Montículos y filas de prioridad

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Noviembre 2009

Outline

Previously en la clase

- Isomorfismo entre arboles ordenados y arboles binarios.
- Arboles binarios representados por estructuras ligadas o arreglos.
- Noción de árbol "equilibrado" importante para lograr eficiencia de algoritmos, para arboles de cualquier tipo.

Outline

Fila de prioridad

- push()
- pop()
- top()

Fila de prioridad

- push()
- pop()
- top()

Fila de prioridad

- push()
- pop()
- top()

Fila de prioridad

- push()
- pop()
- top()

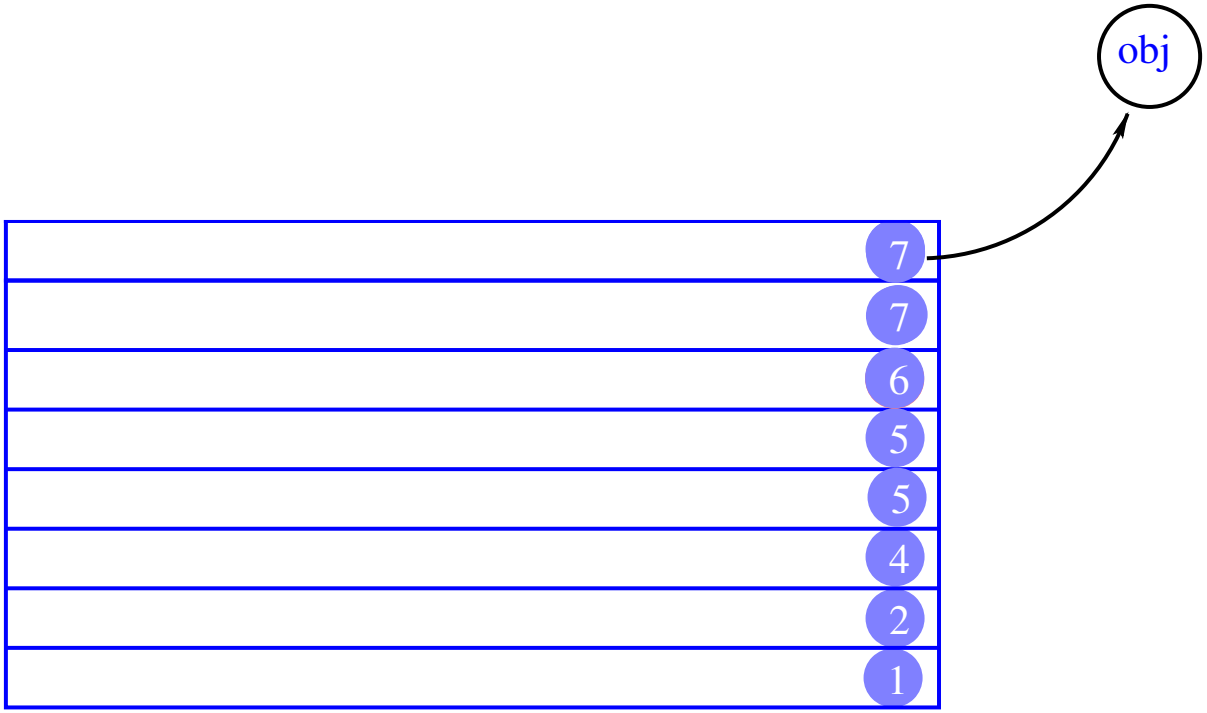
Fila de prioridad

- push()
- pop()
- top()

Fila de prioridad

- push()
- pop()
- top()

Fila de prioridad



- push()
- pop()
- top()

Fila de prioridad

- Objeto asociado con una llave.
- Comportamiento similar a una pila, pero con los datos "ordenados" por su valor.
- Soporta operaciones $\mathrm{push}()$, $\mathrm{pop}()$, $\mathrm{top}()$.
- El $\mathrm{pop}()$ sale el objeto "más prioritario" de la estructura.
- El $\mathrm{top}()$ da una referencia hacia éste.
- ¿Qué tipo de estructura usar?

Fila de prioridad

Solución 1: usar una secuencia (arreglo, lista ligada) en que todos los valores están ordenados por orden de la llave!

- $\mathrm{pop}()$: en $O(1)$ (el elemento máximo es a la derecha).
- $\mathrm{top}()$: en $O(1)$ (igual).
- $\mathrm{push}()$: en promedio, será $O(N)$, para buscar el lugar del nuevo elemento y mover todos los que están superiores hacia la derecha!

Fila de prioridad

Solución 2: usar una secuencia, y en cada operación de $\mathrm{pop}()$ o $\mathrm{top}()$ buscar el máximo (la solución "floja")

- $\mathrm{pop}()$: en $O(N)$ (hay que buscar el elemento mas grande).
- $\mathrm{top}()$: en $O(N)$ (igual).
- $\mathrm{push}()$: en $O(1)$.

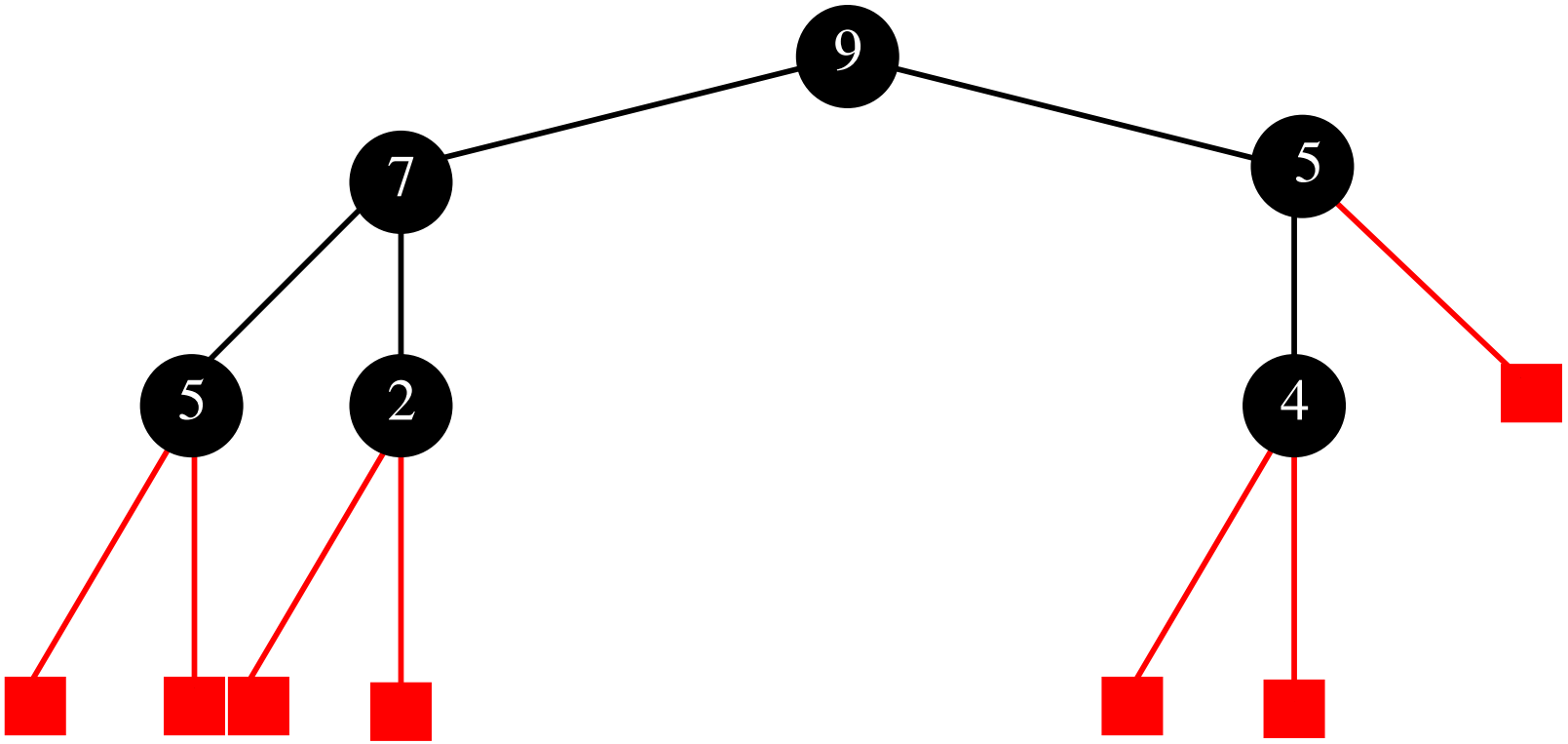
Montículo

Un montículo es una estructura de árbol binario:

- completo,
- tal que todos los elementos del ultimo nivel se encuentren en la parte izquierda,
- ordenado en montículo.

Ordenamiento en montículo: para cualquier par de nodo (ν, ν'), si ν' es un hijo de ν entonces $key(\nu') \leq key(\nu)$.

Montículo



Montículo

- Por construcción el objeto con llave mas alta esta en la raíz del árbol binario.
- Se puede establecer las definiciones y propiedades para montículos-max o montículos-min.
- Recordamos que:

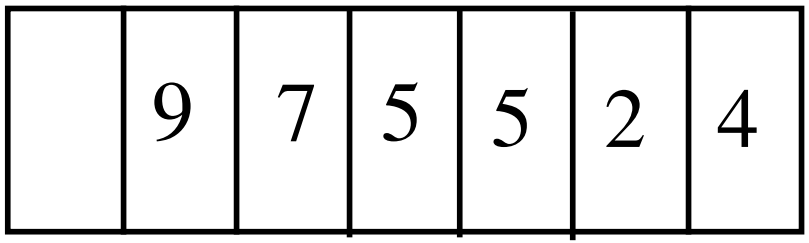
$$h = \lfloor \log_2 N \rfloor + 1$$

donde N es el numero de nodos internos.

Montículo

Ya que es árbol binario completo, es mas interesante usar un arreglo para almacenar los datos!

- Hijos de $a[k]$ en $a[2k]$ y $a[2k+1]$.
- Padre en de $a[k]$ en $a[k/2]$.
- No espacio desperdiciado: no hoyo, datos contiguos.



Montículo

Estructura:

```
template <class T, class K=int>
class heap {
public:
  // Constructor
  heap() {data.push_back(std::pair<T,K>(T(),K()));};
  // Add an object o with priority p
  void add(const T& o, const K& p);
  // Remove object with maximum key
  T removeMax();
  // Get reference to object with maximum key
  const T &getMax();
private:
  std::deque<std::pair<T,K> > data;
};
```

Montículo

El arreglo (montículo-max) usado satisface:

$$\begin{cases} 2 \leq 2i \leq N & \Rightarrow \quad a[i] \geq a[2i] \\ 3 \leq 2i+1 \leq N & \Rightarrow \quad a[i] \geq a[2i+1] \end{cases}$$

- Orden parcial en el árbol, según caminos de la raíz a las hojas.
- La meta es arreglarse para que estas desigualdades permanezcan mientras el árbol está modificado.

Montículo

El arreglo (montículo-max) tiene que permitir:

- Investigar el elemento de llave máxima contenido en el montículo.
- Añadir un elemento nuevo, un par (objeto,llave).
- Quitar el elemento de llave máxima de la estructura.

Montículo

Acceder al máximo en $O(1)$
Problema 1: al añadir un nuevo elemento en final del arreglo, en $a[N+1]$, ya no tenemos necesariamente $a[N+1] \leq a[\frac{N+1}{2}]$, la propiedad de montículo es violada!
Problema 2: mismo tipo de problema al quitar el elemento de llave máxima: ¿qué pongo al lugar de la raíz?

Montículo: recuperar el máximo

Operación muy sencilla:

```
const T &getMax() {
    if (data.size()<2)
      throw std::out_of_range("Empty heap");
    return data[1].first;
  };
```

en $O(1)$...

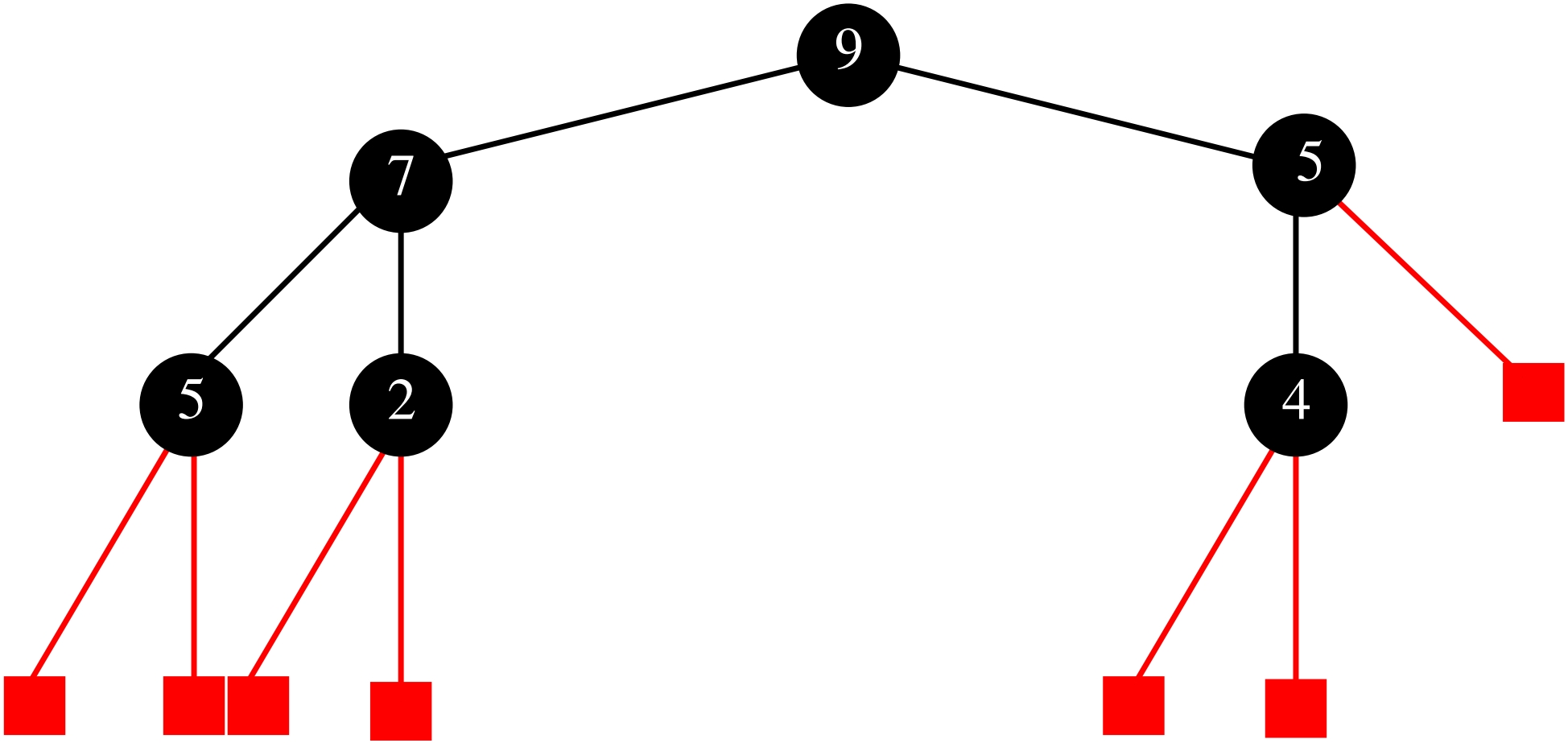
Montículo: añadir elemento

Idea básica al introducir un nuevo elemento ν por add:

- Poner el nuevo elemento ν en el primer nodo externo disponible (al final del arreglo).
- Comparar la llave con la de su padre, si está superior a la de su padre, intercambiar con el padre.
- El sub-arbol obtenido respeta el orden de montículo: los nuevos hijos sí tienen llaves inferiores a la del nuevo padre.
- Iterar esta operación de intercambio hasta que se llegue a la raíz o que se llegue a un nodo ν_t tal que $key(\nu) \leq key(\nu_t)$.

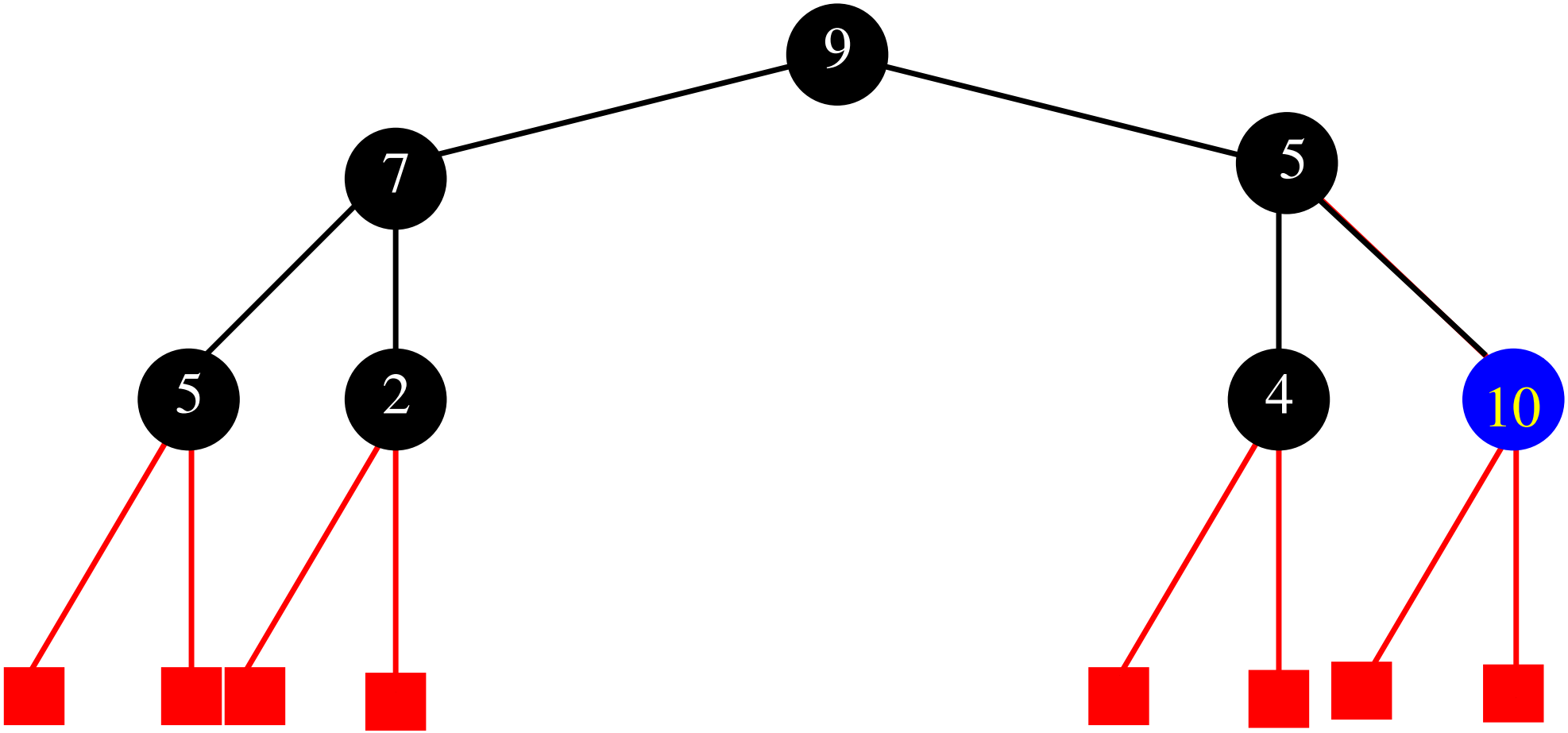
Montículo: añadir elemento

Añadir $\nu = (o, 10)$



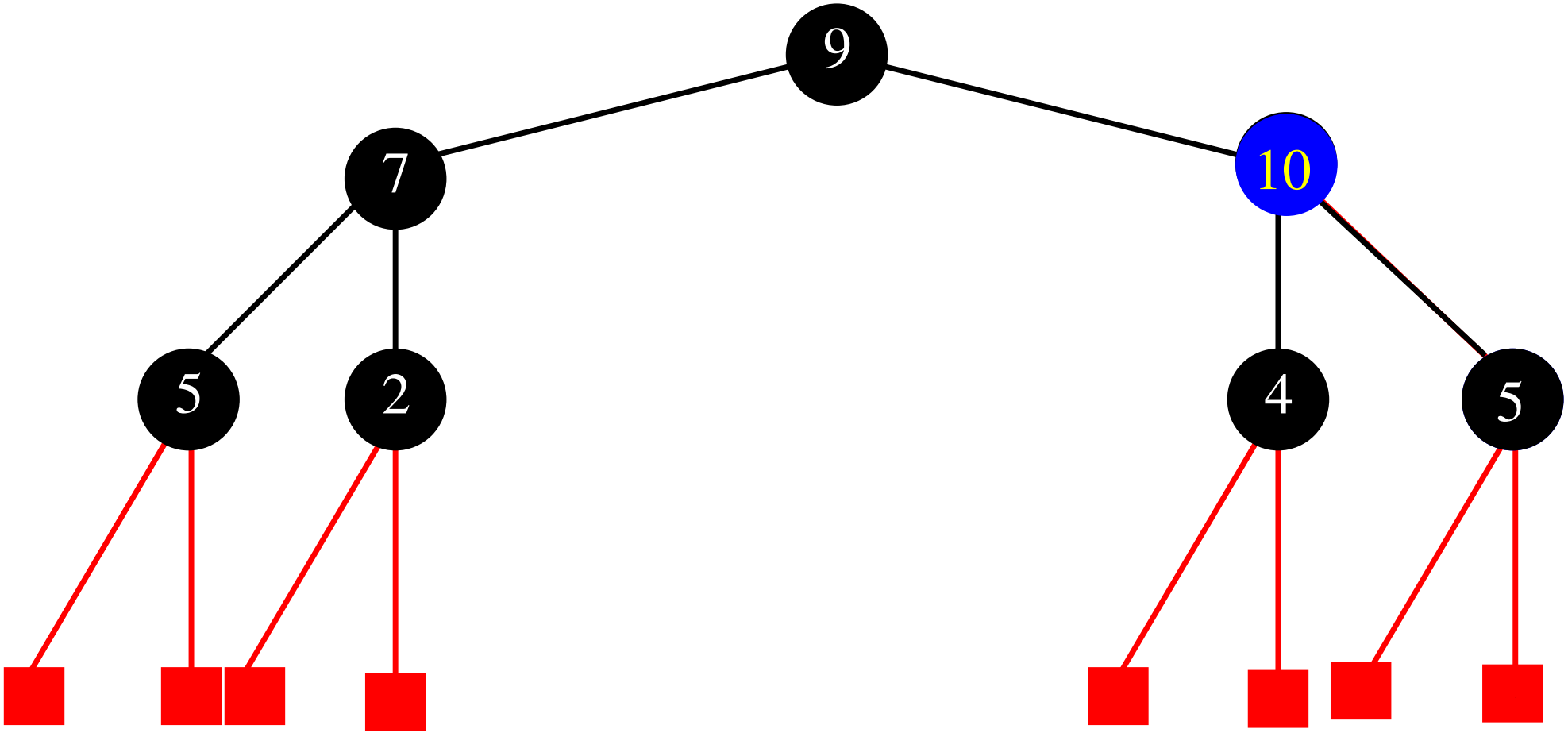
Montículo: añadir elemento

Añadir $\nu = (o, 10)$



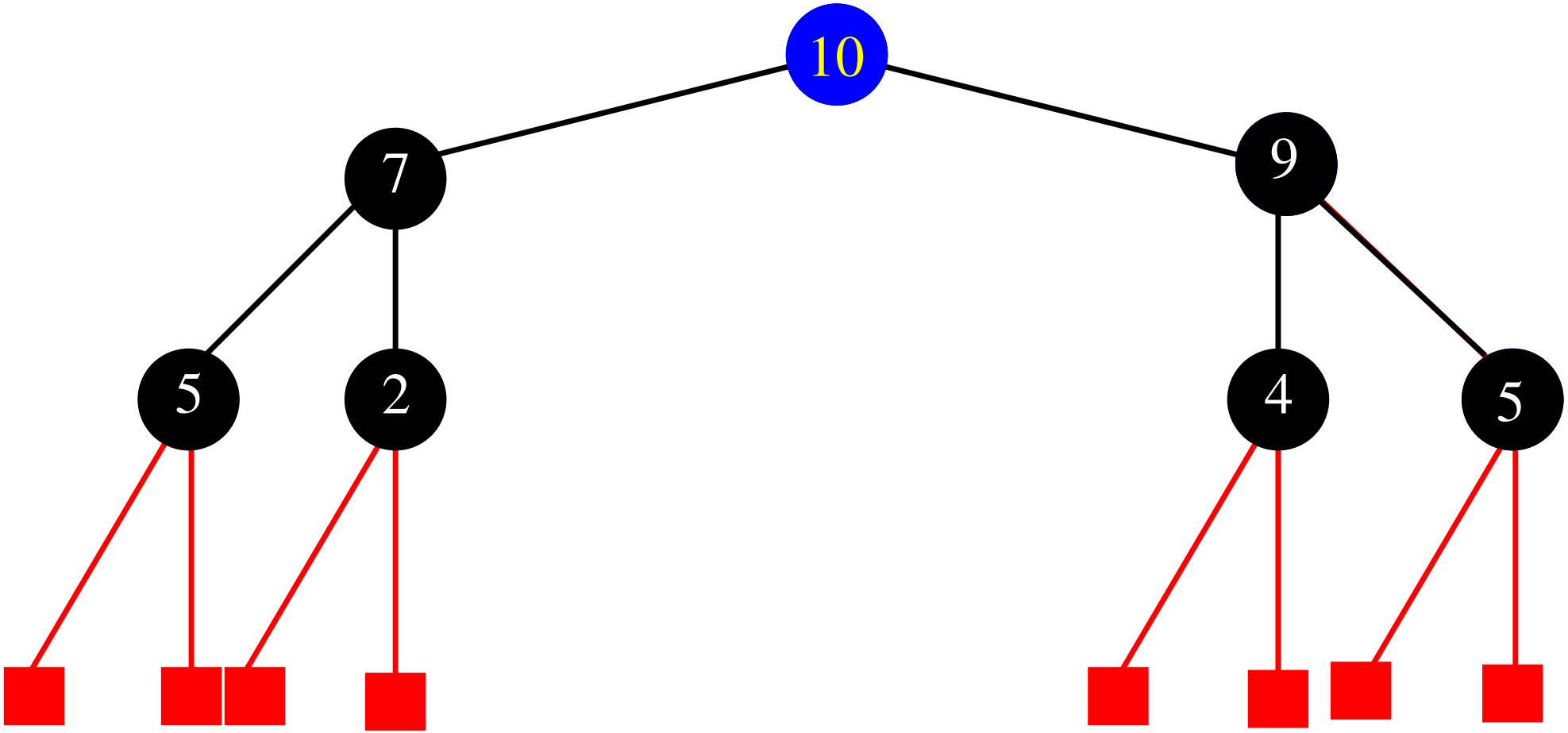
Montículo: añadir elemento

Añadir $\nu = (o, 10)$



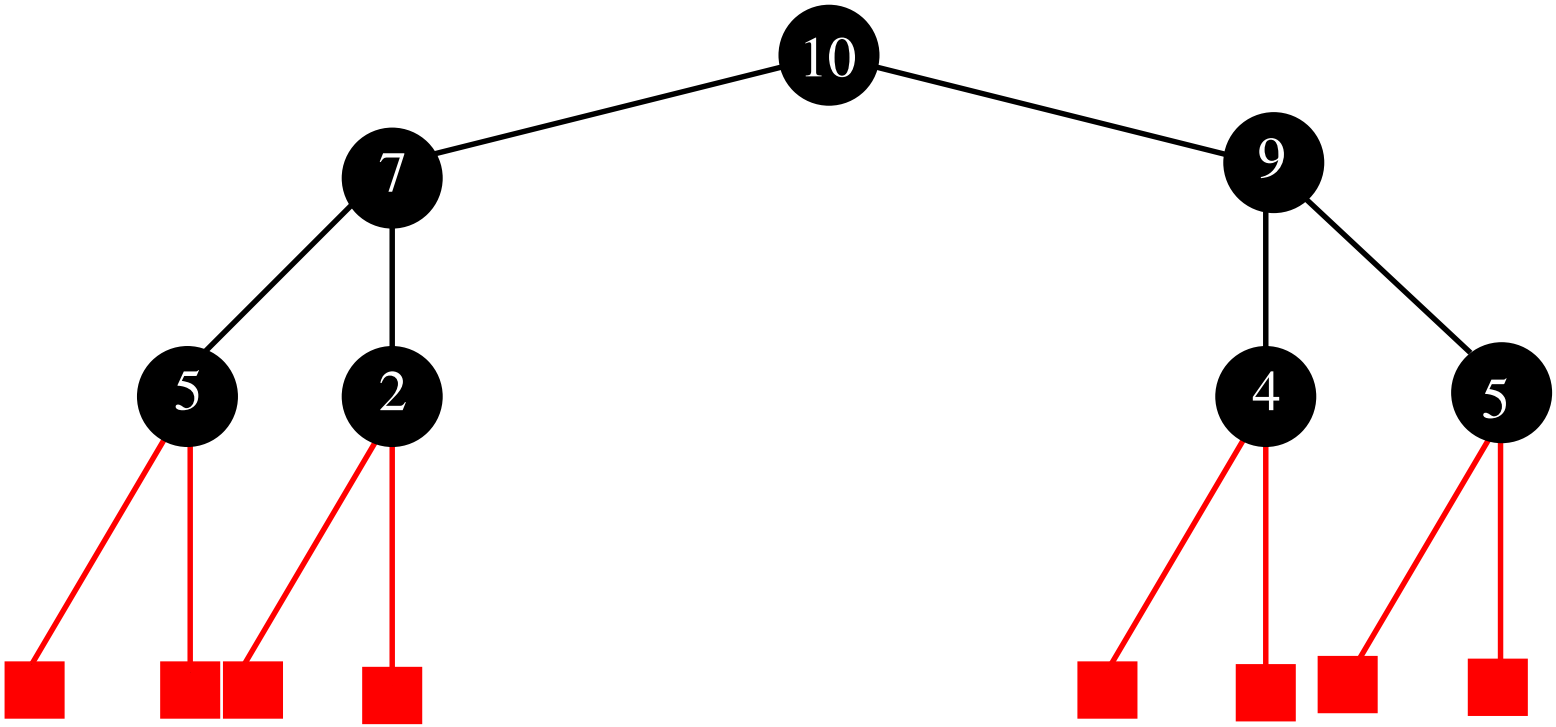
Montículo: añadir elemento

Añadir $\nu = (o, 10)$



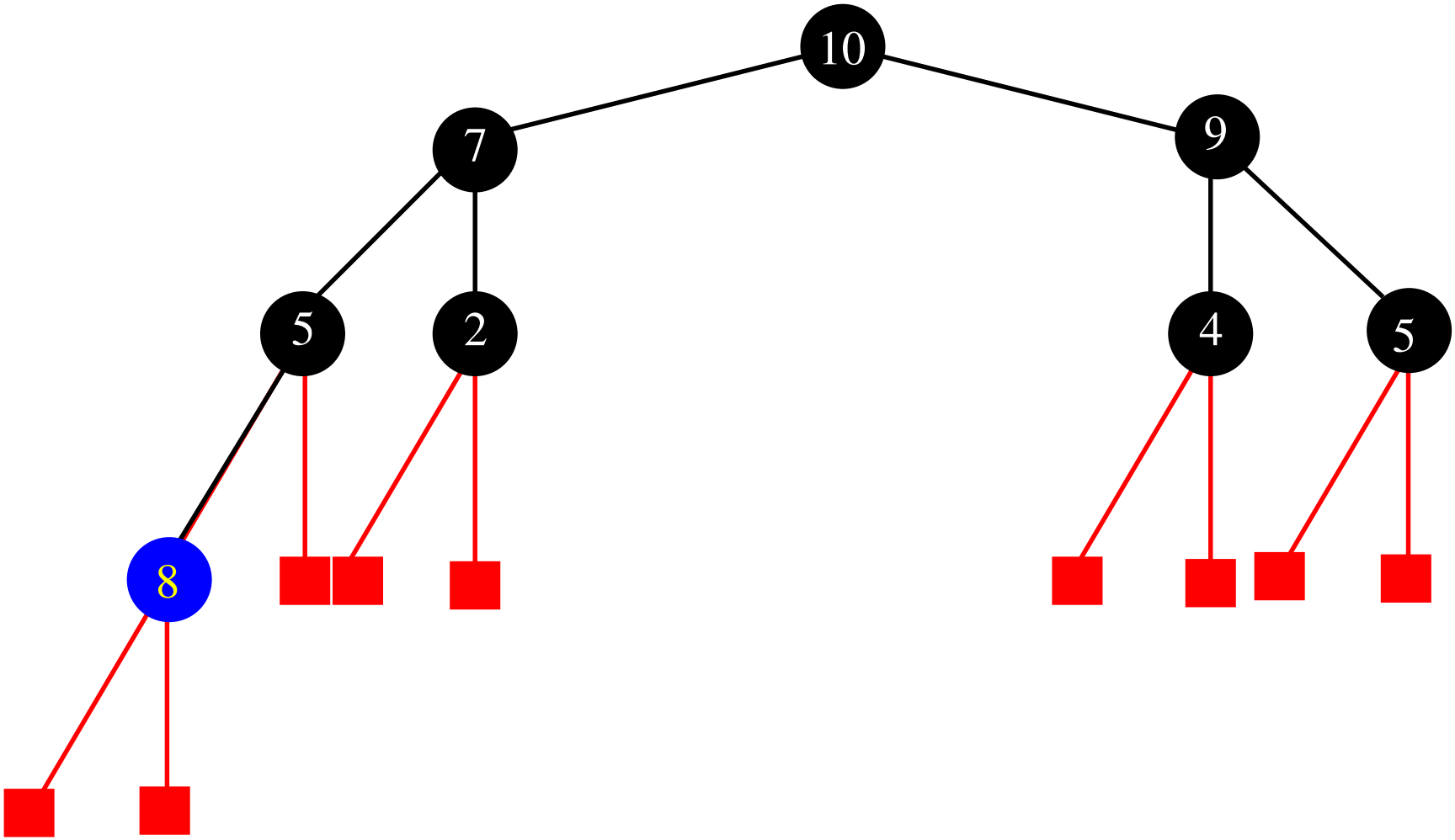
Montículo: añadir elemento

Añadir $\nu = (o, 8)$



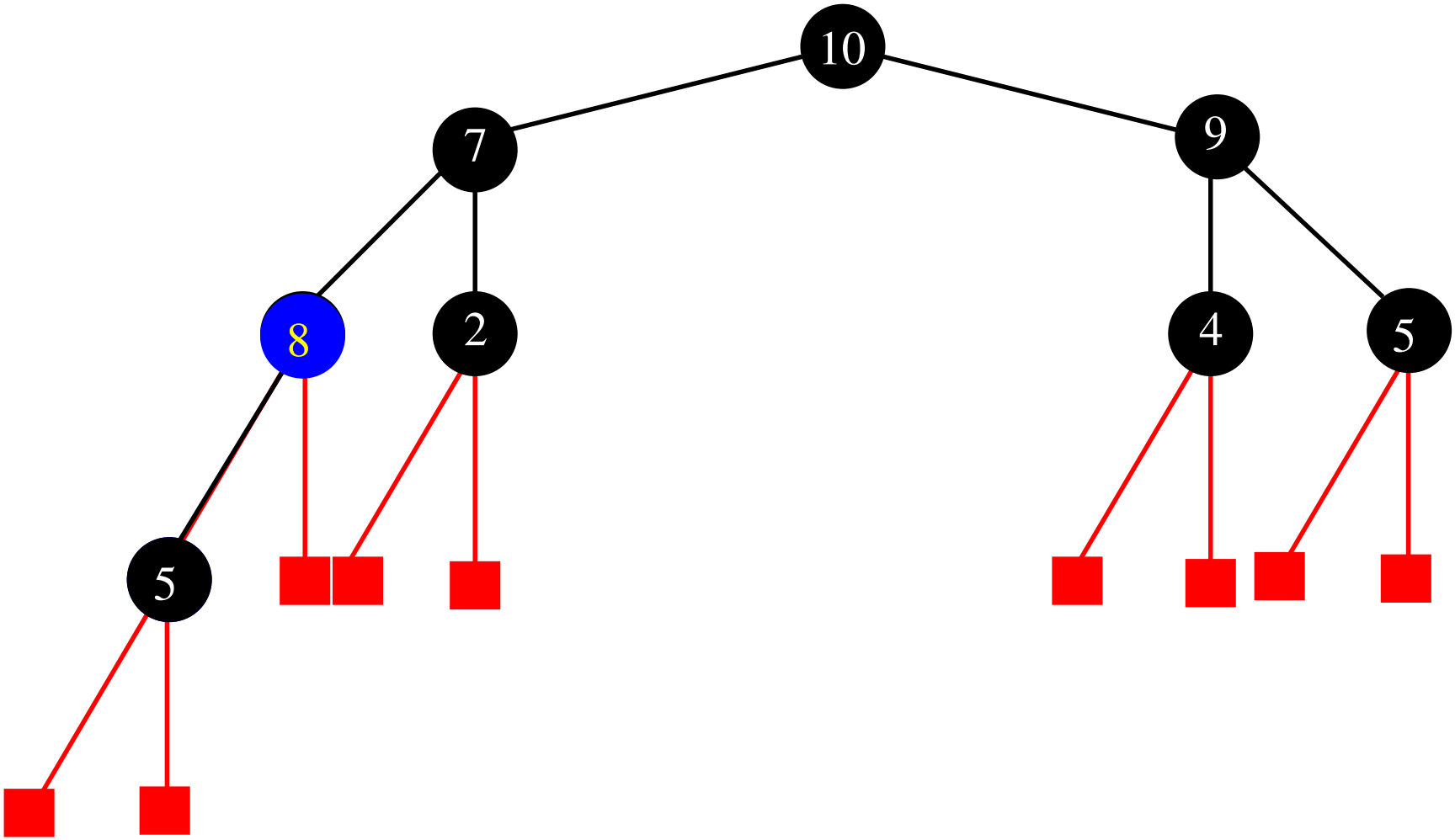
Montículo: añadir elemento

Añadir $\nu = (o, 8)$



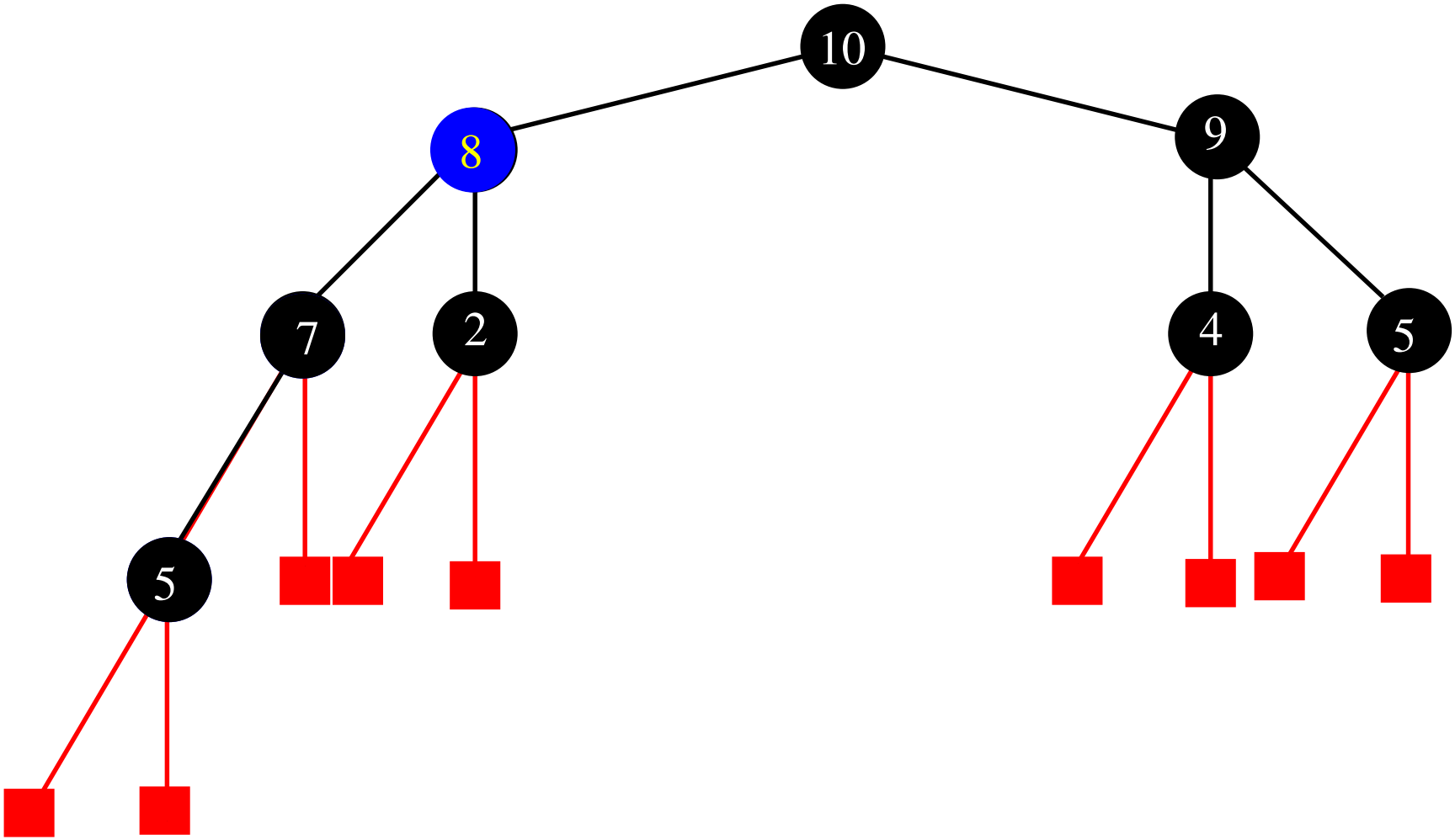
Montículo: añadir elemento

Añadir $\nu = (o, 8)$



Montículo: añadir elemento

Añadir $\nu = (o, 8)$



Montículo: añadir elemento

```
template <class T, class K>
void heap<T,K>::add(const T& o,
                    const K& p) {
  // Put element on last place
  data.push_back(std::pair<T,K>(o,p));
  unsigned int k = data.size()-1;
  while (k>=2 && data[k].second>data[k>>1].second) {
    std::pair<T,K> tmp = data[k>>1];
    data[k>>1] = data[k];
    data[k]    = tmp;
    k = k>>1;
  }
}
```

Montículo: añadir elemento

```
int main() {
  heap<int> h;
  h.add(5,1);  h.add(15,10);
  h.add(12,8); h.add(2,6);
  h.add(2,16); h.add(2,7);
  h.add(2,9);
  h.print();
  return 0;
}
```

Nos da:

```
16 10 9 1 6 7 8
```

que sí corresponde a un montículo.

Montículo: añadir elemento

- Tiempo para todas las operaciones involucradas: $O(\log N)$ en promedio.
- Peor caso como en promedio: $O(\log N)$.
- Mejor caso en $O(1)$.

Mejor que usar un arreglo ordenado pero peor que arreglo normal.

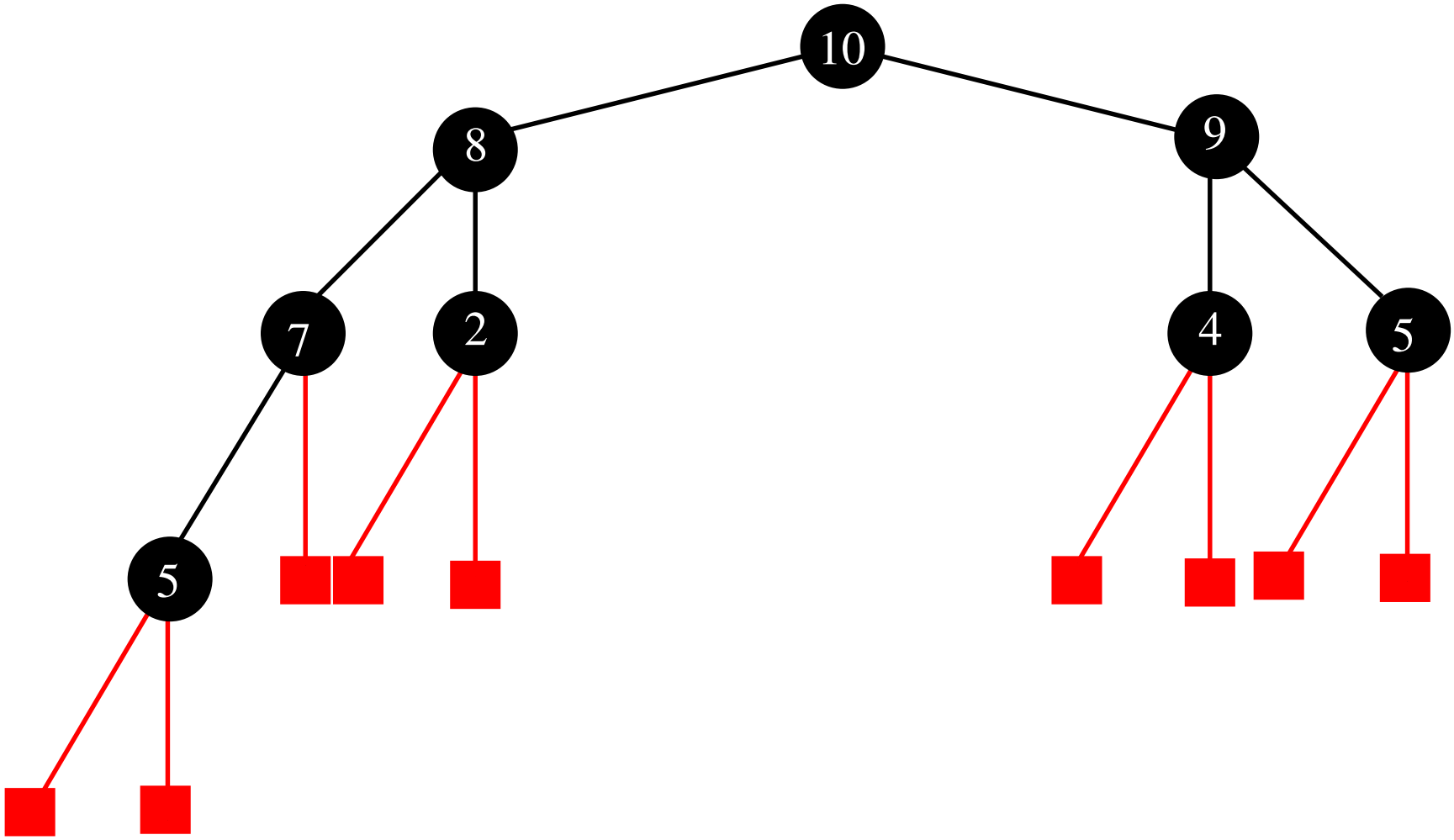
Montículo: quitar elemento máximo

Idea básica al quitar un nuevo elemento ν por $\mathrm{removeMax}$:

- Copiar el ultimo elemento del arreglo en la posición de la raíz y remover el ultimo elemento.
- Comparar la llave de este elemento con la de sus hijos; si uno es de llave superior, intercambiar el de valor máximo entre los hijos con la raíz.
- Así nos aseguramos que todo el árbol excepto el sub-arbol del intercambiado respeta la condición de montículo.
- Iterar hasta que $key(\nu) \geq key(left)$ y $key(\nu) \geq key(right)$.

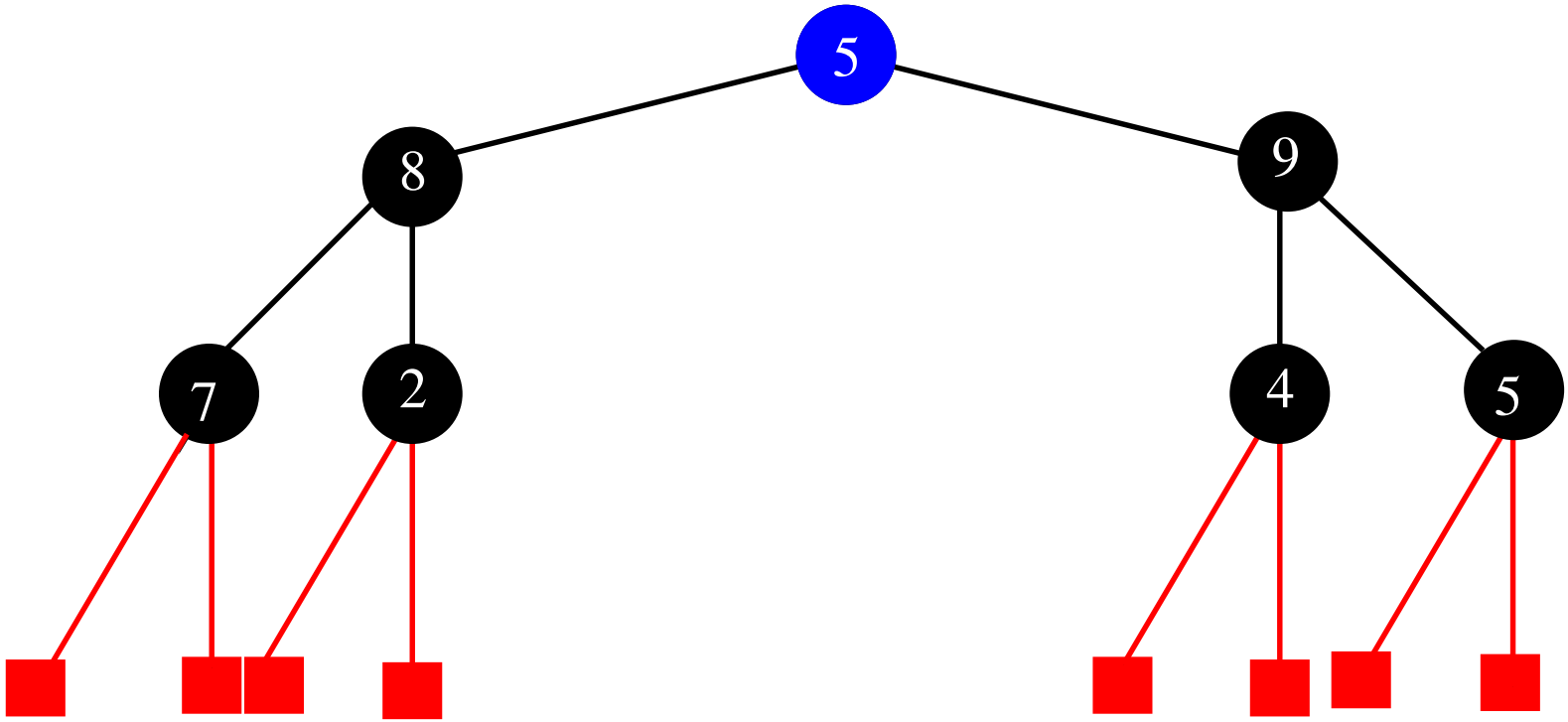
Montículo: quitar elemento máximo

Quitar el elemento de llave máxima



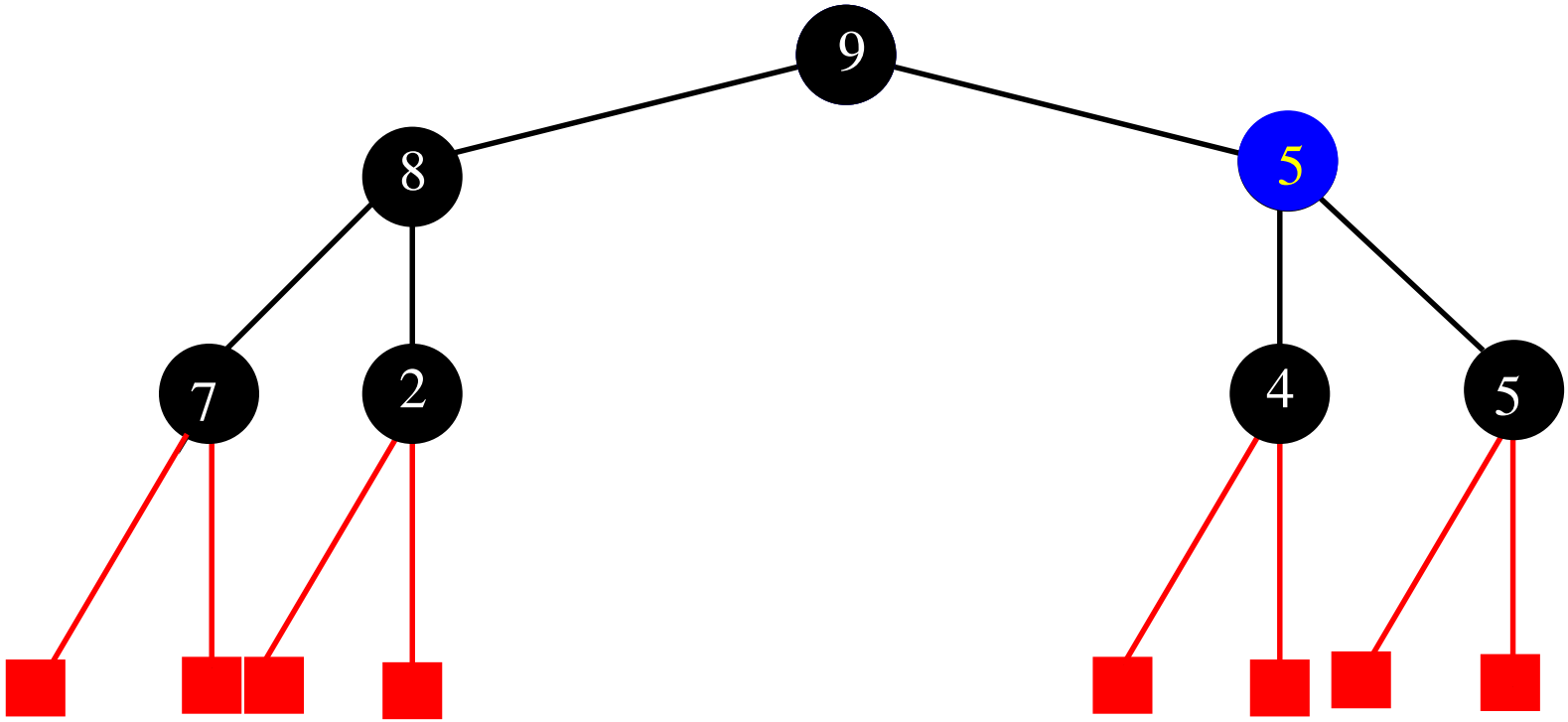
Montículo: quitar elemento máximo

Quitar el elemento de llave máxima



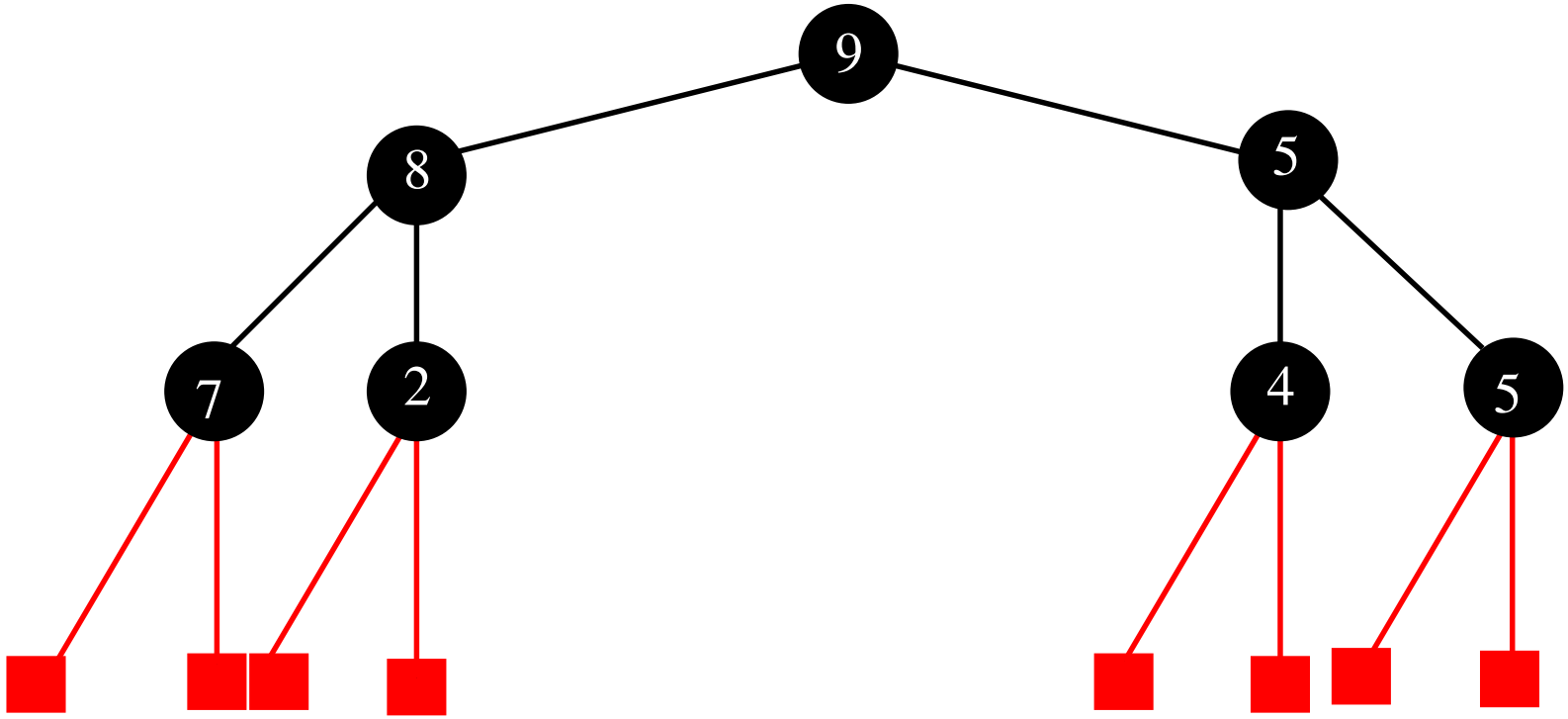
Montículo: quitar elemento máximo

Quitar el elemento de llave máxima



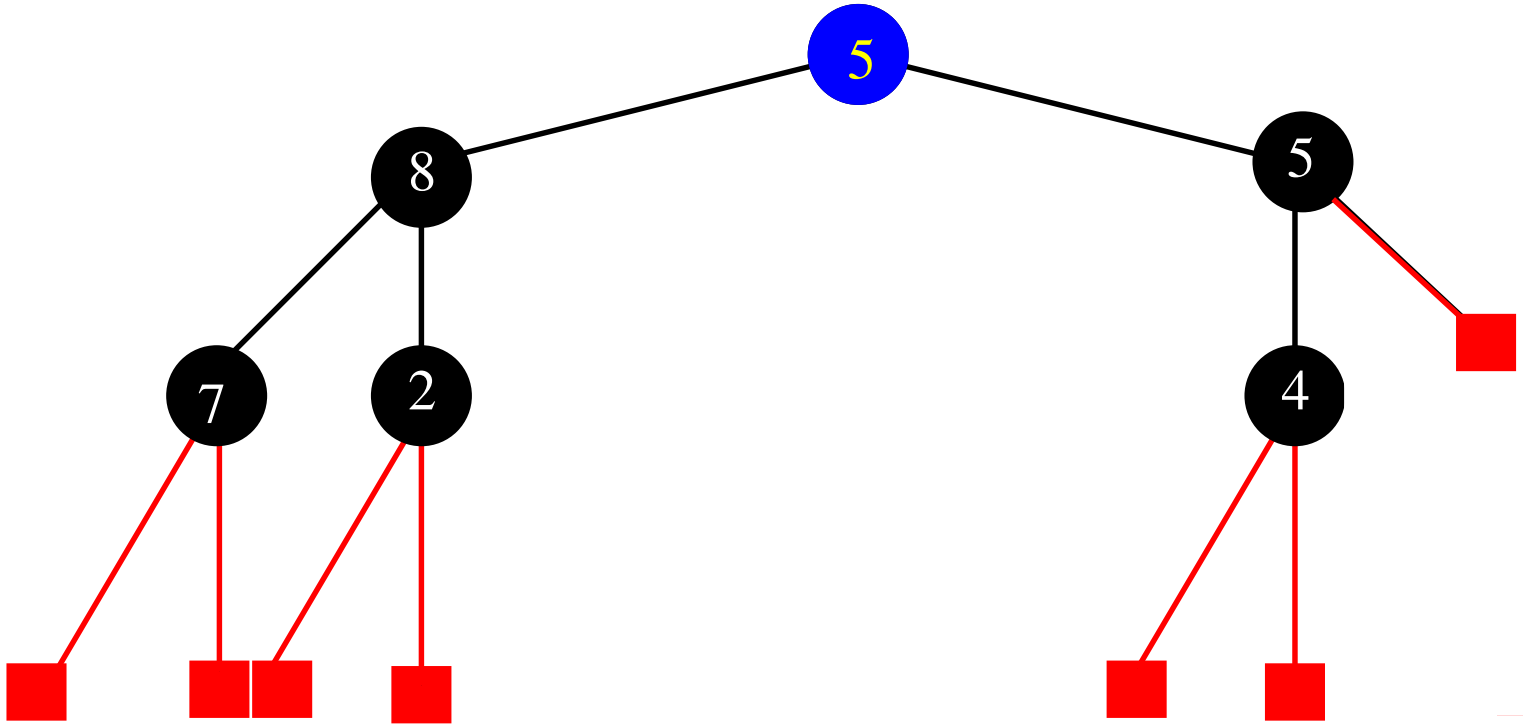
Montículo: quitar elemento máximo

Quitar de nuevo el elemento de llave máxima



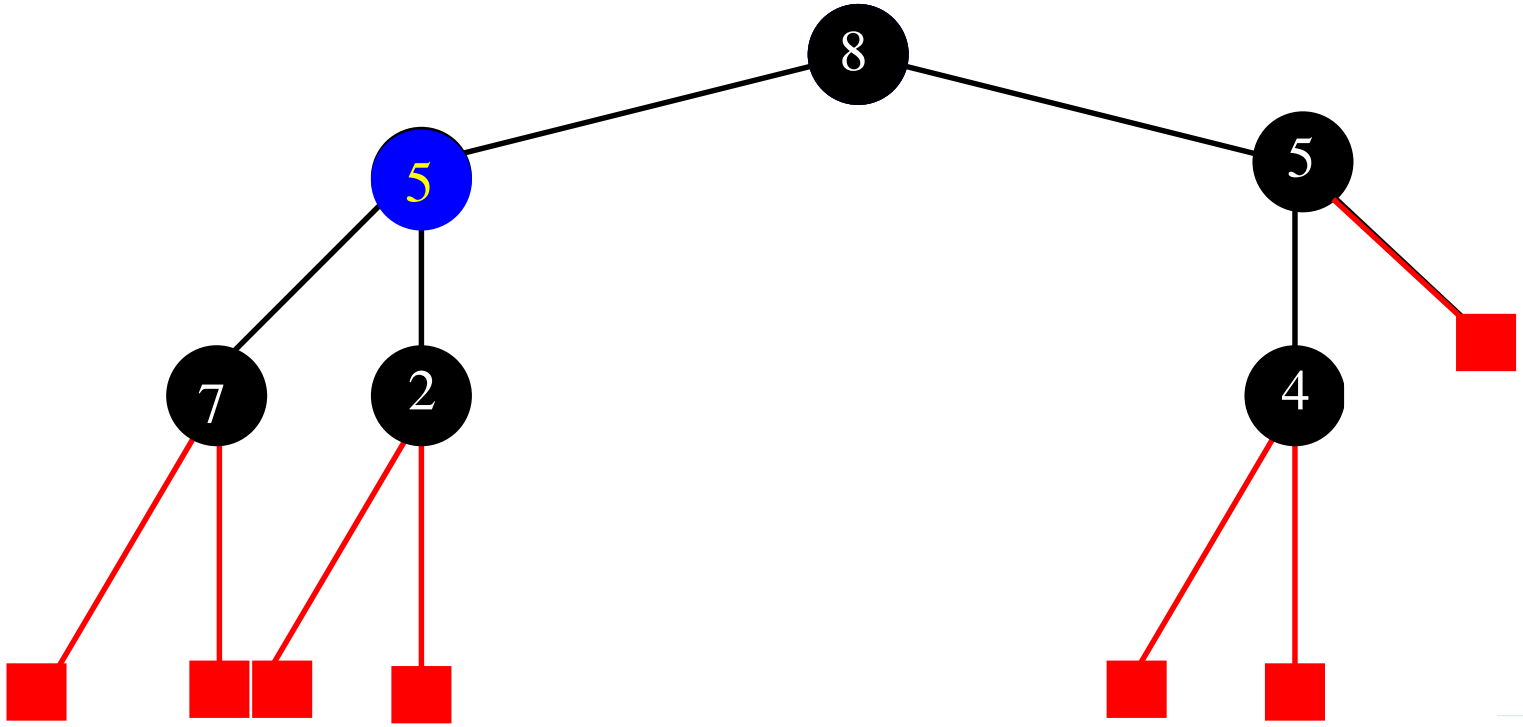
Montículo: quitar elemento máximo

Quitar de nuevo el elemento de llave máxima



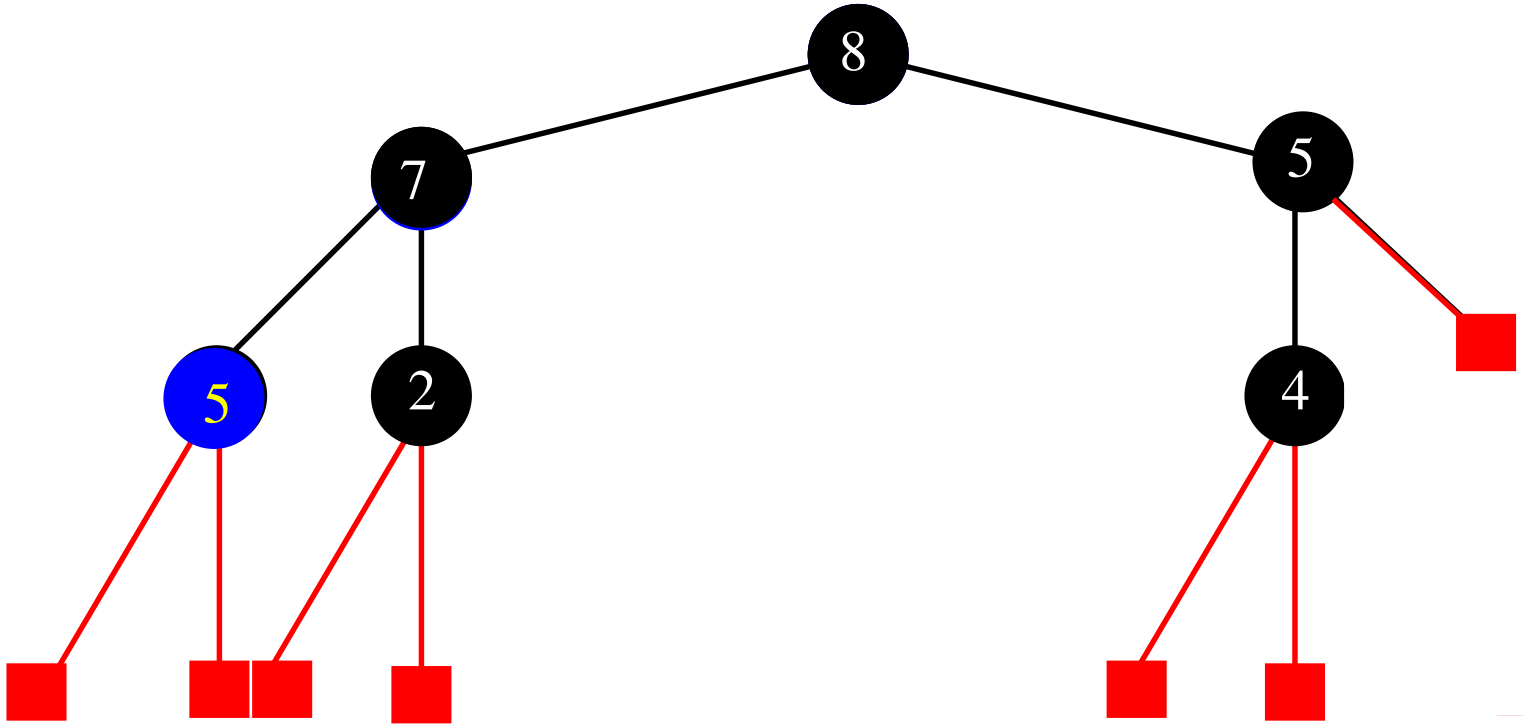
Montículo: quitar elemento máximo

Quitar de nuevo el elemento de llave máxima



Montículo: quitar elemento máximo

Quitar de nuevo el elemento de llave máxima



Montículo: quitar elemento máximo

```
template <class T,class K>
T heap<T,K>::removeMax() {
  if (data.size()<2) throw std::out_of_range("Empty heap");
  T saved=data[1].first;
  data[1] = data[data.size()-1]; data.pop_back();
  int k=1;
  while ((2*k   <data.size() && data[2*k   ].second>data[k].second)||
         (2*k+1<data.size() && data[2*k+1].second>data[k].second)){
    if (2*k+1>=data.size() || data[2*k].second>data[2*k+1].second) {
        std::pair<T,K> tmp = data[2*k];
        data[2*k] = data[k];
        data[k]   = tmp;
        k         = 2*k;
     } else {
        std::pair<T,K> tmp = data[2*k+1];
        data[2*k+1]= data[k];
        data[k]    = tmp;
        k          = 2*k+1;
    }
  }
  return saved;
}
```

Montículo: quitar elemento máximo

```
while (h.getSize()>0) {
    std::cout << h.getMax() << std::endl;
    h.removeMax();
    h.print();
 }
```

Nos da:

```
16
10 8 9 1 6 7
10
9 8 7 1 6
9
8 6 7 1
8
7 6 1
7
6 1
6
1
1
```

Montículo: quitar elemento máximo

- Otra vez, operación en $O(\log N)$, ya que al máximo se baja el nuevo elemento raíz según la altura del árbol.
- En total, operaciones de inserción de un nuevo elemento y de supresión del elemento de llave máxima en $O(\log N)$.

Montículo: otras operaciones

- Cambiar la llave de un objeto: como hacer?
- Quitar un objeto cualquiera: como hacer?
- Construir la fila de prioridad a partir de N elementos.

Montículo: otras operaciones

- Cambiar la llave de un objeto: como hacer?
  - Si la llave es mas chica, hacerlo bajar! (top-down).
  - Si la llave es mas grande, hacerlo subir! (bottom-up).
  - Todo en $O(\log N)$.
- Quitar un objeto cualquiera: como hacer?

.

- Construir la fila de prioridad a partir de N elementos.

Montículo: otras operaciones

- Cambiar la llave de un objeto: como hacer?
  - Si la llave es mas chica, hacerlo bajar! (top-down).
  - Si la llave es mas grande, hacerlo subir! (bottom-up).
  - Todo en $O(\log N)$.
- Quitar un objeto cualquiera: como hacer?
  Remplazar el objeto que remover por el ultimo del montículo y hacerlo bajar o subir, según el valor relativo de la nueva llave!, en $O(\log N)$.
- Construir la fila de prioridad a partir de N elementos.

Montículo: otras operaciones

- Cambiar la llave de un objeto: como hacer?
  - Si la llave es mas chica, hacerlo bajar! (top-down).
  - Si la llave es mas grande, hacerlo subir! (bottom-up).
  - Todo en $O(\log N)$.
- Quitar un objeto cualquiera: como hacer?
  Remplazar el objeto que remover por el ultimo del montículo y hacerlo bajar o subir, según el valor relativo de la nueva llave!, en $O(\log N)$.
- Construir la fila de prioridad a partir de N elementos.
  Introduciendo todos los elementos en un montículo inicialmente vacío: $O(N \log N)$

Montículo: otras operaciones

- Cambiar la llave de un objeto: como hacer?
  - Si la llave es mas chica, hacerlo bajar! (top-down).
  - Si la llave es mas grande, hacerlo subir! (bottom-up).
  - Todo en $O(\log N)$.
- Quitar un objeto cualquiera: como hacer?
  Remplazar el objeto que remover por el ultimo del montículo y hacerlo bajar o subir, según el valor relativo de la nueva llave!, en $O(\log N)$.
- Construir la fila de prioridad a partir de N elementos.
  Introduciendo todos los elementos en un montículo inicialmente vacío: $O(N \log N)$

Notar que las operaciones de tipo "monticulización" no dependen de la estructura usada para representar el árbol!

Montículo: otras operaciones

- Finalmente, solo dos primitivas!
- Entonces prever en el caso de mas operaciones, implementar métodos moveUp y $\mathrm{moveDown}$ y expresar todas en función de estas dos.

Outline

Aplicación: fila de prioridad

- $\mathrm{push}()$: usando $\mathrm{add}()$ del montículo, $O(\log N)$.
- $\mathrm{pop}()$: usando $\mathrm{removeMax}()$, $O(\log N)$.
- $\mathrm{top}()$: usando $\mathrm{getMax}()$, $O(1)$.

Para un uso "normal" (con inserciones y deleciones) ¡usar un montículo!

Aplicación: fila de prioridad

	Inserción	Quitar máximo	Quitar elemento	Máximo	Cambiar llave
Arreglo	$O(1)$	$O(N)$	$O(N)$	$O(N)$	$O(1)$
Lista	$O(1)$	$O(N)$	$O(1)$	$O(N)$	$O(1)$
Arreglo ord.	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(N)$
Lista ord.	$O(N)$	$O(1)$	$O(1)$	$O(1)$	$O(N)$
Montículo	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(1)$	$O(\log N)$

Aplicación: fila de prioridad

Una aplicación: algoritmo de Dijsktra usando la STL

```
for (int i = 0; i<nodeMax;i++)
    shortDist[i] = std::numeric_limits<int>::max();
std::priority_queue<Node> heap;
heap.push(Node(from, 0));
while (!heap.empty()) {
    Node curNode = heap.top(); heap.pop();
    if (shortDist[curNode.getIndex()] <
      std::numeric_limits<int>::max())
      continue;
    shortDist[curNode.getIndex()] = curNode.getKey();
    for (Edge *e = edges[curNode.getIndex()];
            e!=NULL; e = e->next)
     heap.push(Node(e->to,
            curNode.getKey()+e->getKey()));
}
```

Aplicación: HeapSort

Aplicación muy directa del montículo: el ordenamiento de tipo $\mathrm{HeapSort}$, cuya idea basica consiste en:

1. poner todos los elementos uno a uno en un montículo ($O(N \log N)$ en el caso peor).
2. sacar el máximo en un ciclo de iteraciones, $O(N \log N)$.

Pero: requiere el doble de memoria que la estructura, y la construcción del montículo se puede hacer mas eficientemente!

Aplicación: HeapSort

Un código ingenuo:

```
template <class T>
void heapSort(std::vector<T> &v) {
  heap<T,T> h;
  // Fill heap
  for (int i=0;i<v.size();i++)
    h.add(v.at(i),v.at(i));
  // Empty heap and get maxs back in v
  int k=0;
  while (h.getSize()>0) {
    v.at(k++)= h.removeMax();
  }
}
```

Aplicación: HeapSort

Ahora, existe un algoritmo mas sencillo para formar un montículo (en sitio) a partir de un arreglo:

- recorrer todos los nodos a partir de los mas abajo,
- considerar cada uno como una raíz de un montículo,
- si necesario (el nodo es de llave inferior a uno de sus hijos), "monticulizar".

Aplicación: HeapSort

Otra manera de verlo, recursivamente hacer montículos:
BottomUpHead(S)

Require: Sequence S with N pairs (Object,Key)
Ensure: A heap T with the pairs in S.

```
if S is empty then
    return empty heap
end if
Remove the first pair (o,k) from S
Split S into two sequences S1 and S2
T1 ← BottomUpHeap(S1)
T2 ← BottomUpHeap(S2)
create binary tree T with (o,k) at root, T1 at left, T2 at right
If necessary, performs bubbling of (o,k)
return T
```

Aplicación: HeapSort

Complejidad: supongamos $N = 2^h - 1$ (árbol binario perfecto).

- Entonces, consideraríamos primero los 2^{h-2} montículos del nivel $h-2$, y nos costaría al máximo 1 bajada en cada uno.
- Luego, considerar los nodos del nivel arriba, que son 2^{h-3}, y en qué al máximo nos costara 2 bajadas.
- En total, el costo en numero de bajadas es:

$$\begin{aligned} \sum_{k=1}^{h-1} k2^{h-1-k} &= S_h \\ &= 2S_{h-1} + (h-1) \\ &= S_{h-1} + 2^{h-1} - 1. \end{aligned}$$

Entonces:

$$S_h = 2^h - h - 1 \quad .$$

Aplicación: HeapSort

Complejidad: supongamos $N = 2^h - 1$ (árbol binario perfecto).

- Entonces, consideraríamos primero los 2^{h-2} montículos del nivel $h-2$, y nos costaría al máximo 1 bajada en cada uno.
- Luego, considerar los nodos del nivel arriba, que son 2^{h-3}, y en qué al máximo nos costara 2 bajadas.
- En total, el costo en numero de bajadas es:

$$\begin{aligned} \sum_{k=1}^{h-1} k2^{h-1-k} &= S_h \\ &= 2S_{h-1} + (h-1) \\ &= S_{h-1} + 2^{h-1} - 1. \end{aligned}$$

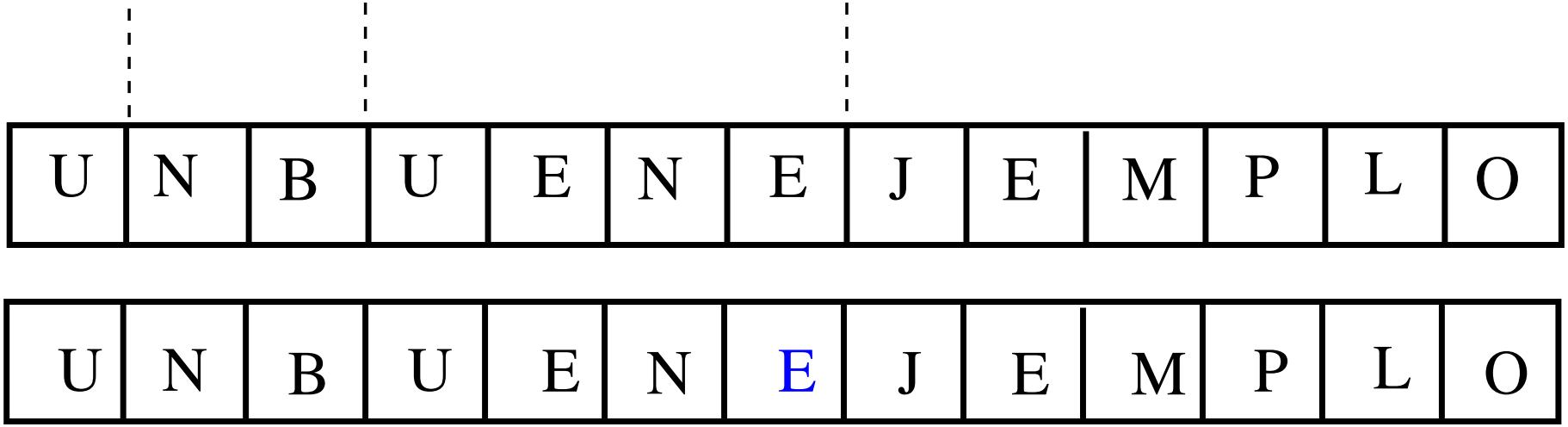
Entonces:

$$S_h = 2^h - h - 1 < N.$$

Lineal, <2N comparaciones!, dominado por la segunda parte.

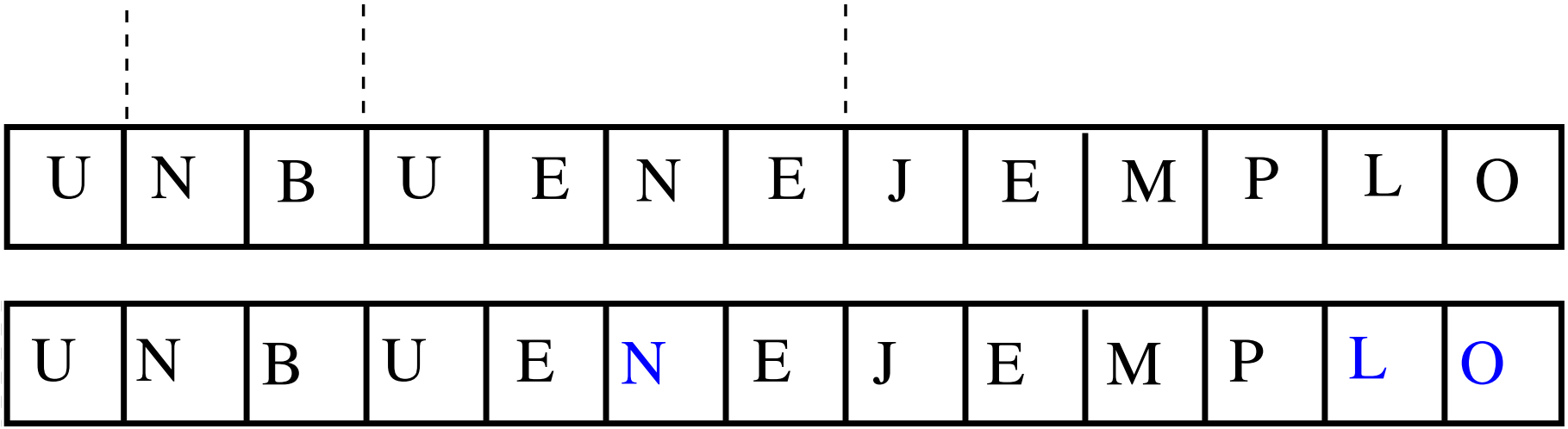
Aplicación: HeapSort

Construcción del montículo:



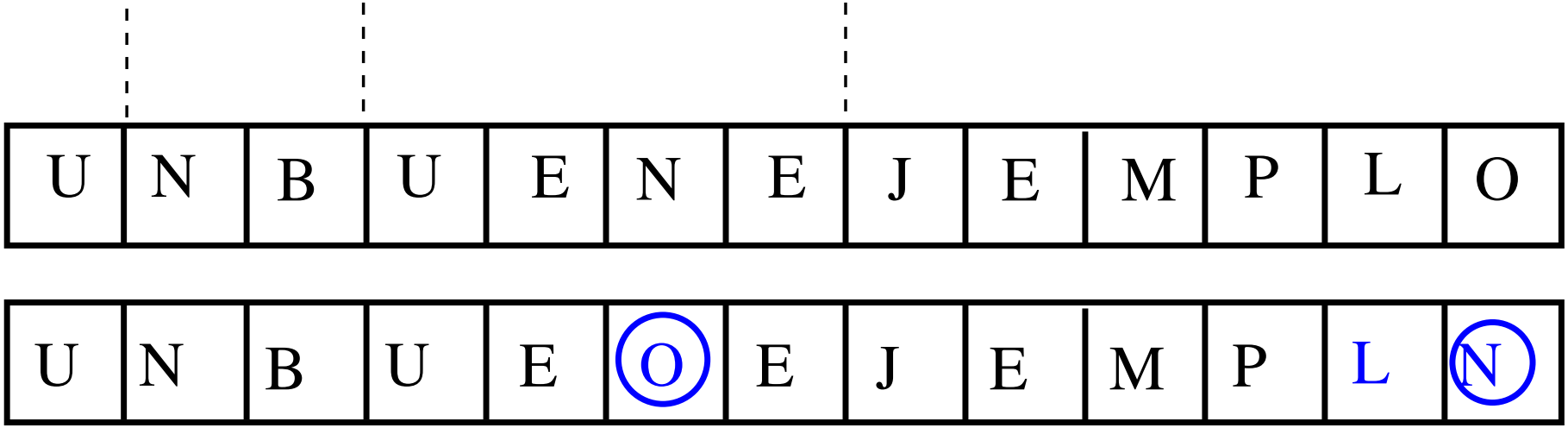
Aplicación: HeapSort

Construcción del montículo:



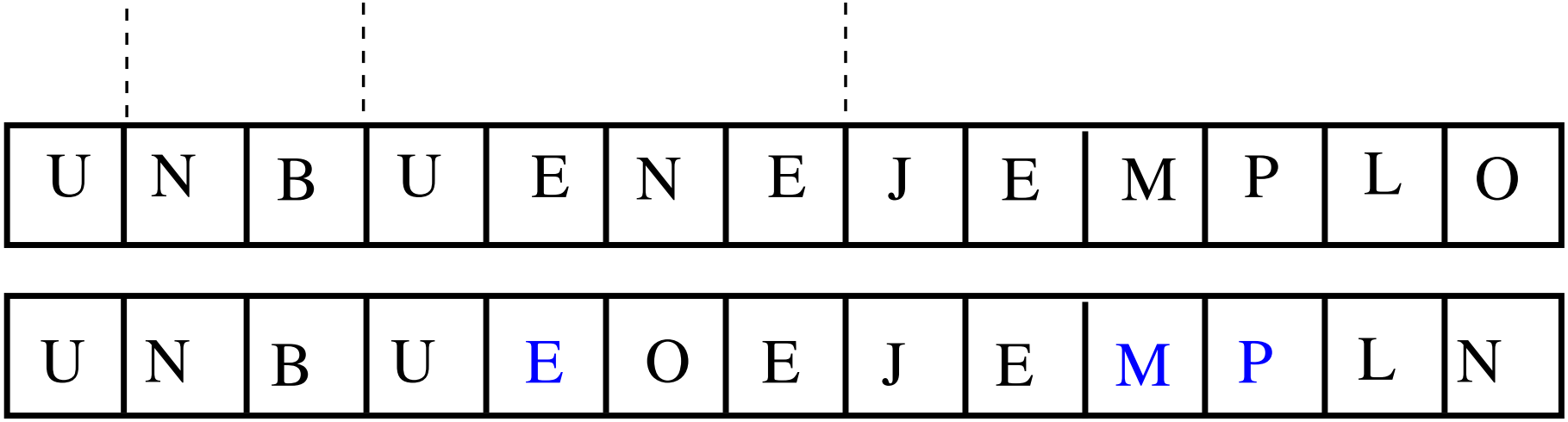
Aplicación: HeapSort

Construcción del montículo:



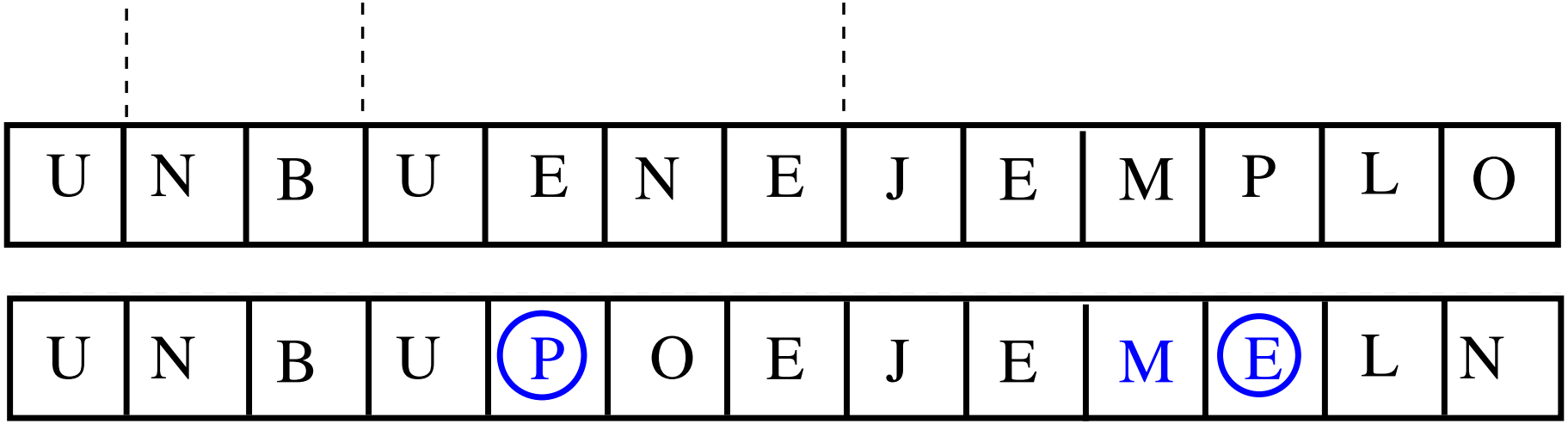
Aplicación: HeapSort

Construcción del montículo:



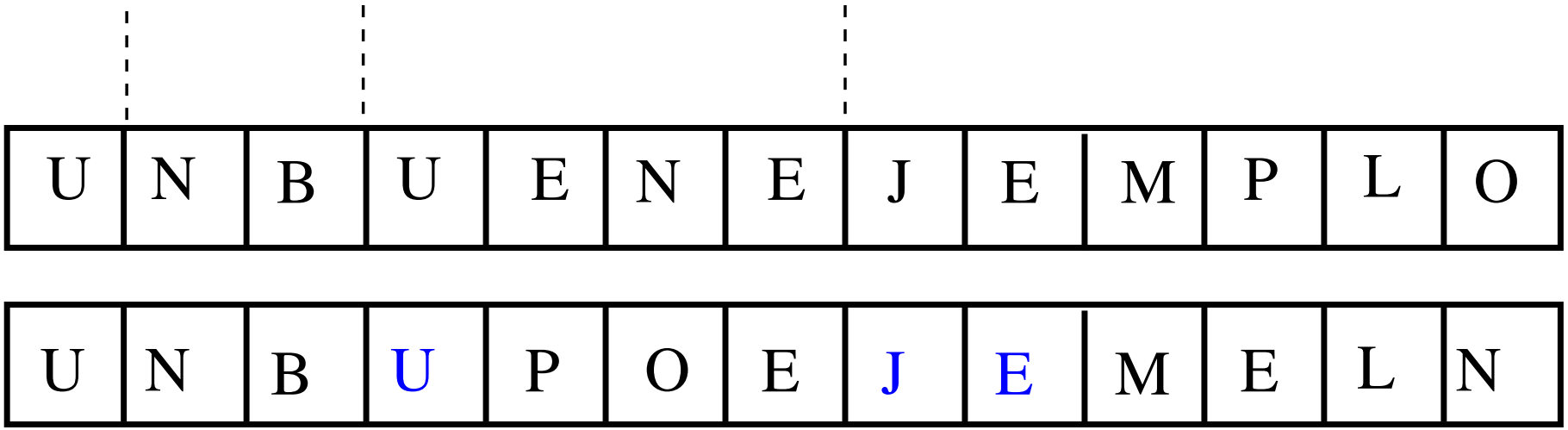
Aplicación: HeapSort

Construcción del montículo:



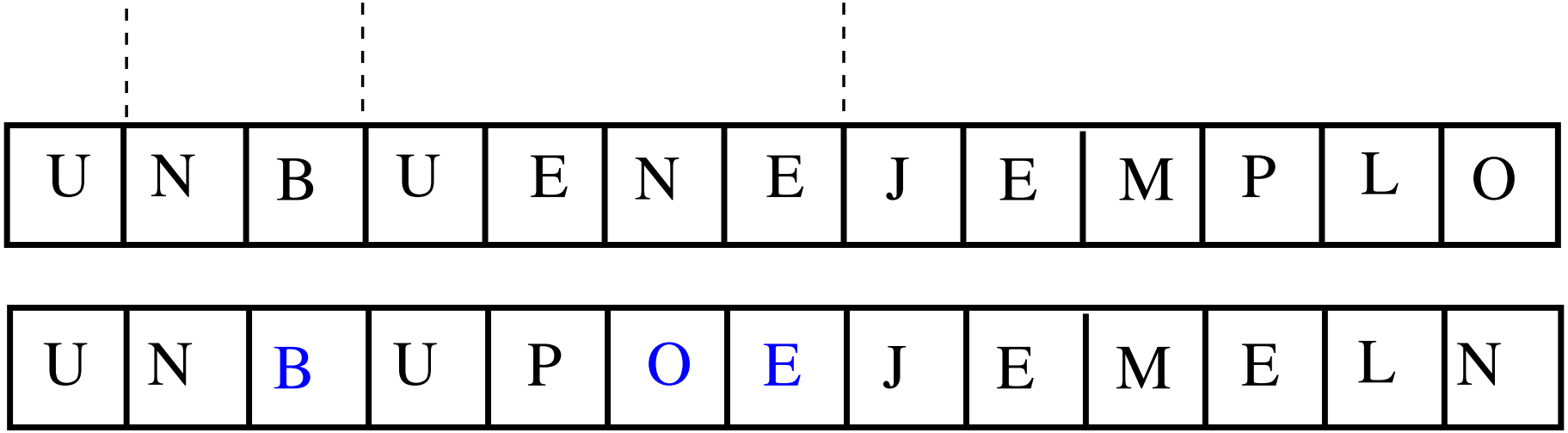
Aplicación: HeapSort

Construcción del montículo:



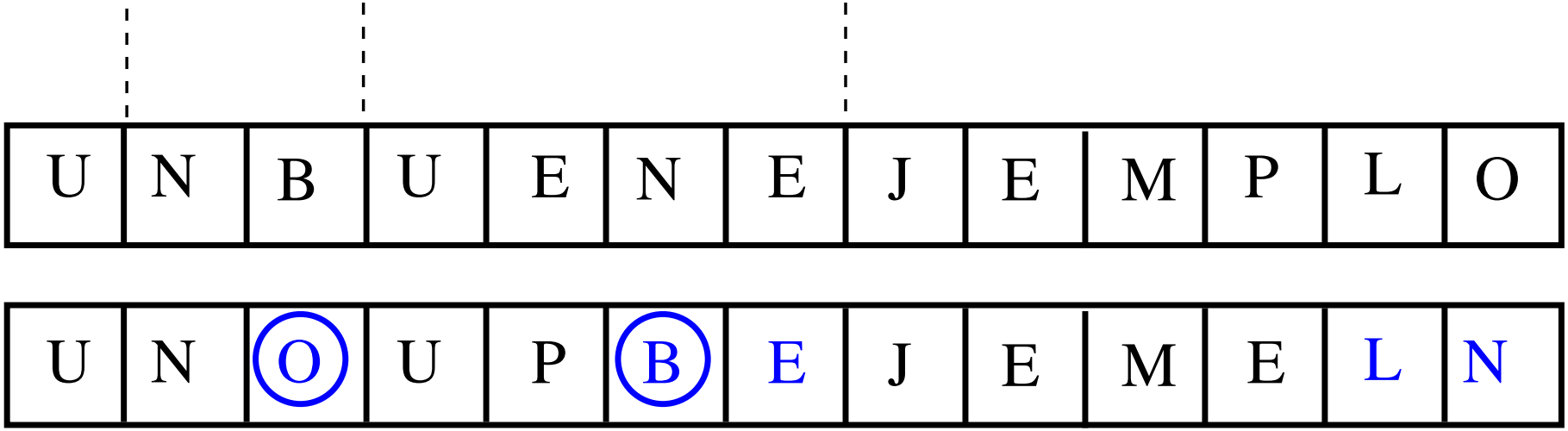
Aplicación: HeapSort

Construcción del montículo:



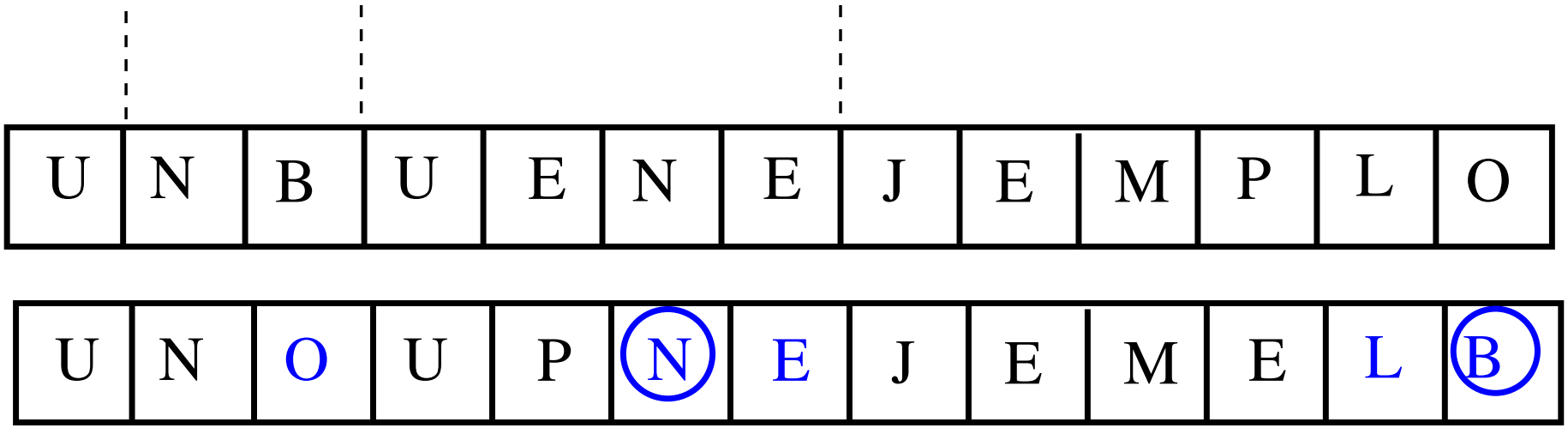
Aplicación: HeapSort

Construcción del montículo:



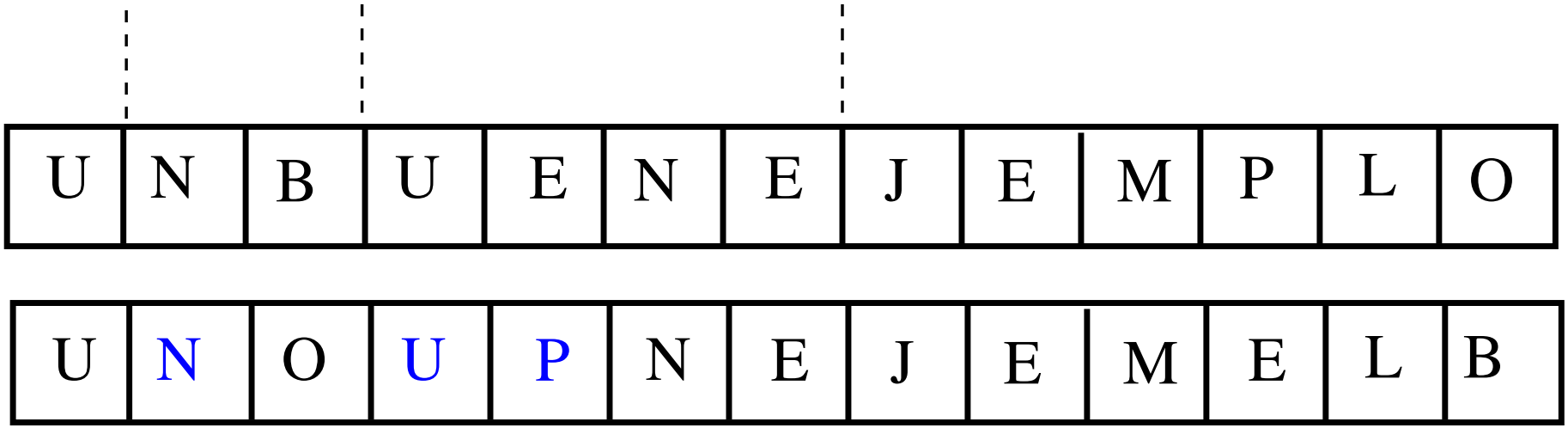
Aplicación: HeapSort

Construcción del montículo:



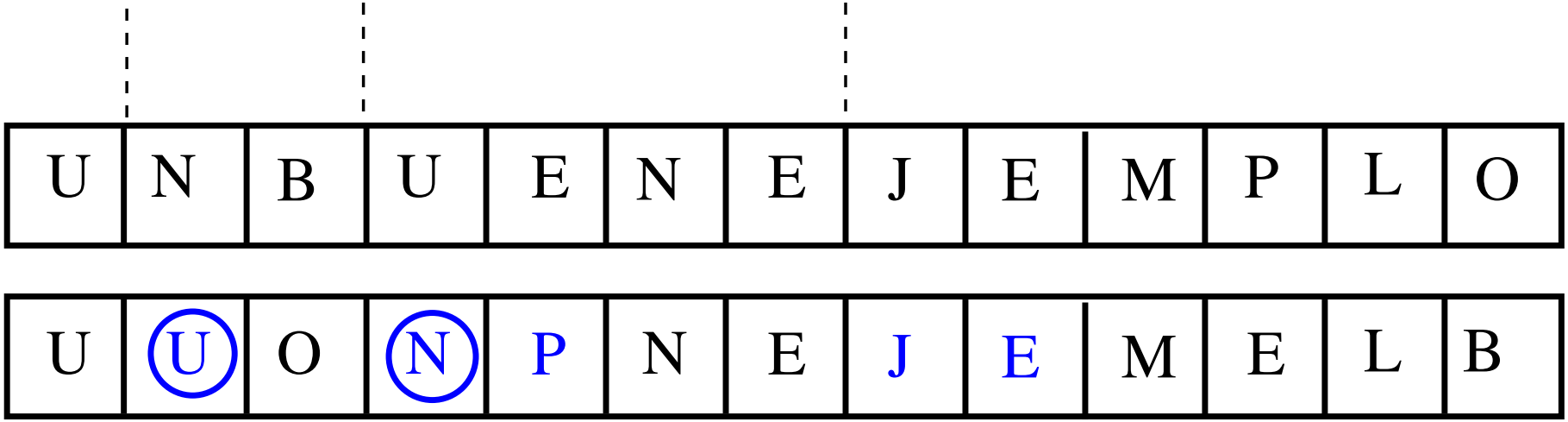
Aplicación: HeapSort

Construcción del montículo:



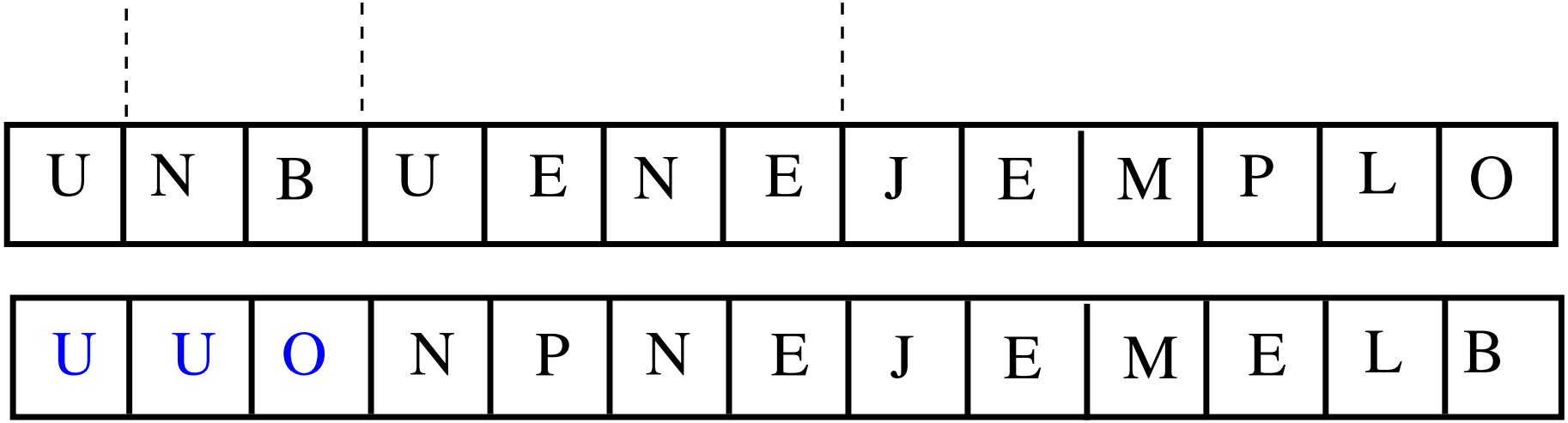
Aplicación: HeapSort

Construcción del montículo:



Aplicación: HeapSort

Construcción del montículo:

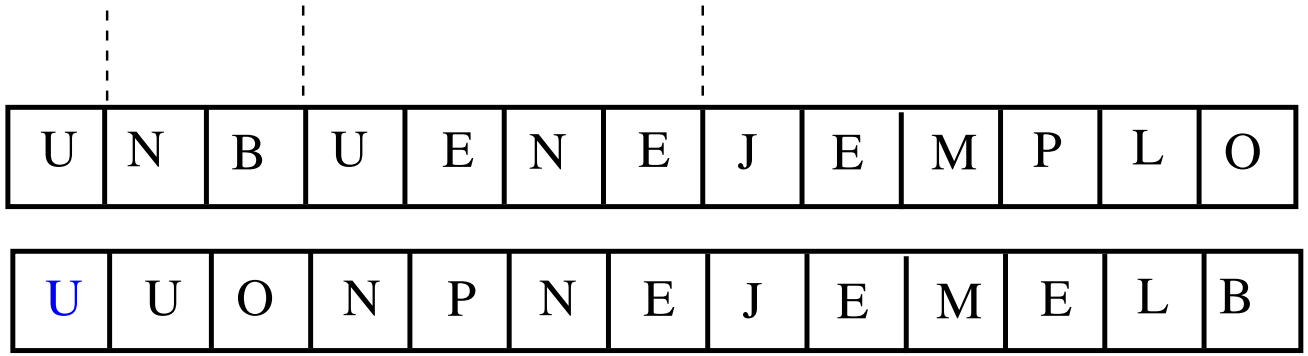


Aplicación: HeapSort

- Una construcción de montículo como explicado se hace en tiempo lineal, contra $N \log N$ para el algoritmo ingenuo.
- HeapSort: primera parte en N.
- HeapSort: segunda parte en sitio con sucesión de movidas del elemento final atrás ($2N \log N$).

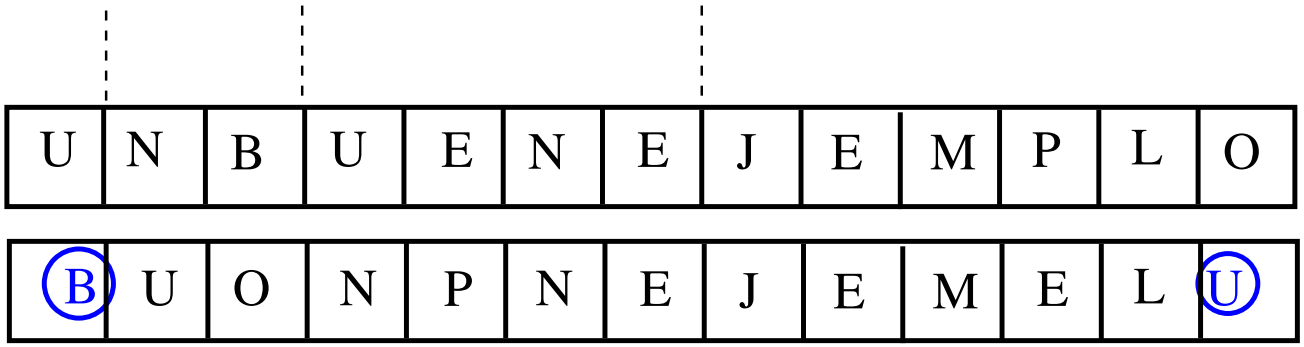
Aplicación: HeapSort

Segunda parte: remover iterativamente el elemento máximo (que esta al principio) intercambiándolo con el elemento final y bajar cada ver el elemento cambiado



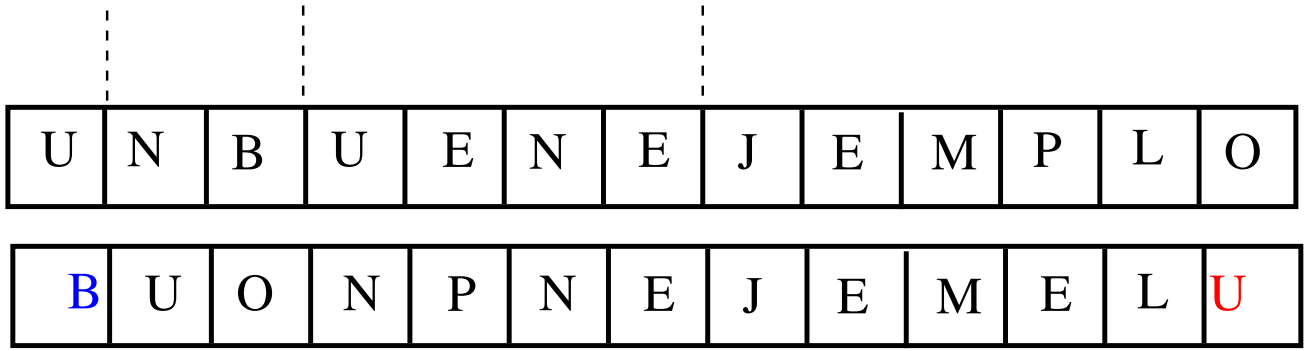
Aplicación: HeapSort

Segunda parte: remover iterativamente el elemento máximo (que esta al principio) intercambiándolo con el elemento final y bajar cada ver el elemento cambiado



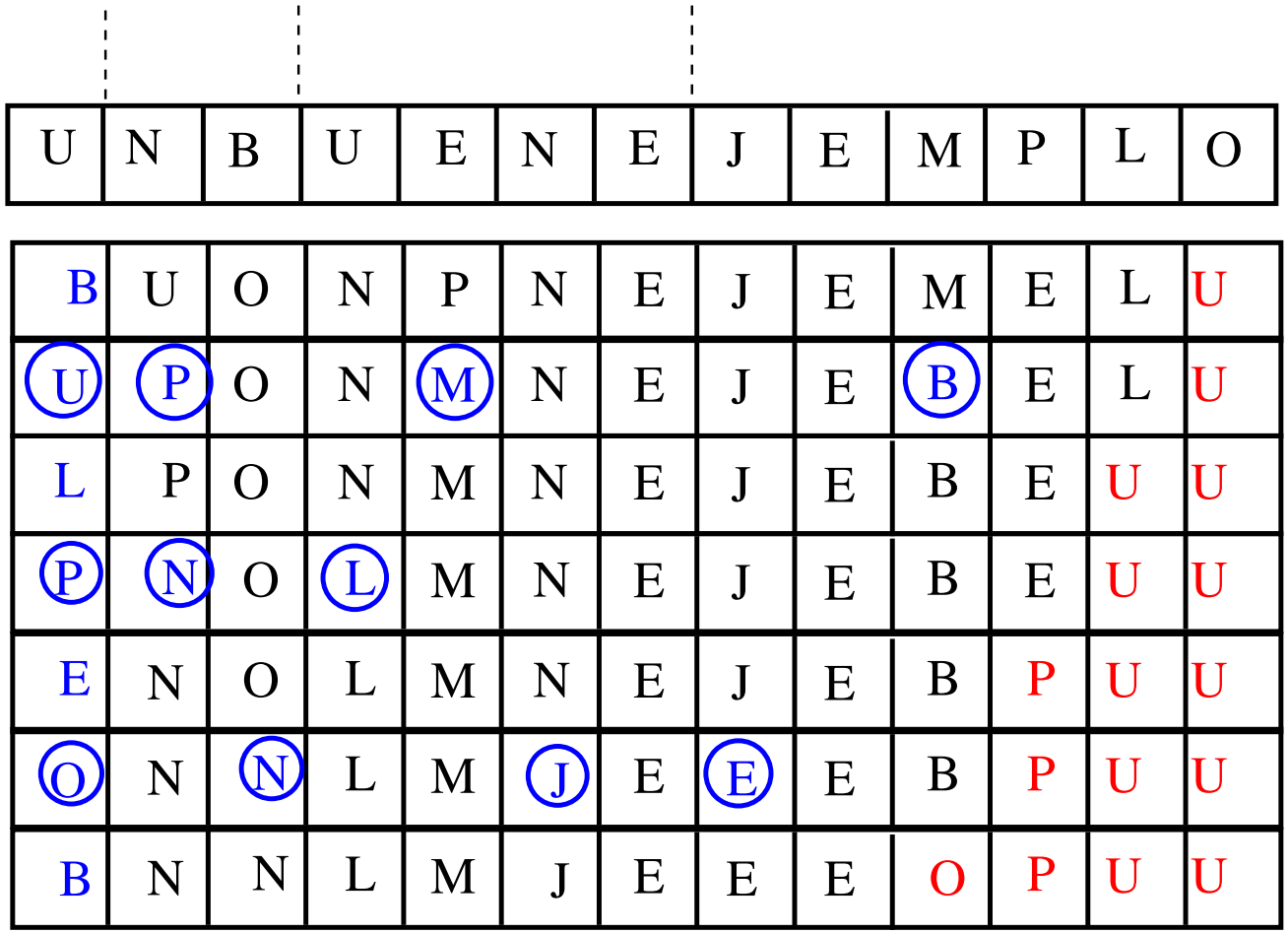
Aplicación: HeapSort

Segunda parte: remover iterativamente el elemento máximo (que esta al principio) intercambiándolo con el elemento final y bajar cada ver el elemento cambiado



Aplicación: HeapSort

Segunda parte: remover iterativamente el elemento máximo (que esta al principio) intercambiándolo con el elemento final y bajar cada ver el elemento cambiado



Aplicación: k elementos mas grandes

Opción 1:

- Construir montículo ($< 2N$ comparaciones).
- Sacar k elementos máximos consecutivamente ($< 2k \log N$ comparaciones).

Opción 2:

- Construir montículo-min con los k primeros elementos ($< 2k$ comparaciones).
- Hacer $N - k$ operaciones de inserción-remover mínimo ($< 2(N - k) \log k$ comparaciones).

Tiempo lineal ($O(N)$) cuando k pequeño o k cerca de N, $O(N \log N)$ sino.

Variaciones

- Usar arboles ternarios... Todo el problema es encontrar un compromiso justo entre el numero de comparaciones que hacer en la monticulización y la altura del árbol
  - En una operación elemental de bajada, en el mejor caso, tres comparaciones en cambio de 2 en el caso binario ($3 \log_3 N = \frac{3}{\log_2 3} \log_2 N \approx 1.89 \log_2 N$ contra $2 \log_2 N$); no ventaja en añadir un orden mas.
- Mejorar la segunda parte del ordenamiento de HeapSort: el elemento re-bajado desde arriba...

Variaciones: primera versión

```
template <class T, class K>
void heap<T,K>::moveDown(int curr,
                         std::pair<K,T> &paseador) {
int child=2*curr+1;
while(child<data.size()) {
  if (data[child].second<data[child-1].second)
   child--;
  if(data[child].second>paseador.second) {
    data[curr]=data[child], curr=child, child=2*curr+1;
  } else break;
}
if((child==data.size()) &&
    data[data.size()-1].second>paseador.second)
    data[curr]=data[data.size()-1], curr=data.size()-
data[curr]=paseador;
}
```

Variaciones: versión mejorada

```
template <class T, class K>
void heap<T,K>::moveDown(int curr,
                         std::pair<K,T> &paseador) {
int memo=vacant;
int child, parent;

int child=2*curr+1;
while(child<data.size()) {
  if (data[child].second<data[child-1].second)
    child--;
  data[curr]=data[child], curr=child, child=2*curr+1;
}
if((child==data.size()))
  data[curr]=data[data.size()-1], curr=data.size()-1;
....
```

Variaciones: versión mejorada

```
int parent= curr/2;
while(curr>memo) {
   if(data[parent].second<paseador.second) {
      data[curr]=data[parent],curr=parent, parent=curr
   } else
      break;
}
data[curr]=paseador;
}
```

Simple y lleva una ganancia de 25% en promedio