Lecture 13.1

Topics:

1. Heap Sort – A Max Heap Application
2. Recursive Print – Linked List and Binary Tree

---

1. Heap Sort – A Max Heap Application

Note that in all of the data structures, one of the concerns is how fast an operation can be performed on these data structures, e.g. sorting.

A detailed study on this topic can be found in a course on algorithm design and analysis, which is the sequel to learning about data structures – In general, the two bounds are $O(n^2)$ and $O(\log_2 n)$.

Other concerns would pertain to different aspect of constructing a data structure.

They are construction complexity (representation and storage of the data), computation/access complexity (would be a consequence for fast operation and memory requirement), application specifics, etc. However, selecting a data structure for data storage and operations would almost always allow better computational estimate and provide predictable results (compared to just the good old arrays!).

In this latest data structure called heap, there is a preferred selection of array in itself. This leads to a natural application that is always important – sorting problem.

1.1 Heap Sort

Clearly, data can be stored in an array or a linked list. Sorting a linked list is a bit more demanding, while sorting an array would provide some insight to the process.

Let's focus on sorting arrays here.

Clearly, “Bubble sort” should only be the term to remind everyone with an academic-only representation and "comically-worth" mentioned as an existing algorithm. There are other sorting algorithms that should be considered, e.g., Quicksort, Merge sort, etc.

Let's consider a new sorting technique call “Heap sor”.

From the name, it is obvious that a heap sort would also involve a heap structure. Note that a heap sort takes about $O(n\log_2 n)$ time (with respect to some operation unit).

Let's work through the details of the sort graphically.

1.2 Heap Sort -- Process

Let's use the result from previous lectures where the final heap is given in **Figure 1** below.

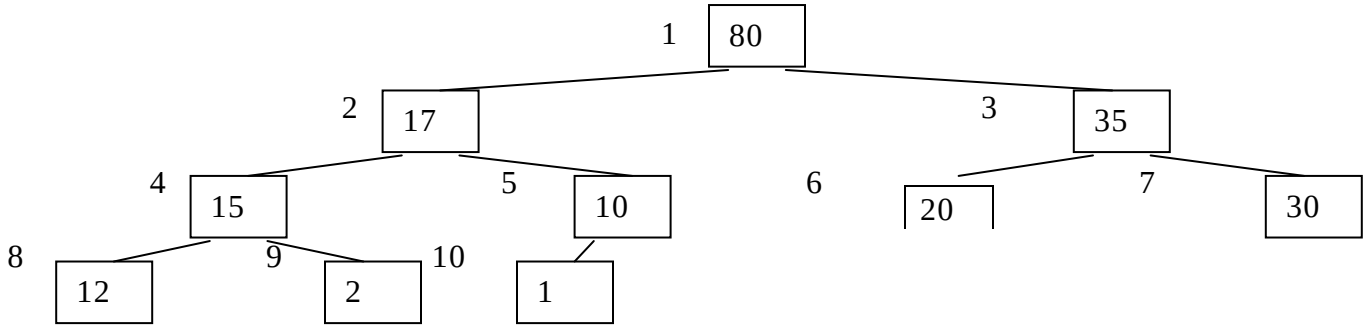

Figure 1 A heap structure

Each time an element (that means the root) is removed from the heap, it will be stored at an assigned location, which allows the remaining structure to be heapified. The steps are then repeated until all elements are removed from the heap and stored back to the same array.

In **Figure 2** below, the shaded squared are the roots from each of the iterations. They are stored orderly and properly after being removed from the (current) heap so that a partially sorted sequence and a heap structure are maintained at all time. Note that only a few iterations are performed here.

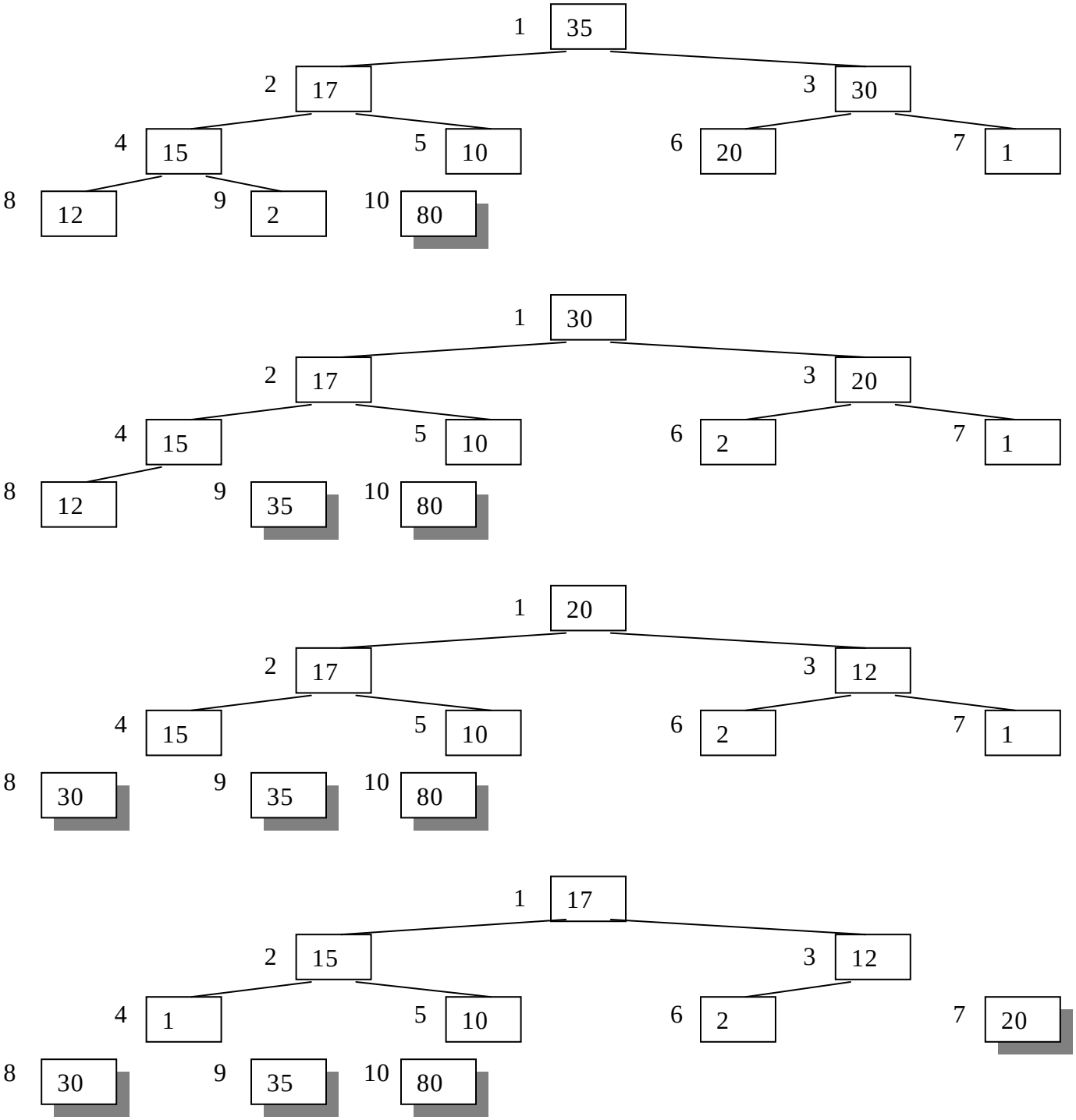

Figure 2 Removal and reheapification

2. Recursive Print -- Linked List & Binary Tree

Several methods to display data in linked lists and binary trees are considered here.

2.1 Linked List

Consider a general linked list of integers given in Example 1 below.

Example 1

```
typedef struct IntNode* IntNodePtr;
typedef struct IntNode* IntList;

struct IntNode {
  int iValue;
  struct IntNode* next;
};

void printIntList(IntList listHead); // function prototype

void printIntList(IntList listHead) { // function definition
  while (listHead != 0) {
    printf("\n%d", listHead->iValue);

    listHead = listHead->next;
  }

  return;
}
```

The code in `printIntList()` will print node values forward, that means from the first node to the last node. Of course, one may want to print **these values backward from the last node to the first node**. How would one do it?

An easy approach is to use recursive calls, where the addresses will be stored by the system automatically each time the call is made while searching for the base case to print.

Consider the following code fragment.

```
void printBackWard(IntList listHead) {
  if (listHead != 0) {
    printBackWard(listHead->next);

    printf("\n%d", listHead->iValue);
  }

  return;
}
```

What about a non-recursive approach?

Clearly, the key is to be able to get the addresses of the nodes in correct order and then to print out the node values one at a time. To do this, one may need to combine with a stack to store/push the addresses and restore/pop them out appropriately. The following code fragment would perform the correct steps.

```
struct IntListStack {
  int stackSize;
  int topOfStack;
  int* stackArray;
};
typedef struct IntListStack WorkingStack;

void printBackWardRecursive(IntList listHead) {
```

```
  WorkingStack tempStack;
  IntList tempNodeAddr;

  while (listHead != 0) {
    push(listHead, tempStack);
    listHead = listHead->next;
  }

  while (!isEmptyStack(tempStack)) {
    tempNodeAddr = top(tempStack);
    printf("\n%d", tempNodeAddr->iValue);
    pop(tempStack);
  }

  return;
}
```

2.2 Binary Trees – Non-recursive Inorder Traversal

Let's consider the display of `InOrder` traversal, where the order is **left-node-right**. How would it be done?

A general algorithm is described in the following pseudocode.

```
(i)  current = root (That means to start at the root)
(ii) while (current is not NULL and tempStack is not empty)
     {
          if (current is not NULL)
          {
               push current onto tempStack
               current = current->left
          }
          else
          {
               pop tempStack into current
               visit current (that means the node)
               current = current->right
          }
     }
```

Consider the following code,

```
void displayInOrderNonRecur(TreePtr myTree) {
  PointerStack tempStack;
  /*Initializing tempStack before using it*/

  while ((myTree != 0) || (!isEmpty(tempStack))) {
    if (myTree != 0) {
      push(myTree, tempStack);
      myTree = myTree->left;
    } else {
      myTree = top(tempStack);
      pop(tempStack);
      printf("\n%d", myTree->iValue);
      myTree = myTree->right;
    }
  }
  return;
}
```