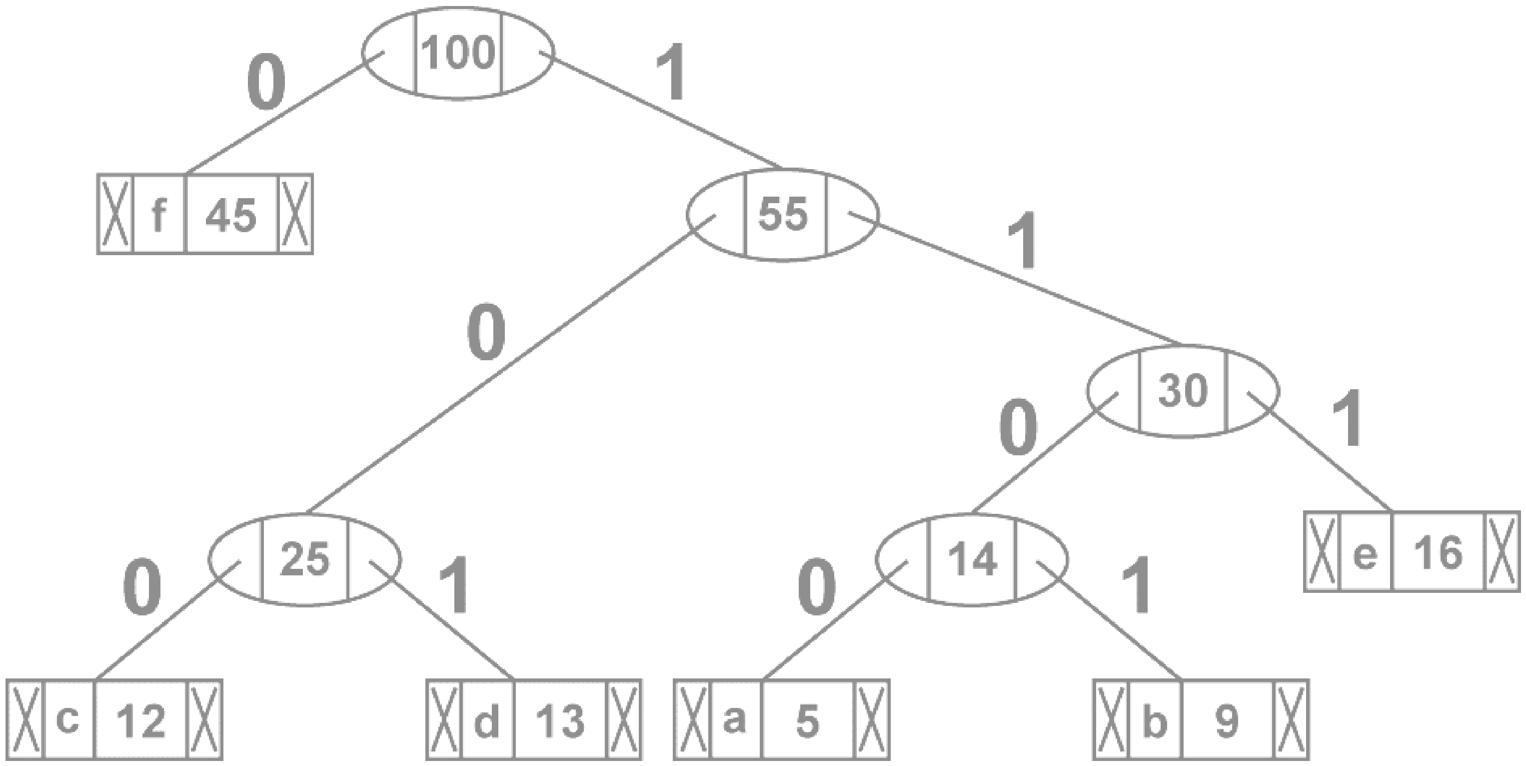
Character	Frequency
f	45
Int-Node	55



Building Huffman Tree using Heap

While moving to the left child, write 0 to the array.
While moving to the right child, write 1 to the array.

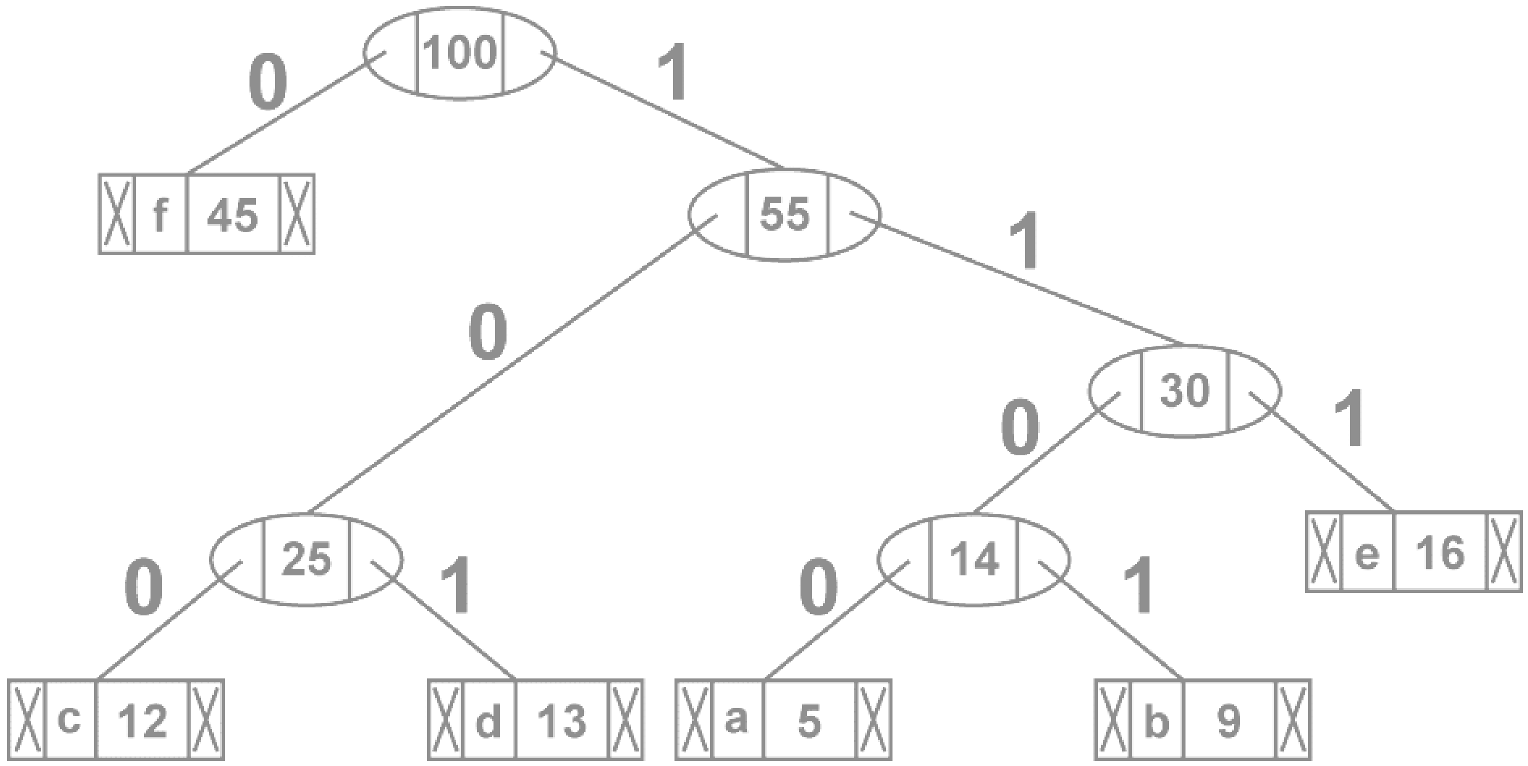
Character	Frequency
Int-Node	100



Building Huffman Tree using Heap

Char	Code	Freq	Bits = Code*Freq
a	1100	5	20
b	1101	9	36
c	100	12	36
d	101	13	39
e	111	16	48
f	0	45	45
Total			224

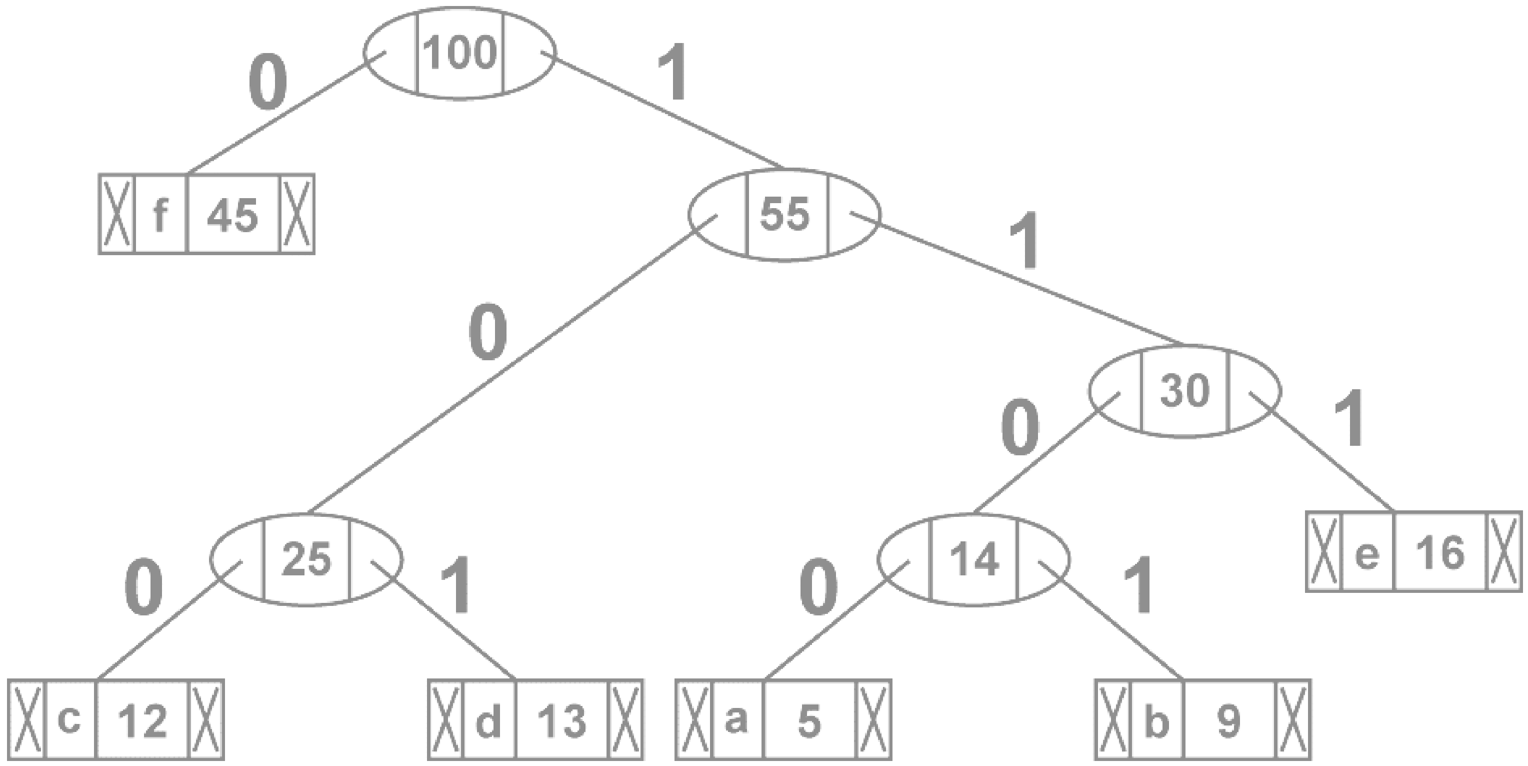
While moving to the left child, write 0 to the array.
While moving to the right child, write 1 to the array.



Building Huffman Tree using Heap

Char	Code	Freq	Bits = Code*Freq
a	1100	5	20
b	1101	9	36
c	100	12	36
d	101	13	39
e	111	16	48
f	0	45	45
Total			224

While moving to the left child, write 0 to the array.
While moving to the right child, write 1 to the array.

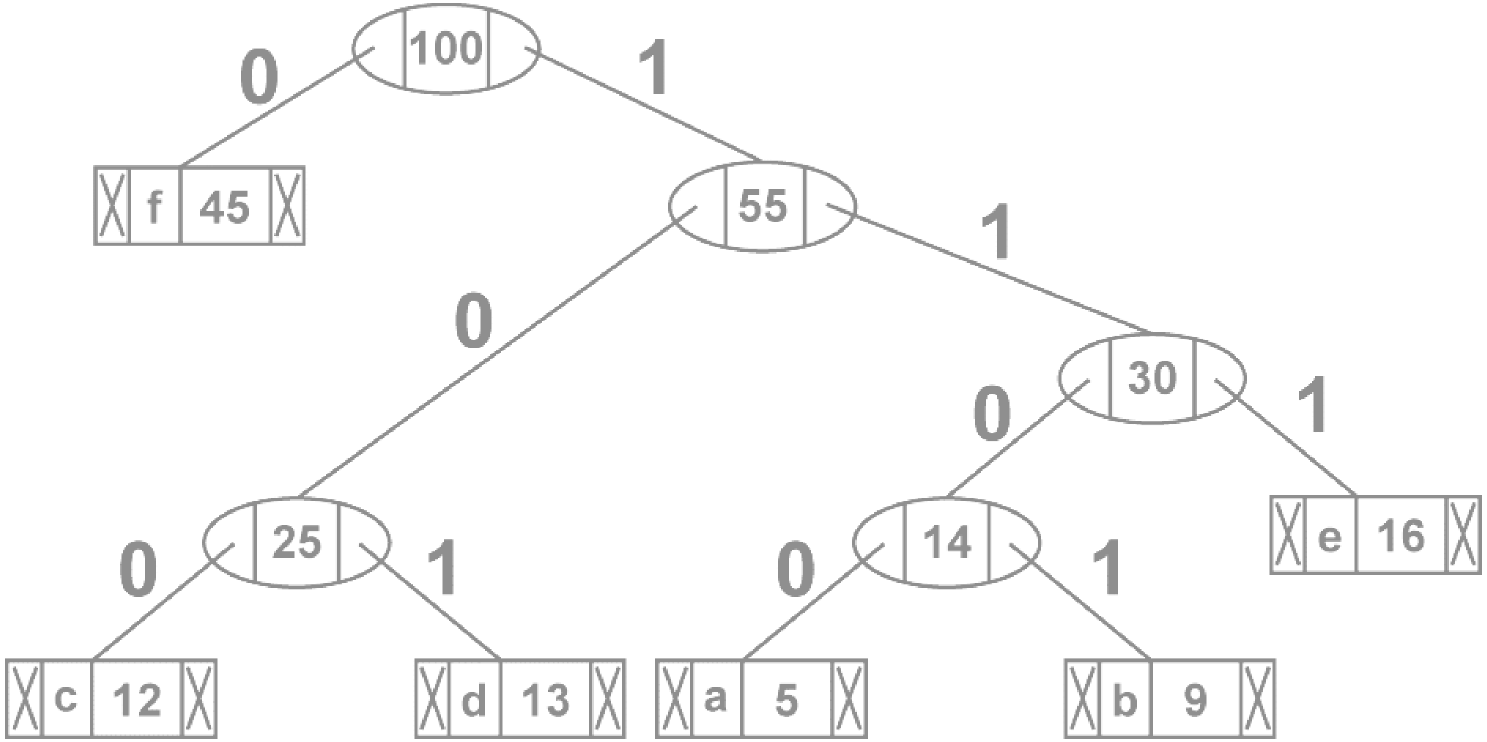



```
Huffman Encoding (Variable Bit)
Char | Freq
a    | 5
b    | 9
c    | 12
d    | 13
e    | 16
f    | 45
----------
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Building Huffman Tree using Heap

Fix Bit VS Variable Bit

- 2 bits = 00, 01, 10, 11 = 4 characters
- 3 bits = 000, 001, 010, 011, 100, 101, 110, 111 = 8 characters
- 2^n
  - $2^n \Rightarrow \mathrm{n} = 2 \Rightarrow 4$
  - $2^n \Rightarrow \mathrm{n} = 3 \Rightarrow 8$



Char	Code	Freq	Bits = Code*Freq
a	1100	5	20
b	1101	9	36
c	100	12	36
d	101	13	39
e	111	16	48
f	0	45	45
Total			224

Char	Code	Freq	Bits = Code*Freq
a	000	5	15
b	001	9	27
c	010	12	36
d	100	13	39
e	101	16	48
f	110	45	135
Total			300

Applications of Huffman Coding

Real-world examples of Huffman Coding in practice (Link)

- **Text Compression**
  - Huffman coding requires that it must know the distribution of the data before it can encode it. Adaptive Huffman coding is an alternative because it can build a Huffman coding tree and encode the data in just a single pass, but it is much more computationally demanding and slower than if the Huffman codes were already known.

- **Audio Compression**
  - Audio is another application area that benefits greatly from Huffman encoding when the scheme is required to be lossless.

method	bit-rate [kbps/channel]
with Huffman coding	47.3
without Huffman coding	56.0

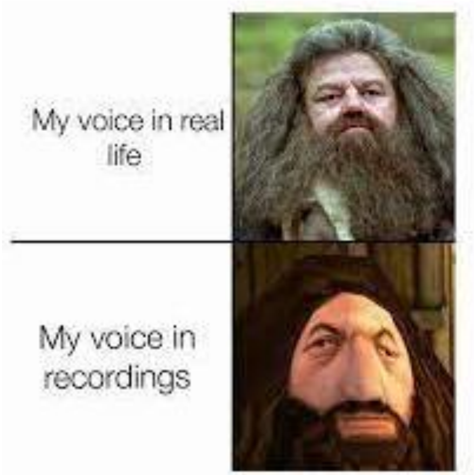


Table Source: Sampled-data audio signal compression with Huffman coding (IEEE Link)

Applications of Huffman Coding

Real-world examples of Huffman Coding

- Revisiting Huffman Coding: Toward Extreme Performance On Modern GPU Architectures (Link)
- Today's high-performance computing (HPC) applications are producing vast volumes of data, which are challenging to store and transfer efficiently during the execution, such that data compression is becoming a critical technique to mitigate the storage burden and data movement cost.
- Huffman coding is arguably the most efficient Entropy coding algorithm in information theory, such that it could be found as a fundamental step in many modern compression algorithms such as DEFLATE.
- On the other hand, today's HPC applications are more and more relying on the accelerators such as GPU on supercomputers, while Huffman encoding suffers from low throughput on GPUs, resulting in a significant bottleneck in the entire data processing.
- In this paper, we propose and implement an efficient Huffman encoding approach based on modern GPU architectures, which addresses two key challenges:
    1) how to parallelize the entire Huffman encoding algorithm, including codebook construction, and
    2) how to fully utilize the high memory-bandwidth feature of modern GPU architectures. The detailed contribution is fourfold.