Data Structure and Algorithm Assignment-1

To- Prof,Aditi Saran-JNU ,SC&SS

By-RISHI ANAND ARYA-M.Tech [Data Science]

**Task-1**

===============================================================

Consider a processor that is to be assigned jobs in order of time (Shortest Job First).

Maintain a priority queue.

Perfom insert & delete operations.

a) Maintain a FIFO queue.

For deletion, select shortest job (highest priority element), delete it and re-arrang the queue by shifting elements.

Insertion:

Description:

-> This function take jobs from list and insert them one-by-one into queue.

-> Time complexity is, O(n).

Input(s):

1. List of jobs

2. Total number of jobs

Output(s):

1. Queue of jobs in the order of arrival

**Algorithm:**

**insert(list, length){**

**for(i = 1 to length){**

**queue[i] = list[i]**

**}**

**}**

**Deletion:**

**Description:**

**-> This function, pick the minimum one from queue and delete it.**

**-> By deletion it means, shifting all the jobs to its left side and appending the picked one at end.**

**-> Time complexity is, O(n \* (n + n)) ~ O(n ^ 2).**

**Input(s):**

**1. Queue of jobs in the order of arrival**

**2. Total number of jobs**

**Output(s):**

**1. Empty queue/queue of jobs in descending order of burst time**

**Algorithm:**

**delete(queue, length){**

**for (i = 1 to length){**

**minJob = queue[i]**

**minJobIndex = i**

**for(j = 2 to length - i){**

**if(queue[j] < minJob){**

**minJob = queue[j]**

**minJobIndex = j**

**}**

**}**

**for(j = 1 to length){**

**if(j + 1 < length){**

**queue[j] = queue[j + 1]**

**} else {**

**queue[j] = minJob**

**}**

**}**

**}**

**}**

b) 1. Use insertion sort to maintain a sorted array.

For insertion, place element at proper position.

For deletion, delete first element and re-arrange the queue by shifting elements.

Insertion:

Description:

-> This function, pick job one-by-one from list, find its appropriate position in queue, shift jobs from that position to right if any and insert the picked one there.

-> Here, queue will be sorted in ascending order after each iteration.

-> Time complexity is, O(n \* (n + n)) ~ O(n ^ 2).

Input(s):

1. List of jobs

2. Total number of jobs

Output(s):

1. Queue of jobs in ascending order of burst time

**Algorithm:**

**insert(list, length){**

**for(i = 1 to length){**

**j = 1**

**while((j < i) & (list[i] >= queue[j])){**

**j = j + 1**

**}**

**index = j**

**for(j = i to index + 1){**

**queue[j] = queue[j - 1]**

**}**

**queue[index] = list[i]**

**}**

**}**

**Deletion:**

**Description:**

**-> This function, pick the first job since it is the minimum one and delete it from the queue.**

**-> By deletion it means, shifting all the jobs to its left side and appending the picked one at end.**

**-> Time complexity is, O(n ^ 2).**

**Input(s):**

**1. Queue of jobs in ascending order of burst time**

**2. Total number of jobs**

**Output(s):**

**1. Empty queue/queue of jobs in ascending order of burst time**

**Algorithm:**

**delete(queue, length){**

**for(i = 1 to length){**

**minJob = queue[i]**

**for(j = 1 to length){**

**if(j + 1 < length){**

**queue[j] = queue[j + 1]**

**} else {**

**queue[j] = minJob**

**}**

**}**

**}**

**}**

2. Maintain queue in reverse order.

Delete last element.

Insertion:

Description:

-> This function, pick job one-by-one from list, find its appropriate position in queue, shift jobs from that position to left if any and insert the picked one there.

-> Here, queue will be sorted in descending order after each iteration.

-> Time complexity is, O(n \* (n + n)) ~ O(n ^ 2).

Input(s):

1. List of jobs

2. Total number of jobs

Output(s):

1. Queue of jobs in descending order of burst time

**Algorithm:**

**insert(list, length){**

**for(i = 1 to length){**

**j = length - 1**

**while((j > length - i) & (queue[j] <= list[i])){**

**j = j - 1**

**}**

**index = j**

**for(j = length - i to index){**

**queue[j] = queue[j + 1]**

**}**

**queue[index] = list[i]**

**}**

**}**

**Deletion:**

**Description:**

**-> This function, delete the minimum one, which will be at the last in queue of remaining elements.**

**-> Time complexity will be O(n).**

**Input(s):**

**1. Queue of jobs in descending order of burst time**

**2. Total number of jobs**

**Output(s):**

**1. Empty queue/queue of jobs in descending order of burst time**

**Algorithm:**

**delete(queue, length){**

**for(i = length to 1){**

**queue[i] = queue[i]**

**}**

**}**

c) Use heap to maintain priority queue.

For insertion, insert the element in a heap so that heap property is maintained.

For deletion, delete first element. Delete and copy last element to first position, and re-heapify to arrange the first element in heap.

Insertion:

Description:

-> The function insert, create a heap and copy all elements of list into it.

-> Then it will call the heapify function which will convert the heap into min-heap.

-> Min-heap is the complete binary tree structure in which every parent node must be less than its children.

-> Time complexity will be O(n + log(n)) ~ O(n)

Input(s):

1. List of jobs

2. Total number of jobs

Output(s):

1. Min heap of jobs

**Deletion:Algorithm:**

**heapify(heap, temp, parent, length){**

**child = parent \* 2**

**while(child <= length){**

**if((child + 1 <= length) & (heap[child] > heap[child + 1])){**

**child = child + 1**

**}**

**if(temp > heap[child]){**

**heap[child / 2] = heap[child]**

**child = child \* 2**

**} else {**

**break**

**}**

**}**

**heap[child / 2] = temp**

**}**

**insert(list, length){**

**for(i = 1 to length){**

**heap[i] = list[i]**

**}**

**for(i = length / 2 to 1){**

**heapify(heap, heap[i], i, length)**

**}**

**}**

**Description:**

**-> The function delete, swap the first element of min-heap, which is the minimum one with the last element of min-heap and then call heapify function.**

**-> The heapify function, take all the remaining elements of min-heap and heapify it. This process is iterative for each remaining jobs of min-heap.**

**-> Time complexity will be O(log(n)).**

**Input(s):**

**1. Min heap of jobs**

**2. Total number of jobs**

**Output(s):**

**1. Empty heap/max heap of jobs**

**Algorithm:**

**heapify(heap, temp, parent, length){**

**child = parent \* 2**

**while(child <= length){**

**if((child + 1 <= length) & (heap[child] > heap[child + 1])){**

**child = child + 1**

**}**

**if(temp > heap[child]){**

**heap[child / 2] = heap[child]**

**child = child \* 2**

**} else {**

**break**

**}**

**}**

**heap[child / 2] = temp**

**}**

**delete(heap, length){**

**for(i = length - 1 to 1){**

**swap(heap[1], heap[i + 1])**

**heapify(heap, heap[1], 1, i)**

**}**

**}**

**===============================================================**

===============================================================

You have two processors.

Maintain 2 queues.

For insertion, insert in the queue which has smallest time interval (sum of the time of all jobs).

For deletion, delete with option1 & option2, for processor 1 & 2, respectively.

Insertion:

Description:

-> This function, insert jobs from list to heap and use heapify function to make it min-heap.

-> Before inserting a job into heap, it compares the sum of total burst times associated with heaps and insert the job into heap with minimum sum of burst times.

-> Time complexity is, O(n + 2 \* logn) ~ O(n).

Input(s):

1. List of jobs

2. Total number of jobs

Output(s):

1. Min heaps of jobs

**Algorithm:**

**heapify(heap, temp, parent, length){**

**child = parent \* 2**

**while(child <= length){**

**if((child + 1 <= length) & (heap[child] > heap[child + 1])){**

**child = child + 1**

**}**

**if(temp > heap[child]){**

**heap[child / 2] = heap[child]**

**child = child \* 2**

**} else {**

**break**

**}**

**}**

**heap[child / 2] = temp**

**}**

**insert(list, length){**

**timeInterval1 = 0**

**timeInterval2 = 0**

**heapLength1 = length / 2**

**heapLength2 = length - heapLength1**

**index1 = 0**

**index2 = 0**

**for(i = 1 to length){**

**if(index1 <= heapLength1){**

**if((index2 <= heapLength2) & (timeInterval1 > timeInterval2)){**

**heap2[index2] = list[i]**

**index2 = index2 + 1**

**timeInterval2 = timeInterval2 + list[i]**

**} else {**

**heap1[index1] = list[i]**

**index1 = index1 + 1**

**timeInterval1 = timeInterval1 + list[i]**

**}**

**} else {**

**heap2[index2] = list[i]**

**index2 = index2 + 1**

**timeInterval2 = timeInterval2 + list[i]**

**}**

**}**

**for(i = heapLength1 / 2 to 1){**

**heapify(heap1, heap1[i], i, heapLength1)**

**}**

**for(i = heapLength2 / 2 to 1){**

**heapify(heap2, heap2[i], i, heapLength2)**

**}**

**}**

Deletion:

Description:

-> This function, delete minimum job from the heap associated with given processor.

-> Then swaps 1st & last job of heap and re-heapify it. This process is iterative for all the remaining jobs in that heap.

-> Time complexity is, O(logn).

Input(s):

1. Two min heaps of jobs

2. Total number of jobs in both the min heaps

Output(s):

1. Empty heaps/max heaps of jobs

Algorithm:

heapify(heap, temp, parent, length){

child = parent \* 2

while(child <= length){

if((child + 1 <= length) & (heap[child] > heap[child + 1])){

child = child + 1

}

if(temp > heap[child]){

heap[child / 2] = heap[child]

child = child \* 2

} else {

break

}

}

heap[child / 2] = temp

}

delete(heap1, heap2, heapLength1, heapLength2, processor){

if(processor1 == true & processor2 = true){

if(processor == 1 && heapLength1 > 0){

swap(heap1[1], heap1[heapLength1])

heapLength1 = heapLength1 - 1

heapify(heap1, heap1[1], 1, heapLength1)

} else if(processor == 2 && heapLength2 > 0){

swap(heap2[1], heap2[heapLength2])

heapLength2 = heapLength2 - 1

heapify(heap2, heap2[1], 1, heapLength2)

}

}

}

Suppose, one processor fails.

Job from that processor have to be transferred to the processor which is in working condition.

a) Delete the element that is minimum from both the queues.

Description:

-> This function, compare minimum jobs from both the heaps, delete the minimum among them and heapify remaining jobs. This process is iterative for each remaining jobs of heaps.

-> Time complexity is, O(logn).

Input(s):

1. Min heaps of jobs

2. Total number of jobs in both the min heaps

Output(s):

1. Empty heaps/max heaps of jobs

Algorithm:

heapify(heap, temp, parent, length){

child = parent \* 2

while(child <= length){

if((child + 1 <= length) & (heap[child] > heap[child + 1])){

child = child + 1

}

if(temp > heap[child]){

heap[child / 2] = heap[child]

child = child \* 2

} else {

break

}

}

heap[child / 2] = temp

}

delete(heap1, heap2, heapLength1, heapLength2){

while(heapLength1 > 0 || heapLength2 > 0){

if(heapLenght1 > 0) {

if((heapLength2 > 0) & (heap1[0] > heap2[0])){

swap(heap2[1], heap2[heapLength2])

heapLength2 = heapLength2 - 1

heapify(heap2, heap2[1], 1, heapLength2)

} else {

swap(heap1[1], heap1[heapLength1])

heapLength1 = heapLength1 - 1

heapify(heap1, heap1[1], 1, heapLength1)

}

} else {

swap(heap2[1], heap2[heapLength2])

heapLength2 = heapLength2 - 1

heapify(heap2, heap2[1], 1, heapLength2)

}

}

}

b) Merge elements of both the queues and process as usual.

Description:

-> This function, first delete all the elements from heap associated with working processor and then move to remaining heap.

-> Time complexity is, O(logn).

Input(s):

1. Min heaps of jobs

2. Total number of jobs in both the min heaps

3. State of processor1 (true if it's in working condition)

Output(s):

1. Empty heaps/max heaps of jobs

Algorithm:

heapify(heap, temp, parent, length){

child = parent \* 2

while(child <= length){

if((child + 1 <= length) & (heap[child] > heap[child + 1])){

child = child + 1

}

if(temp > heap[child]){

heap[child / 2] = heap[child]

child = child \* 2

} else {

break

}

}

heap[child / 2] = temp

}

delete(heap1, heap2, heapLength1, heapLength2, processor1){

if(processor1 == true){

while(heapLength1 > 0){

swap(heap1[1], heap1[heapLength1])

heapLength1 = heapLength1 - 1

heapify(heap1, heap1[1], 1, heapLength1)

}

while(heapLength2 > 0){

swap(heap2[1], heap2[heapLength2])

heapLength2 = heapLength2 - 1

heapify(heap2, heap2[1], 1, heapLength2)

}

} else {

while(heapLength2 > 0){

swap(heap2[1], heap2[heapLength2])

heapLength2 = heapLength2 - 1

heapify(heap2, heap2[1], 1, heapLength2)

}

while(heapLength1 > 0){

swap(heap1[1], heap1[heapLength1])

heapLength1 = heapLength1 - 1

heapify(heap1, heap1[1], 1, heapLength1)

}

}

}

c) Delete elements from queue of failed processor and insert them one-by-one to the working one and then proceed as usual.

Description:

-> In this function, all the jobs from both the heaps are copied in bigger heap and then that new heap is heapified.

-> Then the function swaps first and last jobs in that min-heap and re-heapified again. This process is iterative for each remaining jobs of min-heap.

-> Time complexity is, O(n + 2logn) ~ O(n).

Input(s):

1. Min heaps of jobs

2. Total number of jobs in both the min heaps

3. State of processor1 (true if it's in working condition)

Output(s):

1. Empty heap/max heap of all the jobs

Algorithm:

heapify(heap, temp, parent, length){

child = parent \* 2

while(child <= length){

if((child + 1 <= length) & (heap[child] > heap[child + 1])){

child = child + 1

}

if(temp > heap[child]){

heap[child / 2] = heap[child]

child = child \* 2

} else {

break

}

}

heap[child / 2] = temp

}

delete(heap1, heap2, heapLength1, heapLength2, processor1){

heapLength = heapLength1 + heapLength2

heap[heapLength]

if(processor1 == true){

for(i = 1 to heapLength1){

heap[i] = heap1[i]

}

for(i = heapLength1 to heapLength){

heap[i] = heap2[i - heapLength2]

}

} else {

for(i = 1 to heapLength2){

heap[i] = heap2[i]

}

for(i = heapLength2 to heapLength){

heap[i] = heap1[i - heapLength1]

}

}

for(i = heapLength / 2 to 1){

heapify(heap, heap[i], i, heapLength)

}

for(i = heapLength - 1 to 1){

swap(heap[1], heap[i + 1])

heapify(heap, heap[1], 1, i)

}

}