Universidad Tecnológica de Pereira

CAPÍTULO: COLAS DE PRIORIDAD (PRIORITY QUEUE)
CURSO DE ESTRUCTURA DE DATOS - Código IS304

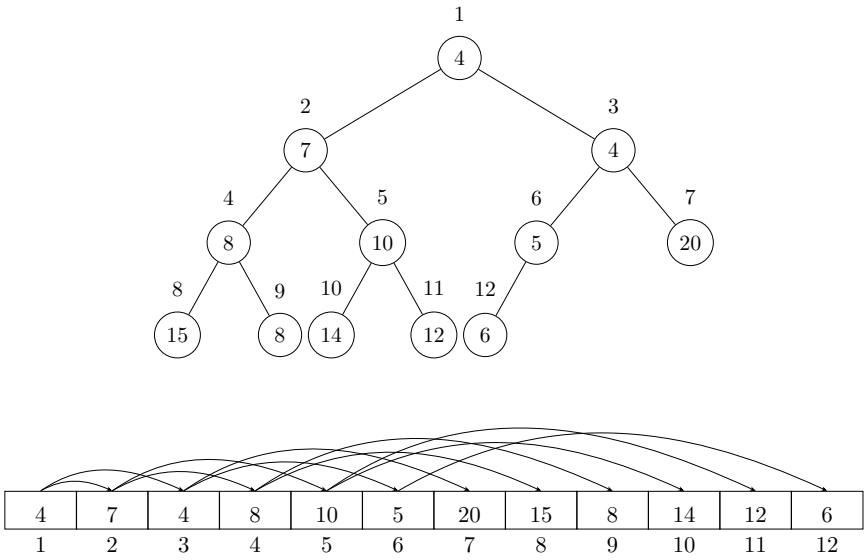
Programa de Ingeniería de Sistemas y Computación
Profesor Hugo Humberto Morales Peña
Semestre Agosto/Diciembre de 2021

Estructura de Datos Montón (Heap)

El algoritmo de *Ordenamiento por Montones - Heap Sort* creado por J. W. J. Williams en 1964[1], es un algoritmo excelente, pero el Quicksort es mejor en la práctica. Sin embargo, la estructura de datos montón (heap) tiene muchos usos. El uso más importante que se le da a la estructura de datos montón es para implementar de forma eficiente colas de prioridad (Priority Queue). A continuación se presentan las propiedades de forma y orden de los montones y la función que garantiza la propiedad de orden de montón mínimo.

Las Propiedades de Forma y Orden en un Montón

La estructura de datos montón es un arreglo de "objetos" muy parecido a un árbol binario completo. Cada nodo del árbol corresponde a un elemento del arreglo que almacena el valor en el nodo. **La propiedad de forma de un montón** indica que el árbol esta completamente lleno en todos los niveles excepto posiblemente en el nivel más bajo, el cual es llenado de izquierda a derecha hasta algún punto, la siguiente figura representa la situación anteriormente planteada.



[1]Williams, J. W. J.. 1964. "Algorithm 232 (HEAPSORT)". Communications of the ACM, 7:347-348.

Sobre un arreglo Q que representa un montón tenemos dos valores, dos medidas: $length[Q]$, el cual es el tamaño del arreglo Q, y $heapSize$, el cual es el número de elementos del montón almacenados en las primeras posiciones del arreglo Q, donde $heapSize \leq length[Q]$.

Hay dos clases de montones binarios: el *montón máximo* y el *montón mínimo*. En ambos casos, los valores de los nodos satisfacen una **propiedad de orden de un montón**, en el caso de un *montón máximo*, la propiedad de orden del montón máximo es que para todo nodo i diferente de la raíz se cumple que $Q[Parent(i)] \geq Q[i]$, esto es que el valor de cada nodo como máximo es el valor de su padre, de esta forma el elemento más grande de un montón máximo está almacenado en la raíz. En el caso de un *montón mínimo*, la propiedad de orden del montón mínimo es que para todo nodo i diferente de la raíz se cumple que $Q[Parent(i)] \leq Q[i]$, esto es que el valor de cada nodo como mínimo es el valor de su padre, de esta forma el elemento más pequeño de un montón mínimo está almacenado en la raíz.

El contenido de la raíz del árbol esta almacenado en $Q[1]$, y dado el índice de un nodo, el índice de su padre `Parent`(i), el índice del hijo por la izquierda `Left`(i) y el índice del hijo por la derecha `Right`(i) pueden ser calculados simplemente con las siguientes funciones:

function Parent(i)
1. **return** $\lfloor \frac{i}{2} \rfloor$

function Left(i)
1. **return** $2 * i$

function Right(i)
1. **return** $2 * i + 1$

Al mirar un montón como un árbol, se define la *altura* de un nodo en el montón como la cantidad de aristas que hay desde el nodo a alguna de las hojas más lejanas que se alcanzan desde él en una ruta simple descendente, de esta forma la altura del montón es la altura del nodo raíz del árbol. La altura de un montón es $\theta(lg\ n)$.

Función para Garantizar la Propiedad de Orden de Montón

Los parámetros de entrada de la función `MinHeapify` son un arreglo Q y un subíndice i sobre el arreglo. La función `MinHeapify` asume que tanto el subárbol izquierdo como el subárbol derecho a partir de la posición i son montones mínimos, pero que $Q[i]$ puede ser más grande que alguno de sus dos hijos, con lo que se violaría la propiedad de montón mínimo. La función `MinHeapify` le permite al valor almacenado en $Q[i]$ "sumergirse" en el montón mínimo hasta lograr que el subárbol con raíz i sea un montón mínimo.

function MinHeapify(Q, i)
1. $l =$ **Left**(i)
2. $r =$ **Right**(i)
3. **if** $l \leq heapSize$ **and** $Q[l] < Q[i]$
4. $\quad least = l$
5. **else** $least = i$
6. **if** $r \leq heapSize$ **and** $Q[r] < Q[least]$
7. $\quad least = r$
8. **if** $least \neq i$
9. $\quad$ **exchange** $Q[i]$ **with** $Q[least]$
10. $\quad$ **MinHeapify**$(Q, least)$

Si $Q[i]$ es menor o igual que la información almacenada en la raíz de los subárboles izquierdo y derecho entonces

el árbol con raíz el nodo i es un montón mínimo y la función termina. De lo contrario, la raíz de alguno de los subárboles tiene información menor que la que se encuentra en $Q[i]$ y es intercambiada con ésta, con lo cual se garantiza que el nodo i y sus hijos cumplen la propiedad de montón mínimo, pero, sin embargo el subárbol hijo con el cual se intercambio la información de $Q[i]$ ahora puede no cumplir la propiedad de montón mínimo, por lo tanto, se debe llamar de forma recursiva a la función `MinHeapify` sobre el subárbol hijo con el cual se hizo el intercambio.

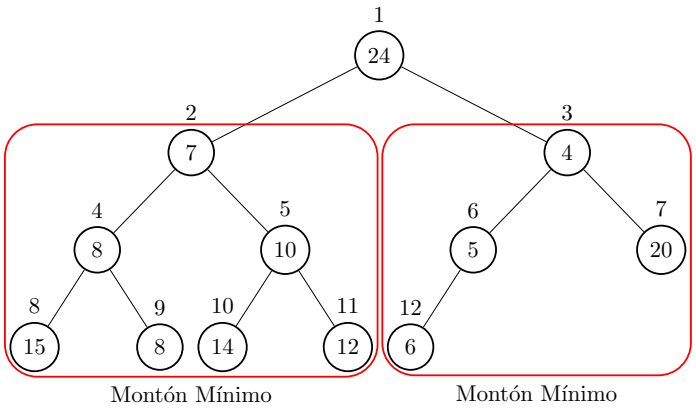
La complejidad en el peor de los casos de la función `MinHeapify` es $O(h)$, donde h es la altura del montón, como la altura del montón es $\theta(\log n)$ entonces `MinHeapify` es $O(\log n)$.

Ejemplo

Realizar el paso a paso del algoritmo `MinHeapify`$(Q,\ 1)$ cuando el arreglo Q tiene los elementos:

$Q[i]$	24	7	4	8	10	5	20	15	8	14	12	6
i	1	2	3	4	5	6	7	8	9	10	11	12

y tiene su equivalencia gráfica por medio de "árboles binarios" como:



Paso a paso del algoritmo MinHeapify

`MinHeapify`$(\ Q,\ 1\ )$

i	l	r	$heapSize$	$least$
1	2	3	12	2
				3

$Q[i]$	**24**	7	**4**	8	10	5	20	15	8	14	12	6
i	**1**	2	**3**	4	5	6	7	8	9	10	11	12

$Q[i]$	**4**	7	**24**	8	10	5	20	15	8	14	12	6
i	**1**	2	**3**	4	5	6	7	8	9	10	11	12

`MinHeapify`$(\ Q,\ 3\ )$

i	l	r	$heapSize$	$least$
3	6	7	12	6

$Q[i]$	4	7	**24**	8	10	**5**	20	15	8	14	12	6
i	1	2	**3**	4	5	**6**	7	8	9	10	11	12

$Q[i]$	4	7	**5**	8	10	**24**	20	15	8	14	12	6
i	1	2	**3**	4	5	**6**	7	8	9	10	11	12

`MinHeapify`(Q, 6)

i	l	r	$heapSize$	$least$
6	12	13	12	12

$Q[i]$	4	7	5	8	10	**24**	20	15	8	14	12	**6**
i	1	2	3	4	5	**6**	7	8	9	10	11	**12**

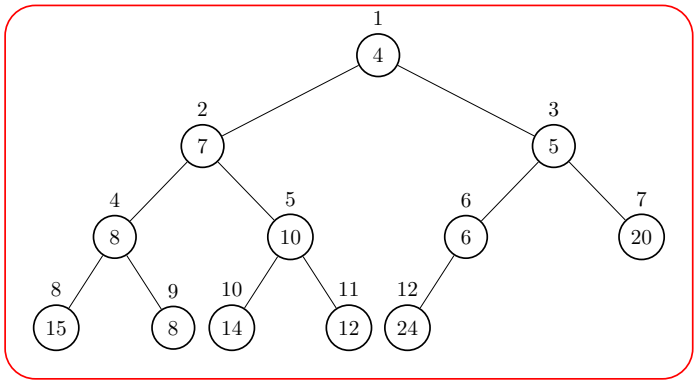
$Q[i]$	4	7	5	8	10	**6**	20	15	8	14	12	**24**
i	1	2	3	4	5	**6**	7	8	9	10	11	**12**

`MinHeapify`(Q, 12)

i	l	r	$heapSize$	$least$
12	24	25	12	12

$Q[i]$	4	7	5	8	10	6	20	15	8	14	12	24
i	1	2	3	4	5	6	7	8	9	10	11	12

Al finalizar el paso a paso del algoritmo `MinHeapify` obtenemos un montón mínimo desde la posición 1 el cual tiene su equivalencia gráfica por medio de "árboles binarios" como:



Montón Mínimo

Colas de Prioridad (Priority Queue)

Como en los montones, las colas de prioridad son de dos tipos: colas de prioridad máxima y colas de prioridad mínima. En este curso trabajaremos las colas de prioridad mínima.

En una cola de prioridad mínima el elemento más pequeño es el que se atiende primero.

Una cola de prioridad mínima soporta las siguientes operaciones:

- **MinPQ_Insert**(Q, key): Inserta el elemento key en la cola de prioridad mínima Q.
- **MinPQ_Minimum**(Q): Retorna el elemento más pequeño de la cola de prioridad mínima Q.
- **MinPQ_Extract**(Q): Remueve y retorna el elemento más pequeño de la cola de prioridad mínima Q.
- **MinPQ_DecreaseKey**(Q, i, key): Disminuye el valor del elemento ubicado en la posición i de la cola de prioridad Q al nuevo valor key. Se asume que el valor de key como máximo es el valor del elemento ubicado en la posición i de Q.

Las funciones que soportan las operaciones del manejo de colas de prioridad mínima son:

function MinPQ_Minimum(Q)

1. **return** $Q[1]$

function MinPQ_Extract(Q)

1. **if** $heapSize < 1$
2. **error** "heap underflow"
3. **else**
4. $min = Q[1]$
5. $Q[1] = Q[heapSize]$
6. $heapSize = heapSize - 1$
7. **MinHeapify**(Q, 1)
8. **return** min

function MinPQ_DecreaseKey(Q, i, key)

1. **if** $key > Q[i]$
2. **error** "new key is higher than current key"
3. **else**
4. $Q[i] = key$
5. **while** $i > 1$ **and** $Q[$**Parent**$(i)] > Q[i]$
6. **exchange** $Q[i]$ **with** $Q[$**Parent**$(i)]$
7. $i =$ **Parent**(i)

function MinPQ_Insert(Q, key)

1. $heapSize = heapSize + 1$
2. $Q[heapSize] = \infty$
3. **MinPQ_DecreaseKey**(Q, $heapSize$, key)

La función **MinPQ_Minimum** tiene una complejidad en tiempo de ejecución de $O(1)$.
Las funciones **MinPQ_Extract**, **MinPQ_DecreaseKey** y **MinPQ_Insert** tienen complejidad en tiempo de ejecución de $O(\log n)$.

Implementación de la Cola de Prioridad Mínima (Minimum Priority Queue)

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myNegativeInfinite -2147483647
#define myPositiveInfinite 2147483647
#define MAXT 1000

int Parent(int i)
{
    return i >> 1;
/*  return i / 2; */
}

int Left(int i)
{
    return i << 1;
/*  return 2 * i; */
}
```

```
int Right(int i)
{
    return (i << 1) + 1;
/*  return 2 * i + 1; */
}

void MinHeapify(int Q[], int i, int heapSize)
{
    int l, r, least, temp;
    l = Left(i);
    r = Right(i);

    if((l <= heapSize) && (Q[l] < Q[i]))
        least = l;
    else
        least = i;

    if((r <= heapSize) && (Q[r] < Q[least]))
        least = r;

    if(least != i)
    {
        temp = Q[i];
        Q[i] = Q[least];
        Q[least] = temp;
        MinHeapify(Q, least, heapSize);
    }
}

int MinPQ_Minimum(int Q[])
{
    return Q[1];
}

int MinPQ_Extract(int Q[], int *heapSize)
{
    int min = myNegativeInfinite;

    if(*heapSize < 1)
        printf("Heap underflow.\n");
    else
    {
        min = Q[1];
        Q[1] = Q[*heapSize];
        (*heapSize)--;
        MinHeapify(Q, 1, *heapSize);
    }

    return min;
}

void MinPQ_DecreaseKey(int Q[], int i, int key)
{
    int temp;

    if(key > Q[i])
        printf("New key is higher than current key.\n");
    else
    {
        Q[i] = key;
        while((i > 1) && (Q[Parent(i)] > Q[i]))
        {
            temp = Q[i];
            Q[i] = Q[Parent(i)];
            Q[Parent(i)] = temp;
            i = Parent(i);
        }
```

```
    }
}

void MinPQ_Insert(int Q[], int key, int *heapSize)
{
    (*heapSize)++;
    Q[*heapSize] = myPositiveInfinite;
    MinPQ_DecreaseKey(Q, *heapSize, key);
}


int main()
{
    int operation, element, Q[MAXT + 2], heapSize = 0;

    while(scanf("%d", &operation) != EOF)
    {
        if(operation == 1) /* Insert into Priority Queue */
        {
            scanf("%d", &element);
            MinPQ_Insert(Q, element, &heapSize);
        }
        else
        {
            if(operation == 2) /* Extract min element of the Priority Queue */
            {
                element = MinPQ_Extract(Q, &heapSize);
                if(element == myNegativeInfinite)
                    printf("The Min Priority Queue is empty.\n");
                else
                    printf("%d\n", element);
            }
            else
                printf("Bad use.\n 1. Insert into PQ \n 2. Extract of the PQ.\n");
        }
    }

    return 0;
}
```

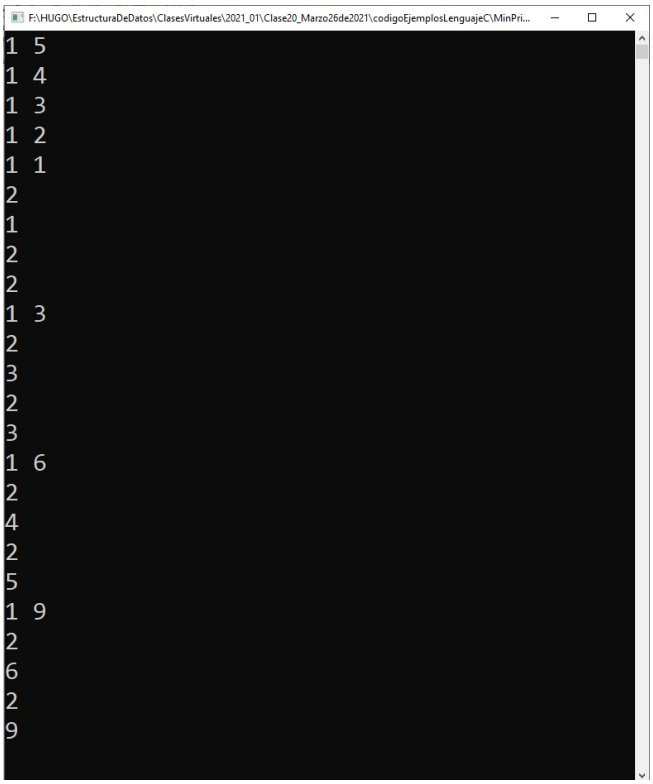


Figura 1: Salida del programa del manejo de la Cola de Prioridad Mínima.

Programming Challenge: "Add All - Sumar Todos"[2]

Si!, el nombre del problema refleja el trabajo a realizar; es justamente sumar un conjunto de números. Usted puede sentirse motivado a escribir un programa en su lenguaje de programación favorito para sumar todo el conjunto de números. Tal problema debe ser algo muy sencillo para su nivel de programación. Por este motivo le vamos a poner un poco de sabor y vamos a volver el problema mucho más interesante.

Ahora, la operación de suma tiene un costo, y el costo es el resultado de sumar dos números. Por lo tanto, el costo de sumar 1 y 10 es 11. Por ejemplo, si usted quiere sumar los números 1, 2 y 3, hay tres formas de hacerlo:

- Forma 1:
  $1 + 2 = 3$, costo $= 3$
  $3 + 3 = 6$, costo $= 6$
  Costo total $= 9$
- Forma 2:
  $1 + 3 = 4$, costo $= 4$
  $2 + 4 = 6$, costo $= 6$
  Costo total $= 10$
- Forma 3:
  $2 + 3 = 5$, costo $= 5$
  $1 + 5 = 6$, costo $= 6$
  Costo total $= 11$

Yo creo que usted ya ha entendido su misión, sumar un conjunto de números enteros tal que el costo de la operación suma sea mínimo.

[2] https://www.hackerrank.com/contests/utp-open-2018/challenges/add-all-2-1

Input specification:

La entrada contiene múltiples casos de prueba. Cada caso de prueba debe comenzar con un número entero positivo N ($2 \leq N \leq 5000$), seguido por N números enteros positivos (todos son mayores o iguales a 1 y menores o iguales a 100000). La entrada es finalizada por un caso donde el valor de N es igual a cero. Este caso no debe ser procesado.

Output specification:

Para cada caso de prueba imprimir una sola línea con un número entero positivo que representa el costo total mínimo de la operación suma después de sumar los N números.

Example input:

```
3
1 2 3
4
1 2 3 4
5
5 4 3 2 1
0
```

Example output:

```
9
19
33
```

Solución del reto "Add All - Sumar Todos"

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myNegativeInfinite -2147483647
#define myPositiveInfinite 2147483647
#define MAXT 5000

int Parent(int i)
{
    return i >> 1; /* return i / 2; */
}

int Left(int i)
{
    return i << 1; /* return 2 * i; */
}

int Right(int i)
{
    return (i << 1) + 1; /* return 2 * i + 1; */
}

void MinHeapify(int Q[], int i, int heapSize)
{
    int l, r, least, temp;
    l = Left(i);
    r = Right(i);

    if((l <= heapSize) && (Q[l] < Q[i]))
        least = l;
    else
        least = i;
```

```
    if((r <= heapSize) && (Q[r] < Q[least]))
        least = r;

    if(least != i)
    {
        temp = Q[i];
        Q[i] = Q[least];
        Q[least] = temp;
        MinHeapify(Q, least, heapSize);
    }
}

int MinPQ_Minimum(int Q[])
{
    return Q[1];
}

int MinPQ_Extract(int Q[], int *heapSize)
{
    int min = myNegativeInfinite;

    if(*heapSize < 1)
        printf("Heap underflow.\n");
    else
    {
        min = Q[1];
        Q[1] = Q[*heapSize];
        (*heapSize)--;
        MinHeapify(Q, 1, *heapSize);
    }

    return min;
}

void MinPQ_DecreaseKey(int Q[], int i, int key)
{
    int temp;

    if(key > Q[i])
        printf("New key is higher than current key.\n");
    else
    {
        Q[i] = key;
        while((i > 1) && (Q[Parent(i)] > Q[i]))
        {
            temp = Q[i];
            Q[i] = Q[Parent(i)];
            Q[Parent(i)] = temp;
            i = Parent(i);
        }
    }
}

void MinPQ_Insert(int Q[], int key, int *heapSize)
{
    (*heapSize)++;
    Q[*heapSize] = myPositiveInfinite;
    MinPQ_DecreaseKey(Q, *heapSize, key);
}

int main()
{
    int n, element, Q[MAXT + 2];
    int index, heapSize;
    long long int result;

    while(scanf("%d", &n) && (n > 0))
```

```
    {
        heapSize = 0;

        for(index = 1; index <= n; index++)
        {
            scanf("%d", &element);
            MinPQ_Insert(Q, element, &heapSize);
        }

        result = 0;

        for(index = 1; index < n; index++)
        {
            element = MinPQ_Extract(Q, &heapSize);
            element += MinPQ_Extract(Q, &heapSize);
            result += element;
            MinPQ_Insert(Q, element, &heapSize);
        }

        printf("%lld\n", result);
    }

    return 0;
}
```

Figura 2: Salida del programa para el reto “Add All”.