Lecture 12.2

Topics:

1. Heaps – Complete Binary Trees
2. Heaps – Insertion, Removal, and Initialization
3. Example – Max Heap

---

1. Heaps – Complete Binary Trees

Definitions

Max/Min Tree

A max (min) tree is a tree in which the value in each node is greater (less) than or equal to those in its children (if any).

Complete Binary Tree

A complete binary tree is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.

Max/Min (Binary) Heap

A max heap (min heap) is a max (min) tree that is also a complete binary tree.

Figures 1 & 2 depict the max and min heaps. Note that they are drawn as trees, more specifically as a complete binary tree.

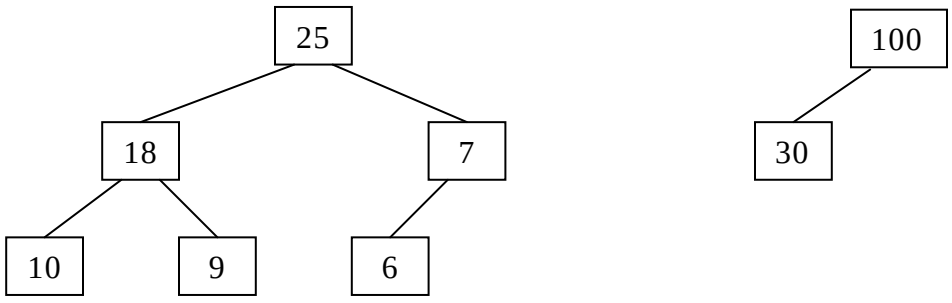

Figure 1 A max heap

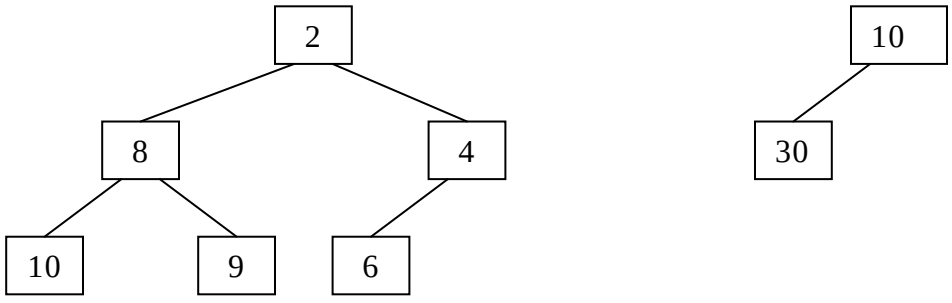

Figure 2 A min heap

A heap can be represented in a one-dimensional array that follows the property given below.

Property 1

Let `i` where `1 <= i <= n`, be the number assigned to an element of a complete binary tree. The following are true:

(i) If `i = 1`, then this element is the root of the binary tree. If `i > 1`, then the parent of this element has been assigned the number `L[i/2].`

(ii) If `2i > n`, then this element has no left child. Otherwise, its left child has been assigned the number `2i.`

(iii) If `2i + 1 > n`, then this element has no right child. Otherwise, its right child has been assigned the number `2i + 1.`

2. Heaps – Insertion, Removal, and Initialization

Heap as a data structure will have the two basic operations of insertion and removal. In a heap, the operations will be perform at the two "ends": removing the required value at the top of the heap (tree) plus performing a reconfiguration step, and inserting a new element by adding a new node plus performing a reconfiguration step.

Let's look at these operations.

2.1 Insertion

An insertion of an element into a (max) heap should retain its structure and property at the end. Let's consider a max heap with five elements below.

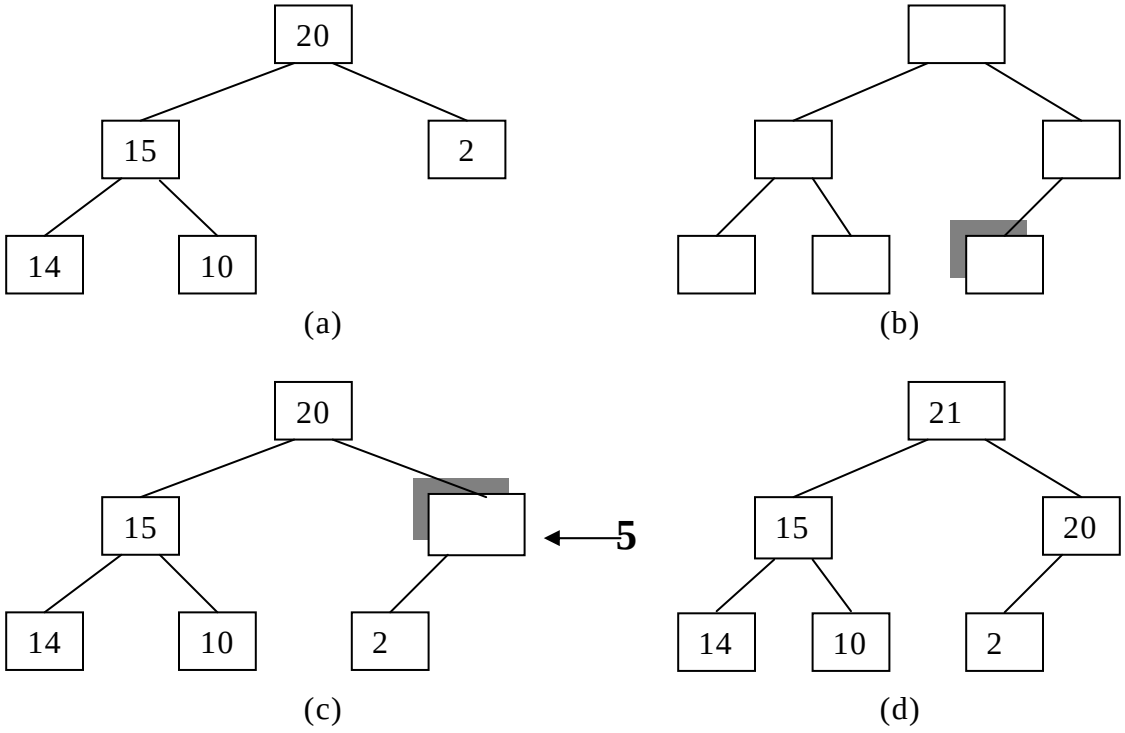

Figure 3 Insertion into a max heap

The insertion can be completed by first placing the new element into the new node. Secondly, one can bubble the new element up the tree (along the path from the new node to the root) until the new element has a parent whose priority is greater than or equal to that of the new element.

For examples, if the element has value 1, it may be inserted as the left child of 2. If instead the value of the new element is 5, the element cannot be inserted as the left child of 2. Therefore the element with value 2 is moved down to the left child. We then determine whether placing the 5 at the old

position of 2 results in a max heap. Since the parent element 20 is at least as large as the element 5, we can insert the new element of 5 at the position as shown in Figure 3(c).

If the new element is 21 and not 5, then the 2 is moved down to its left child position. The 21 cannot be inserted into the old position occupied by the 2; this is because the parent 20 of this position is smaller than 21. Hence the 20 is moved down to its right child, and the 21 is inserted at the root of the heap as shown in Figure 3(d).

2.2 Removal

The removal of an element in a max heap results to taking out the element at the root; that is `removeMax()`. Clearly, the structure needs to be rearranged to retain its max heap characteristics.

For example, a removal from the max heap of Figure 3(d) results in the deletion of the element 21 Figure 4(a). The structure is to be reheapified and the elements must also be rearranged. Clearly, the element 2 is not in the heap and it cannot be placed at the root. The larger of the two children of the root is moved into the root, thereby creating a vacancy at position 3 (verified with Property 1 above). Since this position has no children, the 2 may be put there as in Figure 4(c).

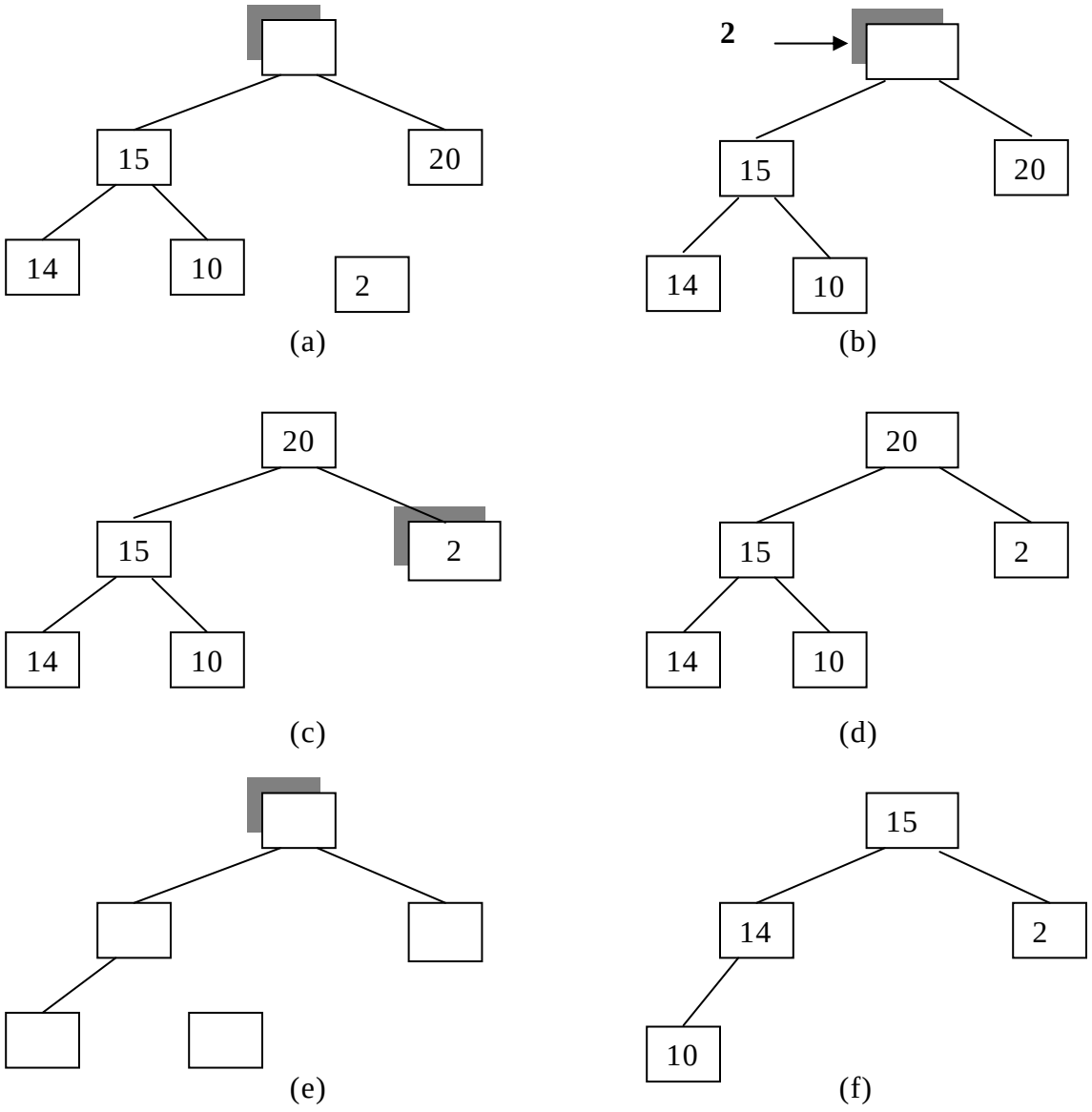

Figure 4 Removal of an element in a max heap

Now if we do another removal, the heap has the binary tree structure shown in Figure 4(e) where the shaded position will be filled with the elements of values from the remaining nodes. With the node at position 5 removed its value 10 cannot be put into the root. The larger of the two children of the root (15 and 2) is moved into the root. Keep doing this until the final structure to have the values as shown in Figure 4(f).

1.3 Initialization

We need to initialize the heap that contains `n > 0` elements. We can construct this initial nonempty heap by performing **`n`** `insert()`'s into an initially empty heap.

Let's start with an array **`a`** of n elements. Assume that the array and size are given with the following values:

```
int a[] = { 20, 12, 35, 15, 10, 80, 30, 17, 2, 1 };
```

How should we rearrange the values?

The process begins with the last element that has a child (i.e., the element value 10). This element is at position i = L[n / 2] of the array (i.e. at position 5). If the subtree that rooted at this position is a max heap, then no adjustment is needed. If the subtree is not a max heap , then we adjust the subtree so that it is a max heap. We then examine the subtree whose root is at (i - 1), then (i - 2), and so on until we have examined the root of the entire binary tree, which is at position 1.

Let's apply this process on the binary tree of Figure 4(a).

(1) Initially, i = 5
(2) The subtree with root at i is a max heap because element 10 is greater element 1.
(3) Next, i = 4
(4) The subtree with root at i is not a max heap, as 15 < 17.
    Adjustment:
    - Interchanging 15 and 17 to get the tree of Figure 4(b).
(5) Next, i = 3
(6) The subtree with root at i is not a max heap, as 35 < 80.
    Adjustment:
    - Interchanging 35 and 80 to get the tree of Figure 4(c).
(7) Next, i = 2
(8) The subtree is not a max heap. This is caused by the value of 17 is greater than 12.
    Adjustment:
    - The steps need to produce a max heap for the subtree.
    - Moving 17 to the original position occupied by 12 to make 17 the root of the restructured subtree.
    - Comparing 12 with the larger of the two children of the node at position 4. As 12 < 15, then 15 is moved to position 4. The vacancy at position 8 has no children, so 12 is inserted at position 8. The subtree whose root is element 17 at position 2 is a max heap as shown in Figure 4(d).
(9) Next, i = 1
(10) The subtrees at positions 2 and 3 are max heaps at this point, but the tree (with root at position) is not a max heap. This is because 20 < max{17, 80}. So 80 must be the root of the tree.
    Adjustment:
    - Moving 80 to the original position occupied by 20 to make 80 the root of the whole tree. This would leave a vacancy at position 3.

- With 20 < max{35, 30}, moving 35 to position 3. This will leave a vacancy at position 6 that has no children. So 20 can occupy position 6 as shown in Figure 4(e).

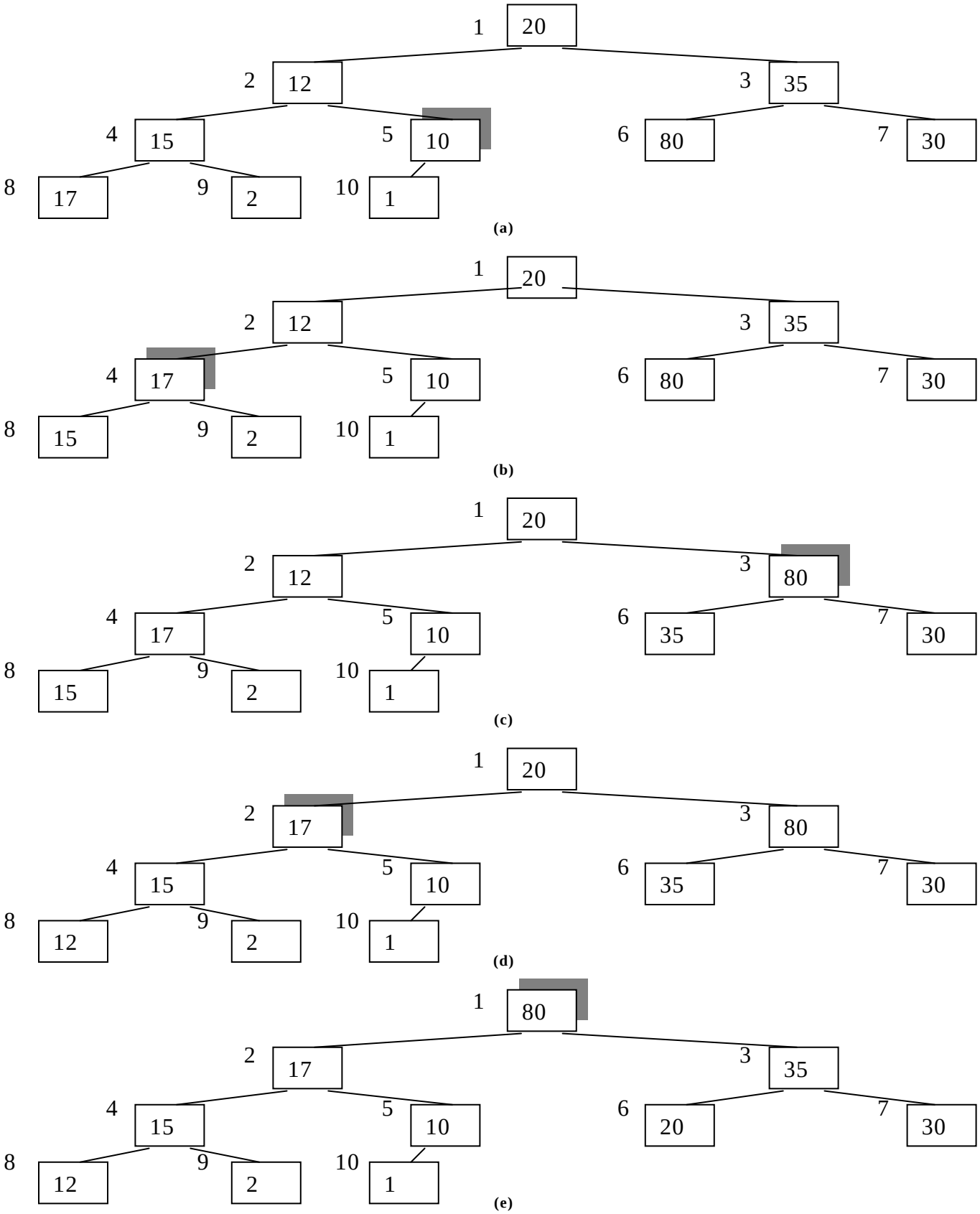

Figure 4 Heapification

3. Example

```
/**
 *Program:    cis27Lec1221Update.c
 *Discussion: Max Heap
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

const int MAXVALUE = 9999999;
struct HeapForm {
  int size;
  int current;
  int* data;
};

typedef struct HeapForm* Heap;

Heap initializeEmpty(int maxSize);
Heap initialize(int data[], int  arraySize, int dataSize);
void insert (Heap heap, int element);
int removeMax(Heap heap);
void printHeapAll(Heap heap);
void printHeapCurrent(Heap heap);

Heap initializeEmpty(int maxSize) {
  Heap heap;

  heap = (Heap) malloc(sizeof(struct HeapForm));
  assert(heap);

  heap->size = maxSize;
  heap->current = 0;

  heap->data = (int *) malloc(sizeof(int) * (heap->size + 1));
  assert(heap->data);

  heap->data[0] = MAXVALUE;

  return heap;
}

void insert(Heap heap, int element) {
  int* temp;
  int i;
  int leafPos;

  temp = heap->data;

  /*Increase the array size if necessary*/
  if (heap->current == (heap->size - 1)) {
    heap->data = (int *) malloc(sizeof(int) * 2 * heap->size);
    for (i = 0; i <= heap->current; i++) {
      heap->data[i] = temp[i];
    }

    free(temp);
  }

  /* Find place for element
```

```
    leafPos starts at new leaf and moves up tree */
  leafPos = ++heap->current;
  while (leafPos != 1 && (heap->data[leafPos / 2] < element)) {
    /* Cannot put element in heap->data[ leafPos ] */
    heap->data[leafPos] = heap->data [leafPos / 2]; /* Move
                                                       element down */
    leafPos /= 2; /* Move to parent */
  }
  heap->data[leafPos] = element;

  return;
}

void printHeapAll(Heap heap) {
  int i;

  for (i = 0; i < heap->size; i ++) {
    printf("Element at position %d : %d\n", i, heap->data[i]);
  }
  return;
}

void printHeapCurrent(Heap heap) {
  int i;

  for (i = 0; i < heap->current + 1; i ++) {
    printf("Element at position %d : %d\n", i, heap->data[i]);
  }
  return;
}

int main() {
  int ch;

  int data[] = {20, 12, 35, 15, 10, 80, 30, 17, 2, 1};

  int dataSize = 10;
  int arraySize = 12;

  Heap myHeap;

  myHeap = initializeEmpty(10);

  printf("\nValues initialized in heap :\n");
  printHeapAll(myHeap);

  printf("\nCurrent values in heap :\n");
  printHeapCurrent(myHeap);

  insert(myHeap, 90);

  printf("\nAfter inserting one value : ");
  printf("\n\tCurrent values in heap --\n");

  printHeapCurrent(myHeap);

  insert(myHeap, 8);

  printf("\nAfter inserting one value : ");
  printf("\n\tCurrent values in heap --\n");
```

```
  printHeapCurrent(myHeap);

  insert(myHeap, -8);

  printf("\nAfter inserting one value : ");
  printf("\n\tCurrent values in heap --\n");

  printHeapCurrent(myHeap);

  insert(myHeap, 10);

  printf("\nAfter inserting one value : ");
  printf("\n\tCurrent values in heap --\n");

  printHeapCurrent(myHeap);

  insert(myHeap, 80);

  printf("\nAfter inserting one value : ");
  printf("\n\tCurrent values in heap --\n");

  printHeapCurrent(myHeap);
  insert(myHeap, 100);

  printf("\nAfter inserting one value : ");
  printf("\n\tCurrent values in heap --\n");

  printHeapCurrent(myHeap);

  insert(myHeap, 45);

  printf("\nAfter inserting one value : ");
  printf("\n\tCurrent values in heap --\n");

  printHeapCurrent(myHeap);

  insert(myHeap, 180);

  printf("\nAfter inserting one value : ");
  printf("\n\tCurrent values in heap --\n");

  printHeapCurrent(myHeap);

  insert(myHeap, 200);

  printf("\nAfter inserting one value : ");
  printf("\n\tCurrent values in heap --\n");

  printHeapCurrent(myHeap);


  printf("\nEnter a character + ENTER to stop ...\n");
  scanf("%d", &ch);

  return 0;
}
```

OUTPUT

```
Values initialized in heap :
Element at position 0 : 9999999
```

```
Element at position 1 : -842150451
Element at position 2 : -842150451
Element at position 3 : -842150451
Element at position 4 : -842150451
Element at position 5 : -842150451
Element at position 6 : -842150451
Element at position 7 : -842150451
Element at position 8 : -842150451
Element at position 9 : -842150451

Current values in heap :
Element at position 0 : 9999999

After inserting one value :
        Current values in heap --
Element at position 0 : 9999999
Element at position 1 : 90

After inserting one value :
        Current values in heap --
Element at position 0 : 9999999
Element at position 1 : 90
Element at position 2 : 8

After inserting one value :
        Current values in heap --
Element at position 0 : 9999999
Element at position 1 : 90
Element at position 2 : 8
Element at position 3 : -8

After inserting one value :
        Current values in heap --
Element at position 0 : 9999999
Element at position 1 : 90
Element at position 2 : 10
Element at position 3 : -8
Element at position 4 : 8

After inserting one value :
        Current values in heap --
Element at position 0 : 9999999
Element at position 1 : 90
Element at position 2 : 80
Element at position 3 : -8
Element at position 4 : 8
Element at position 5 : 10

After inserting one value :
        Current values in heap --
Element at position 0 : 9999999
Element at position 1 : 100
Element at position 2 : 80
Element at position 3 : 90
Element at position 4 : 8
Element at position 5 : 10
Element at position 6 : -8
```

```
After inserting one value :
        Current values in heap --
Element at position 0 : 9999999
Element at position 1 : 100
Element at position 2 : 80
Element at position 3 : 90
Element at position 4 : 8
Element at position 5 : 10
Element at position 6 : -8
Element at position 7 : 45

After inserting one value :
        Current values in heap --
Element at position 0 : 9999999
Element at position 1 : 180
Element at position 2 : 100
Element at position 3 : 90
Element at position 4 : 80
Element at position 5 : 10
Element at position 6 : -8
Element at position 7 : 45
Element at position 8 : 8

After inserting one value :
        Current values in heap --
Element at position 0 : 9999999
Element at position 1 : 200
Element at position 2 : 180
Element at position 3 : 90
Element at position 4 : 100
Element at position 5 : 10
Element at position 6 : -8
Element at position 7 : 45
Element at position 8 : 8
Element at position 9 : 80

Enter a character + ENTER to stop ...
q
```