Chapter #9:
HEAP STRUCTURES
-MIN-MAX HEAPS-

Fundamentals of Data Structures in C

Horowitz, Sahni and Anderson-Freed

Computer Science Press

MIN-MAX HEAPS

Double-ended priority queue

```
1) insert an element with arbitrary key
2) delete an element with the largest key
3) delete an element with the smallest key

- max heap supports operations 1) and 2)
- min heap supports operations 1) and 3)

- min-max heap supports all of the above
  operations
```

MIN-MAX HEAPS

```
Def) A min-max heap is a complete
  binary tree such that if it is not
  empty, each element has a field
  called key
- alternating levels of the tree are
  min levels and max levels
- the root is on a min level

- min node and max node
  (on next page)
```

MIN-MAX HEAPS

```
let x: any node in a min-max heap
```

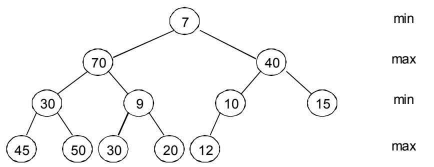
min node

```
- x is on a min level
- x has the minimum key from among
  all elements in the subtree with
  root x
```

max node

```
- x is on a max level
- x has the maximum key from among
  all elements in the subtree with
  root x
```

MIN-MAX HEAPS

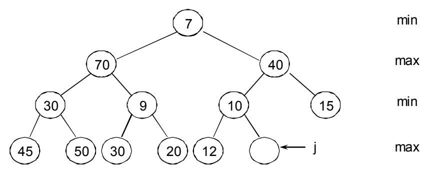


a 12 element min-max heap

MIN-MAX HEAPS

Inserting into a min-max heap

Ex) insert the element with key 5 into the following min-max heap

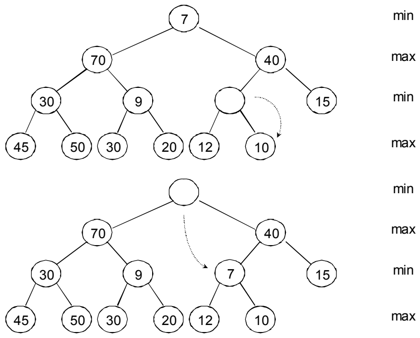


min-max-heap with new node *j*

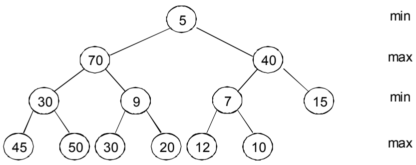
MIN-MAX HEAPS

comparing the new key 5 with the key 10 that is in the parent of *j*
- node with key 10 is on a min level
- 5 < 10
- 5 is guaranteed to be smaller than all keys in nodes that are both on max levels and on the path from *j* to root
- min-max-heap property is to be verified only with respect to min nodes on the path from *j* to the root

MIN-MAX HEAPS



MIN-MAX HEAPS



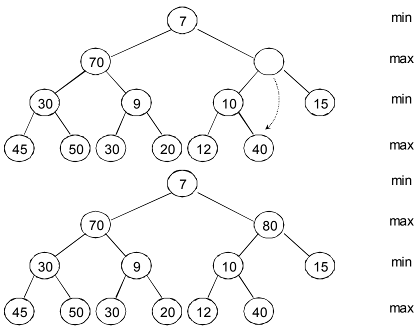
min-max-heap after inserting 5

MIN-MAX HEAPS

Ex)insert the element with key 80 into the min-max heap

comparing the new key 80 with the key 10 that is in the parent of *j*
- node with key 10 is on a min level
- 80 > 10
- 80 is guaranteed to be larger than all keys in nodes that are both on min levels and on the path from *j* to root
- min-max-heap property is to be verified only with respect to max nodes on the path from *j* to the root

MIN-MAX HEAPS



min-max-heap after inserting 80

MIN-MAX HEAPS

C declarations for min-max heap

```
#define MAX_SIZE 100
/* maximum size of heap plus 1 */
#define FALSE 0
#define TRUE 1
#define SWAP(x,y,t) ((t)=(x),(x)=(y),(y)=(t))
typedef struct {
   int key;
   /* other fields */
} element;
element heap[MAX_SIZE];
```

MIN-MAX HEAPS

function *level()*

- determine whether a node is on a min level or on a max level of a min-max heap
- return FALSE for a min level
- return TRUE for a max level

MIN-MAX HEAPS

function *verify_max() and verify_min*()

- begin at a max(min) node i
- follow the path of max(min) nodes from i to the root of the min-max heap
- search for the correct node in which to insert an item
- complexity: O($\log_2 n$), where n : number of element in a min-max heap

MIN-MAX HEAPS

```
void min_max_insert(element heap[], int *n, element item) {
   int parent;
   (*n)++;
   if (*n == MAX_SIZE) {
      fprintf(stderr,"the heap is full\n");
      exit(1);
   }
   parent = (*n)/2;
   if (!parent) heap[1] = item;
   else switch(level(parent)) {
      case FALSE: /* min level */
         if (item.key < heap[parent].key) {
            heap[*n] = heap[parent];
            verify_min(heap, parent, itme);
         }
         else
            verify_max(heap, *n, item);
         break;
      case TRUE: /* max level */
         if (item.key > heap[parent].key) {
            heap[*n] = heap[parent];
            verify_max(heap, parent, item);
         }
         else
            verify_min(heap, *n, item);
   }
}
```

procedure to insert into a min-max heap

MIN-MAX HEAPS

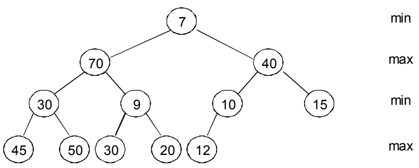
```
void verify_max(element heap[], int i, element item) {
/* follow the nodes from the max node i to the root
   and insert item into its proper place */
   int grandparent = i/4;
   while (grandparent)
      if (item.key > heap[grandparent].key) {
         heap[i] = heap[grandparent];
         i = grandparent;
         grandparent /= 4;
      }
      else
         break;
   heap[i] = item;
}
```

function verify_max()

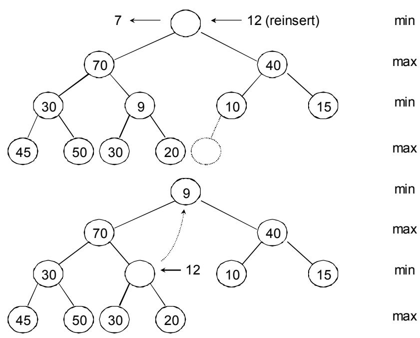
MIN-MAX HEAPS

Deleting from a min-max heap

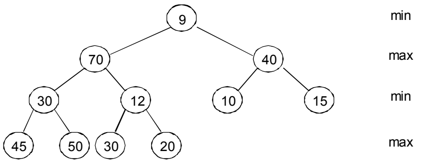
(delete the element with smallest key is in the root)



MIN-MAX HEAPS



MIN-MAX HEAPS



min-max-heap after deleting min

MIN-MAX HEAPS

reinsert an element *item* into a
min-max-heap, whose root is empty

1) the root has no children
- *item* is inserted into the root

MIN-MAX HEAPS

2) the root has at least one child
- let the node which has the smallest key be node *k* (one of its children or grandchildren)

a) *item.key* ≤ *heap*[*k*].*key*
- *item* is inserted into the root

b) *item.key* > *heap*[*k*].*key* and *k* is a child of the root
- element *heap*[*k*] is moved to the root
- *item* is inserted into node *k*

c) *item.key* > *heap*[*k*] and *k* is a grandchild of the root
- *parent*: the parent of k
- if *item.key* > *heap*[*parent*].*key* then interchange heap[parent] and *item*
- repeat the above process for sub-min-max heap

MIN-MAX HEAPS

function *min_child_grandchild*(*i*)

- determine the child or grandchild of the node i that has the smallest key
- return the address of the child

MIN-MAX HEAPS

```
element delete_min(element heap[], int *n) {
   int i, last, k, parent;
   element temp, x;
   if (!(*n)) {
      fprintf(stderr,"the heap is empty\n");
      heap[0].key = INT_MAX;
      return heap[0];
   }
   heap[0] = heap[1]; /* save the element */
   x = heap[(*n)--];
   for (i = 1, last = (*n)/2; i <= last;) {
       k = min_child_grandchild(i, *n);
       if (x.key <= heap[k].key) break;
       heap[i] = heap[k];
       if (k <= 2*i+1) {
          i = k;
          break;
       }
       parent = k/2;
       if (x.key > heap[parent].key)
          SWAP(heap[parent], x, temp);
       i = k;
   }
   heap[i] = x;
   return heap[0];
}
```

function to delete the element with the minimum key