Course Code: CL-2001	Course : Data Structures - Lab
Instructor(s) :	Eman Shahid

LAB - 11

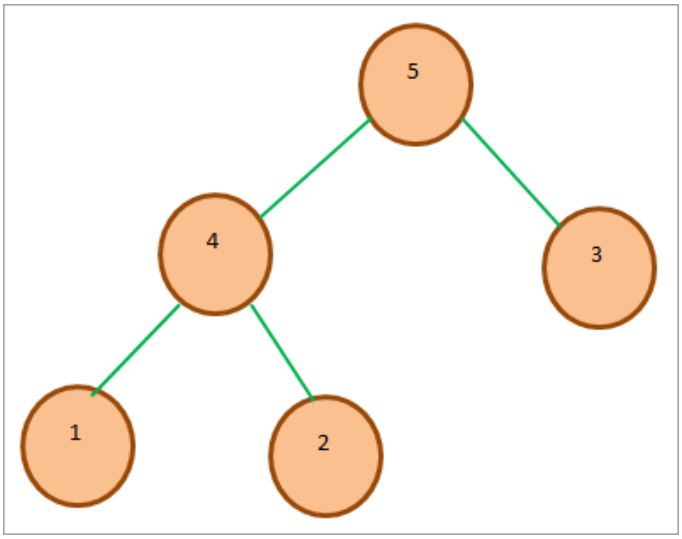
Heaps, Priority Queues and Huffman Coding

Heaps

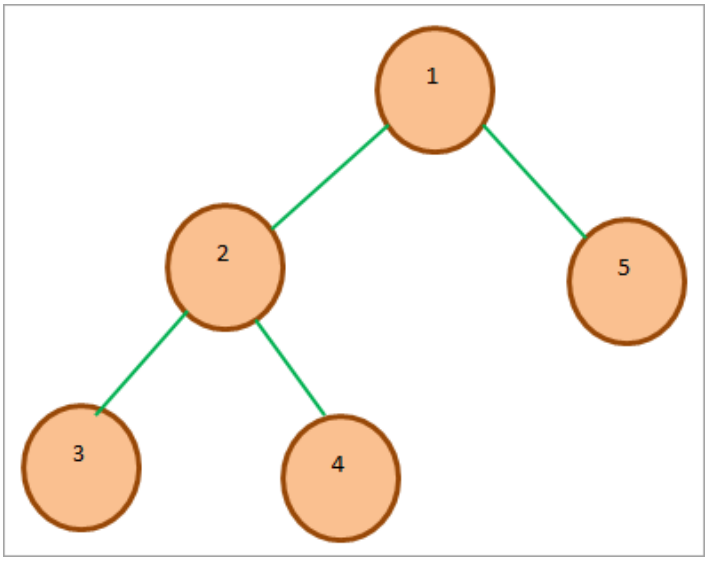
A heap is a special data structure in Java. A heap is a tree-based data structure and can be classified as a complete binary tree. All the nodes of the heap are arranged in a specific order.

In the heap data structure, the root node is compared with its children and arranged according to the order. So if a is a root node and b is its child, then the property, key (a)>= key (b) will generate a max heap.

The above relation between the root and the child node is called as "Heap Property".
#1) Max-Heap: In a Max-Heap the root node key is the greatest of all the keys in the heap. It should be ensured that the same property is true for all the subtrees in the heap recursively.



#2) Min-Heap: In the case of a Min-Heap, the root node key is the smallest or minimum among all the other keys present in the heap. As in the Max heap, this property should be recursively true in all the other subtrees in the heap.



Binary Heap

A binary heap fulfills the below properties:

- A binary heap is a complete binary tree. In a complete binary tree, all the levels except the last level are completely filled. At the last level, the keys are as far as left as possible.
- It satisfies the heap property. The binary heap can be max or min-heap depending on the heap property it satisfies.
- A binary heap is normally represented as an Array. As it is a complete binary tree, it can easily be represented as an array. Thus in an array representation of a binary heap, the root element will be A[0] where A is the array used to represent the binary heap.

So in general for any ith node in the binary heap array representation, A[i], we can represent the indices of other nodes as shown below.

A [(i-1)/2]	Represents the parent node
A[(2*i)+1]	Represents the left child node
A[(2*i)+2]	Represents the right child node

Min-Heap

Operations:

We can perform the following three operations in Min heap:

insertNode()

We can perform insertion in the Min heap by adding a new key at the end of the tree. If the value of the inserted key is smaller than its parent node, we have to traverse the key upwards for fulfilling the heap property. The insertion process takes O(log n) time.

extractMin()

It is one of the most important operations which we perform to remove the minimum value node, i.e., the root node of the heap. After removing the root node, we have to make sure that heap property should be maintained. The extractMin() operation takes O(Logn) time to remove the minimum element from the heap.

getMin()

The getMin() operation is used to get the root node of the heap, i.e., minimum element in O(1) time.

Agorithm Pseudocode:

```
proceduredesign_min_heap
Array arr: of size n => array of elements
// call min_heapify procedure for each element of the array to form min heap
repeat for (k = n/2 ; k >= 1 ; k--)
    call procedure min_heapify (arr, k);
```

```
procedureremin_heapify (vararr[ ] , var k, varn)
{
varleft_child  = 2*k;
varright_child = 2*k+1;
var smallest;
if(left_child<= n and arr[left_child ] <arr[ k ] )
smallest = left_child;
else
smallest = k;
if(right_child<= n and arr[right_child ] <arr[smallest] )
smallest = right_child;
if(smallest != k)
   {
swaparr[ k ] and  arr[ smallest ]);
callmin_heapify (arr, smallest, n);
   }
}
```

Max-Heap
Operations:

insertNode()

We can perform insertion in the Max heap by adding a new key at the end of the tree. If the value of the inserted key is greater than its parent node, we have to traverse the key upwards for fulfilling the heap property. The insertion process takes O(log n) time.

extractMax()

It is one of the most important operations which we perform to remove the maximum value node, i.e., the root node of the heap. After removing the root node, we have to make sure that heap property should be maintained. The extractMax() operation takes O(Log n) time to remove the maximum element from the heap.

getMax()

The getMax() operation is used to get the root node of the heap, i.e., maximum element in O(1) time

Agorithm Pseudocode:

```
Array arr: of size n => array of elements
// call min_heapify procedure for each element of the array to form max heap
repeat for (k = n/2 ; k >= 1 ; k--)
```

```
   call procedure max_heapify (arr, k);
proceduremin_heapify (vararr[ ] , var k, varn)
{
varleft_child  = 2*k + 1;
varright_child = 2*k+ 2;
if(left_child<= n and arr[left_child] >arr[ largest ] )
largest = left_child;
else
largest = k;
if(right_child<= n and arr[right_child] >arr[largest] )
largest = right_child;
if(largest != k)
   {
swaparr[ k ] and  arr[ largest ]);
callmax_heapify (arr, largest, n);
   }
}
```

Heap Sort Algorithm

First convert the array into heap data structure using heapify, than one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.

Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.

- Swap: Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
- Remove: Reduce the size of the heap by 1.
- Heapify: Heapify the root element again so that we have the highest element at root.
- The process is repeated until all the items of the list are sorted.

Priority Queue:

Priority queue is a type of queue in which every element has a key associated to it and the queue returns the element according to these keys, unlike the traditional queue which works on first come first serve basis.

Thus, a max-priority queue returns the element with maximum key first whereas, a min-priority queue returns the element with the smallest key first.

Maximum key is returned first in the max-queue

Minimum key is returned first in the min-queue

Priority queues are used in many algorithms like Huffman Codes, Prim's algorithm, etc. It is also used in scheduling processes for a computer, etc.

Heaps are great for implementing a priority queue because of the largest and smallest element at the root of the tree for a max-heap and a min-heap respectively. We use a max-heap for a max-priority queue and a min-heap for a min-priority queue.

There are mainly 4 operations we want from a priority queue:

1. Insert → To insert a new element in the queue.

2. Maximum/Minimum → To get the maximum and the minimum element from the max-priority queue and min-priority queue respectively.

3. Extract Maximum/Minimum → To remove and return the maximum and the minimum element from the max-priority queue and min-priority queue respectively.

4. Increase/Decrease key → To increase or decrease key of any element in the queue.

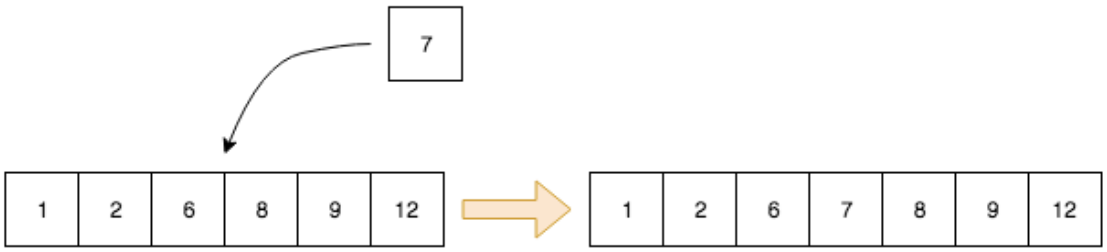
A priority queue stores its data in a specific order according to the keys of the elements. So, inserting a new data must go in a place according to the specified order. This is what the insert operation does.

The entire point of the priority queue is to get the data according to the key of the data and the Maximum/Minimum and Extract Maximum/Minimum does this for us.

Implementation using arrays

A priority queue can be implemented using data structures like arrays, linked lists, or heaps. The array can be ordered or unordered.

Using an ordered array
The item is inserted in such a way that the array remains ordered i.e. the largest item is always in the end. The insertion operation is illustrated in figure. The item with priority 7 is inserted between the items with priorities 6 and 8.



Code:

```
// insert an item at the appropriate position of the
// queue so that the queue is always ordered
void enqueue(int item) {
	// Check if the queue is full
	if (n == MAX_SIZE - 1) {
		printf("%s\n", "ERROR: Queue is full");
		return;
	}

	int i = n - 1;
	while (i >= 0 && item < queue[i]) {
		queue[i + 1] = queue[i];
		i--;
	}
	queue[i + 1] = item;
	n++;
}

// remove the last element in the queue
int dequeue() {
	int item;
	// Check if the queue is empty
	if (n == 0) {
		printf("%s\n", "ERROR: Queue is empty");
		return -999999;
	}
	item = queue[n - 1];
	n = n - 1;
	return item;
}
```

Huffman Coding

It is a lossless data compression mechanism. It is also known as data compression encoding. It is widely used in image (JPEG or JPG) compression.

Is it possible to reduce the amount of space required to store a character?

Yes, it is possible by using variable-length encoding. In this mechanism, we exploit some characters that occur more frequently in comparison to other characters. In this encoding technique, we can represent the same piece of text or string by reducing the number of bits.

Huffman Encoding

Huffman encoding implements the following steps.

- It assigns a variable-length code to all the given characters.
- The code length of a character depends on how frequently it occurs in the given text or string.
- A character gets the smallest code if it frequently occurs.
- A character gets the largest code if it least occurs.
- Huffman coding follows a prefix rule that prevents ambiguity while decoding. The rule also ensures that the code assigned to the character is not treated as a prefix of the code assigned to any other character.

There are the following two major steps involved in Huffman coding:

First, construct a Huffman tree from the given input string or characters or text.
Assign, a Huffman code to each character by traversing over the tree.
Let's brief the above two steps.

Huffman Tree

Step 1: For each character of the node, create a leaf node. The leaf node of a character contains the frequency of that character.

Step 2: Set all the nodes in sorted order according to their frequency.

Step 3: There may exist a condition in which two nodes may have the same frequency. In such a case, do the following:

1. Create a new internal node.
2. The frequency of the node will be the sum of the frequency of those two nodes that have the same frequency.
3. Mark the first node as the left child and another node as the right child of the newly created internal node.

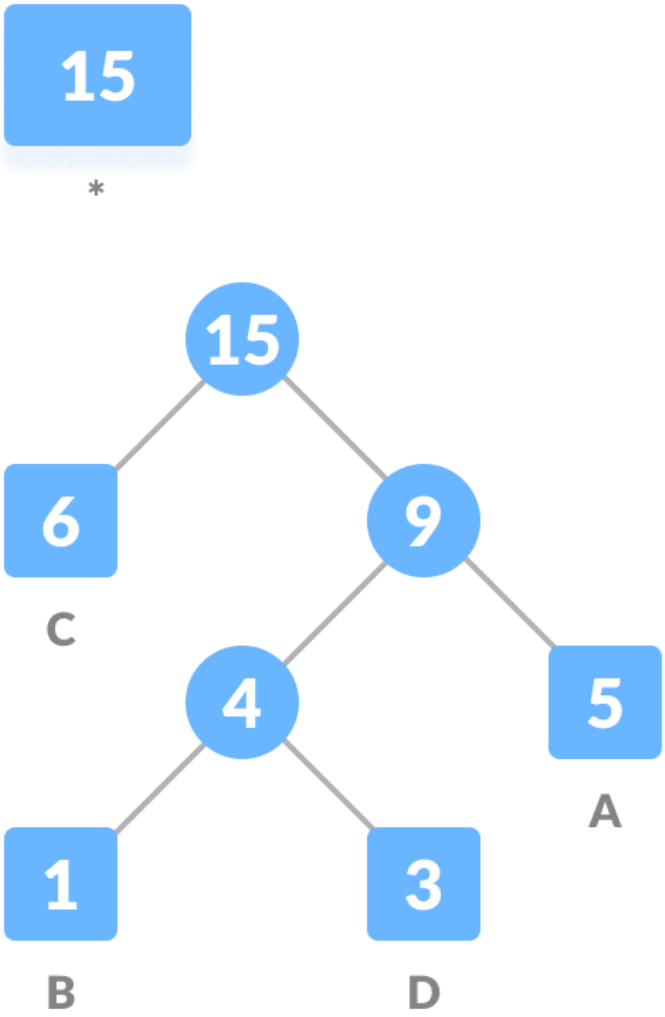
Step 4: Repeat step 2 and 3 until all the node forms a single tree. Thus, we get a Huffman tree.

Huffman Decoding

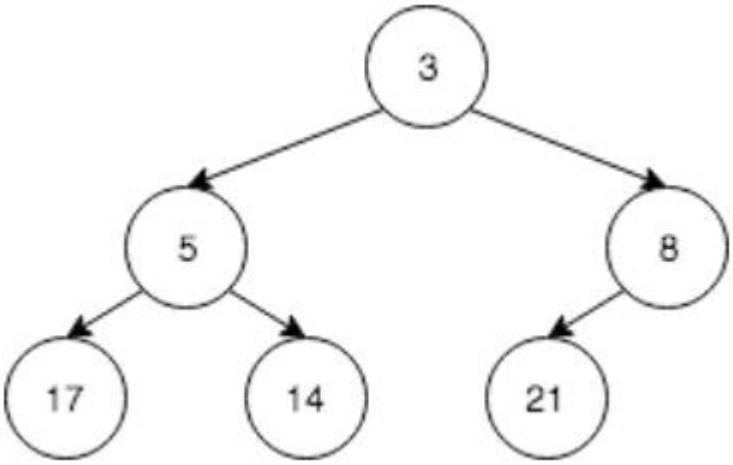
Huffman decoding is a technique that converts the encoded data into initial data. As we have seen in encoding, the Huffman tree is made for an input string and the characters are decoded based on their position in the tree. The decoding process is as follows:

- Start traversing over the tree from the root node and search for the character.
- If we move left in the binary tree, add 0 to the code.
- If we move right in the binary tree, add 1 to the code.

The child node holds the input character. It is assigned the code formed by subsequent 0s and 1s.



Lab Tasks:

1. 

   Suppose 7 is inserted into the heap. What will 7's left and right children be?
   Left child =? Right Child =?
   First build the heap and then display the right and left child for node 7.
   Then apply heapsort and display the array in descending order.

2. Given an array representation of min Heap, convert it to max Heap.

3. Job scheduling helps the device to run the most important programs first. Those jobs are stored in either a high-priority queue or a low-priority queue, depending on how important the program is.

   For example, processes that are fundamental to how the device operates (e.g. displaying things to the screen, dealing with system input and output) have a higher priority than user-installed applications (e.g. web browsers, music playing app, calculator app). At any given time, jobs waiting in the high-priority queue will be executed first, in the order that they were requested in. If there are no high-priority jobs waiting to be executed, then the jobs in the low-priority queue can be executed.

   Implement the following:
   - Add code to the main() function so that every time a new job is created (i.e. every time get_job is called), that job object is enqueued to the appropriate queue: high_priority_queue or low priority_queue.
   - Create a high_priority method to check if a job is high-priority (True) or low-priority (False). Notice that get_job may also return no job (None) which will NOT go into either queue, so that must be checked for as well.
   - Complete the main() function so that whenever a process has finished indicated (when process_complete returns True), a new process is started by dequeuing a job from the appropriate queue. i.e. If there is at least one job in the high-priority queue, it should be dequeued and assigned to the current_job variable. However, if the high-priority queue is empty and there is at least one job in the low-priority

queue, then it should be dequeued and assigned to the current_job variable. If a job has been successfully dequeued from either queue, the process running variable should be set to True.

Additional Assingment

4. Write a program for Huffman Coding. Your program should do the following:
   l) Accept a text message (could be multiple lines of text)
   2) Create a Huffman tree for this message
   3) Create a code table
   4) Encode the message into binary
   5) Decode the message from binary back to text
   If the message is short, the program should be able to display the Huffman tree after creating it. You can assume there are only letters and spaces in the message. You can use String variables to store binary numbers as arrangements of the characters 1 and 0.