# Segundo Parcial de Programación 2

## 6 de julio de 2018

**Pregunta sobre el Laboratorio**

Nombre la estructura de datos con la que se debía implementar el TAD Conjunto para poder cumplir las restricciones de tiempo pedidas.

**Solución:**
Con un árbol binario de búsqueda.

**Problema 1 (10 puntos)**

(a) Describa las dos propiedades que debe cumplir un árbol binario para ser un heap (montículo binario).

(b) Dibuje el heap resultante de insertar los números 6, 7, 8, 9, 1, 2, 3, 4, 5, 14, 12 en ese orden y a partir de un heap vacío.

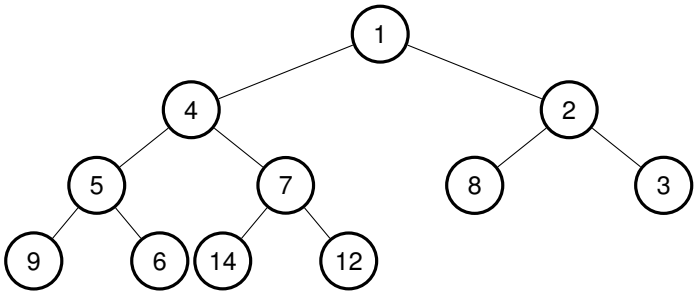
(c) Dibuje el heap resultante de eliminar el mínimo elemento 2 veces a partir del heap resultante de la parte (b).

**Solución:**

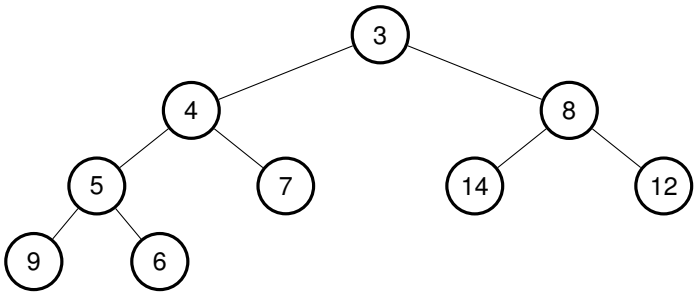
(a) Para ser considerado un heap, un árbol binario debe cumplir:

- **Propiedad de la estructura**: Todos los niveles están completos, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha.
- **Propiedad de orden**: La clave de cada nodo es prioritaria respecto a las de sus descendientes.

(b)




(c)




**Problema 2 (20 puntos)**

Considere la siguiente definición del tipo FS de árboles generales con la semántica primer hijo (pH)–siguiente

hermano (sH) que representa un *file system*, FS, que es una estructura de directorios de un sistema operativo en el que no se tienen en cuenta los archivos. Cada nodo almacena el *nombre* del directorio y la *fecha* en que fue creado.

```
struct nodoFS {
  char * nombre;
  Fecha fecha;
  nodoFS *pH, *sH;
}
typedef nodoFS * FS;
```

Implemente el procedimiento *masCercano* que, dados un árbol `t` de tipo `FS`, recorriendo por niveles `t` retorne la fecha de creación del directorio de nombre `nomDir` que esté en el nivel menos profundo. Si en ese nivel hay más de un directorio con ese nombre retorna la fecha de creación de cualquiera de ellos. Si `nomDir` no se encuentra en `t`, la función deberá retornar `NULL`. La función no debe recorrer nodos y niveles del árbol que no sean estrictamente necesarios y no debe visitar a un nodo múltiples veces.

```
Fecha masCercano(FS t, char* nomDir);
```

Asuma que dispone de la función booleana `sonIguales` para comparar strings.

Si usa un TAD auxiliar de los vistos en el curso, deberá especificarlo pero no implementarlo.

**Solución:**

```
Fecha masCercano(FS t, char *nomDir) {
  Fecha resultado = NULL;
  if (t != NULL) {
    Cola cola = crearCola();
    encolar(cola, t);
    while (!esVacia(cola)) {
      FS carpeta = frente(cola);
      desencolar(cola);
      if (sonIguales(carpeta->nombre, nomDir)) {
        resultado = carpeta->fecha;
        while (!esVacia(cola))
          desencolar(cola);
      } else {
        FS subcarpeta = carpeta->pH;
        while (subcarpeta != NULL) {
          encolar(cola, subcarpeta);
          subcarpeta = subcarpeta->sH;
        }
      }
    }
    destruirCola(cola);
  }
  return resultado;
}
```

Se basa en una Cola no acotada de elementos de tipo `FS`, cuya especificación es:

```
// Crea una cola no acotada de elementos de tipo FS, vacía.
Cola crearCola();

// Encola un elemento de tipo FS.
void encolar(Cola & cola, FS t);

// Desencola el elemento más antiguo de 'cola'. Precondición: !esVacia(cola)
void desencolar(Cola & cola);

// Retorna el elemento más antiguo de 'cola'. Precondición: !esVacia(cola)
FS frente(Cola cola);

// Retorna 'true' si y sólo si la cola está vacía.
bool esVacia(Cola cola);
```

```
// Destruye la memoria asociada a la cola pero no la memoria asociada a sus elementos.
void destruirCola(Cola & cola);
```

**Problema 3 (25 puntos)**

(a) Especifique el TAD Cola de Prioridad no acotada de enteros. Además de las operaciones estándar debe incluir una operación que devuelve la cantidad de elementos de la cola. La prioridad de cada elemento está determinada por su valor y el elemento prioritario es el que tiene menor valor. Debe incluir precondiciones y poscondiciones.

(b) Implemente el TAD anterior, de tal manera que todas las operaciones, excepto la operación para eliminar, tengan tiempo de ejecución $O(1)$ en el peor caso.

(c) ¿Qué cambiaría en la especificación de la parte (a) si la cola de prioridad fuera acotada?

**Solución:**

(a)
```
/* Crea una cola de prioridad no acotada de enteros, vacía,
   donde los números de menor valor son los más prioritarios. */
ColaPrio crearColaPrio();

// Encola un entero en la cola de prioridad.
void encolar(ColaPrio & cp, int valor);

// Devuelve el elemento prioritario. Precondición: cantElems(cp) > 0
int min(ColaPrio cp);

// Desencola el elemento prioritario. Precondición: cantElems(cp) > 0
void desencolar(ColaPrio & cp);

// Devuelve la cantidad de elementos.
int cantElems(ColaPrio cp);

/* Destruye la memoria asociada a la cola de prioridad, pero no la
   asociada a sus elementos. */
void destruirColaPrio(ColaPrio & cp);
```

La presencia de la operación que devuelve la cantidad de elementos hace opcional la operación que indica si la cola de prioridad está vacía.

(b)
```
struct nodo {
  int elem;
  nodo * sig;
};
struct cabezal {
  int cant;
  int min;
  nodo * primero;
};
typedef cabezal * ColaPrio;

ColaPrio crearColaPrioridad() {
  ColaPrio cp = new cabezal;
  cp->cant = 0;
  cp->primero = NULL;
  return cp;
}

void encolar(ColaPrio & cp, int valor) {
  nodo * n = new nodo;
  n->elem = valor;
  n->sig = cp->primero;
  cp->primero = n;
  if (cp->cant == 0 || cp->min > valor)
```

```
    cp->min = valor;
  cp->cant++;
}

int min(ColaPrio cp) {
  return cp->min;
}

void desencolar(ColaPrio & cp) {
  if (cp->primero->elem == cp->min) {
    nodo * aux = cp->primero->sig;
    delete cp->primero;
    cp->primero = aux;
  } else {
    nodo * n = cp->primero;
    while (n->sig->elem != cp->min)
      n = n->sig;
    // Notar que no se precisa verificar n->sig != NULL en la condición del
    // while porque se sabe que tiene que haber un elemento de valor cp->min.

    nodo * aux = n->sig->sig;
    delete n->sig;
    n->sig = aux;

  }

  cp->cant--;

  // Actualizo el mínimo.
  if (cp->cant > 0) {
    cp->min = cp->primero->elem;
    nodo * n = cp->primero->sig;
    while (n != NULL) {
      if (cp->min > n->elem)
        cp->min = n->elem;

      n = n->sig;
    }
  }
}

int cantElems(ColaPrio & cp) {
  return cp->cant;
}

void destruirColaPrio(ColaPrio & cp) {
  nodo * n = cp->primero;
  while (n != NULL) {
    nodo * aux = n;
    n = n->sig;
    delete aux;
  }
  delete cp;
}
```

(c) El constructor debería recibir un parámetro que indique el tamaño máximo. A su vez, la operación para encolar debería tener una política para cuando se llene la cola de prioridad, por ejemplo borrar el elemento menos prioritario o que sea una precondición que no esté llena la cola de prioridad. Finalmente, se debe proveer una operación para saber si la cola de prioridad está llena, aunque también puede hacerse con una función que dé la capacidad máxima y comparar su valor con el resultado de `cantElems` para saber si la cola de prioridad está llena (en caso que sean iguales).