Heap

Data Structures & Algorithms

CONTENT

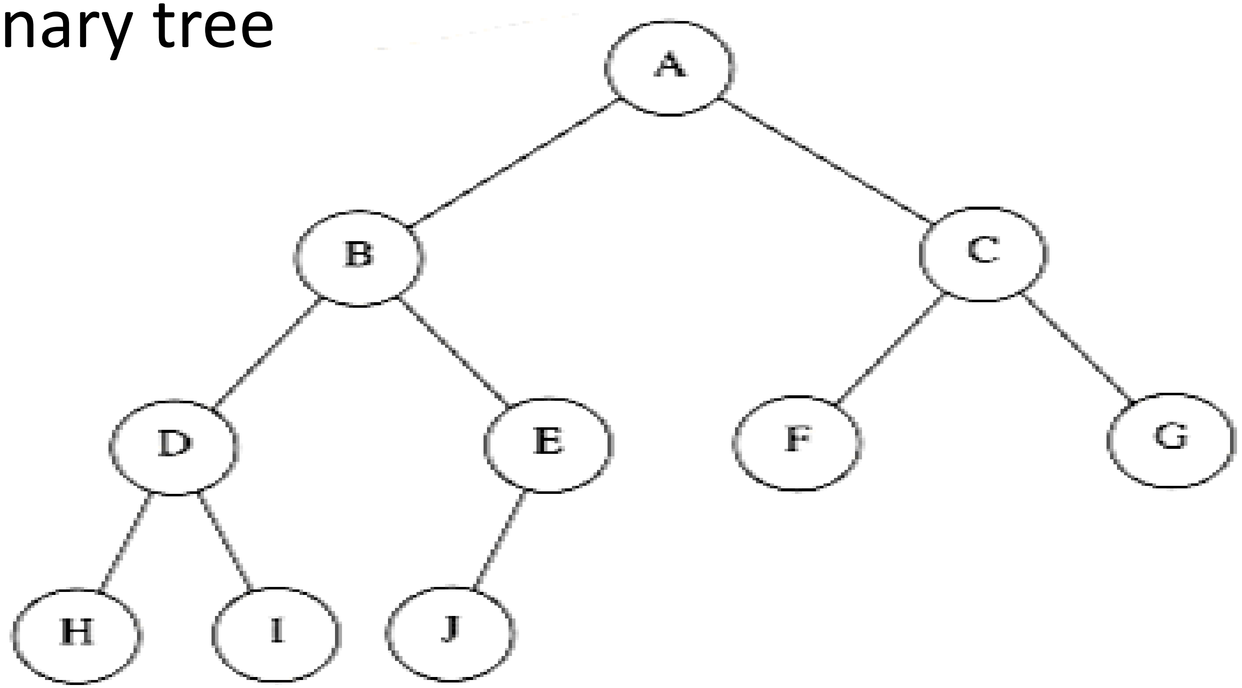
- Introduction
- Basic Operations
- Heap Sort

CONTENT

- **Introduction**
- Basic Operations
- Heap Sort

- **Heap**
  - is an application of complete binary tree (also called **priority queue**)
- **Definition**
  - max/min tree
    - a tree in which the key value in each node is no smaller/greater than the key values in its children (if any)
  - max/min heap
    - a max/min complete binary tree
  - Parent A[i] (for array A[1..n], A[1] is the root)
    - Left child: A[2i]
    - Right child: A[2i + 1]

- **Examples**
  - A complete binary tree

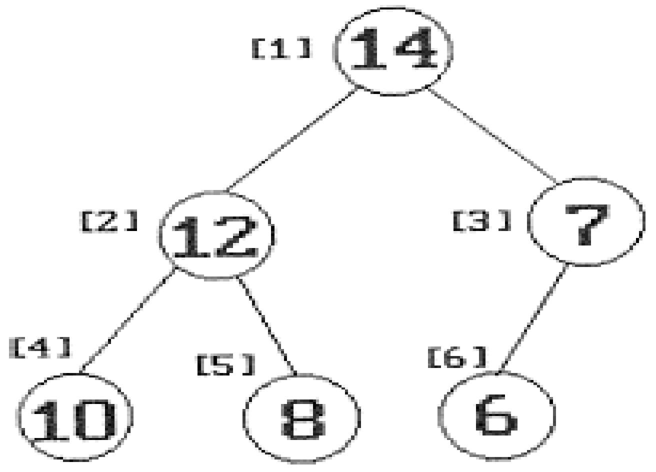


  - Array implementation of the tree

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

- **Heap Representation**
  - Since heaps are complete trees, we may use an array representation

```
#define MAX_ELEMENTS 100
typedef struct {
   int key;
   /* other fields */
} element;
element heap[MAX_ELEMENTS];
int n = 0;
```

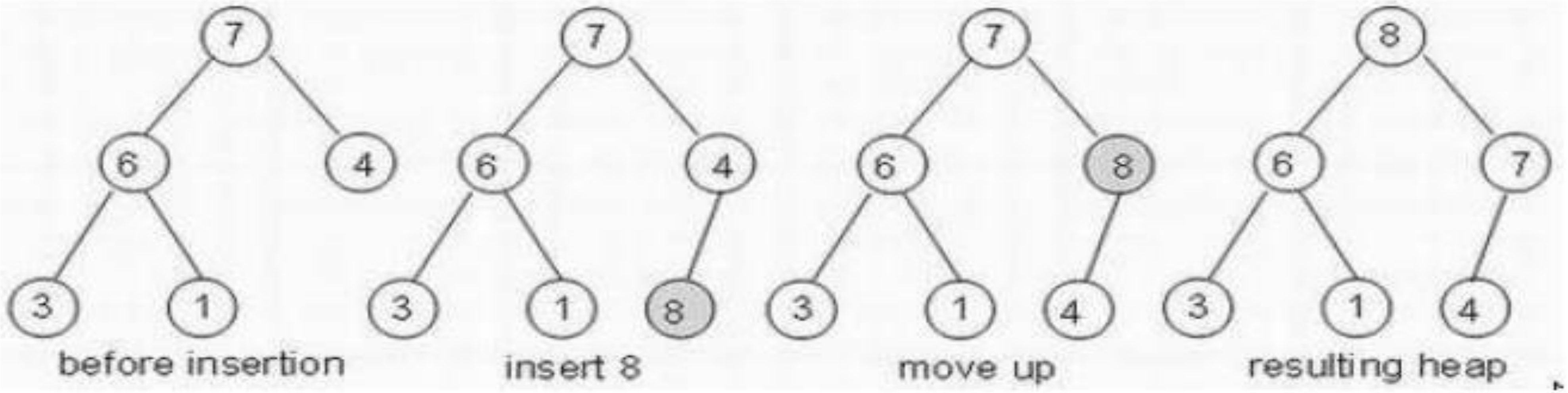


	14	12	7	10	8	6
0	1	2	3	4	5	6

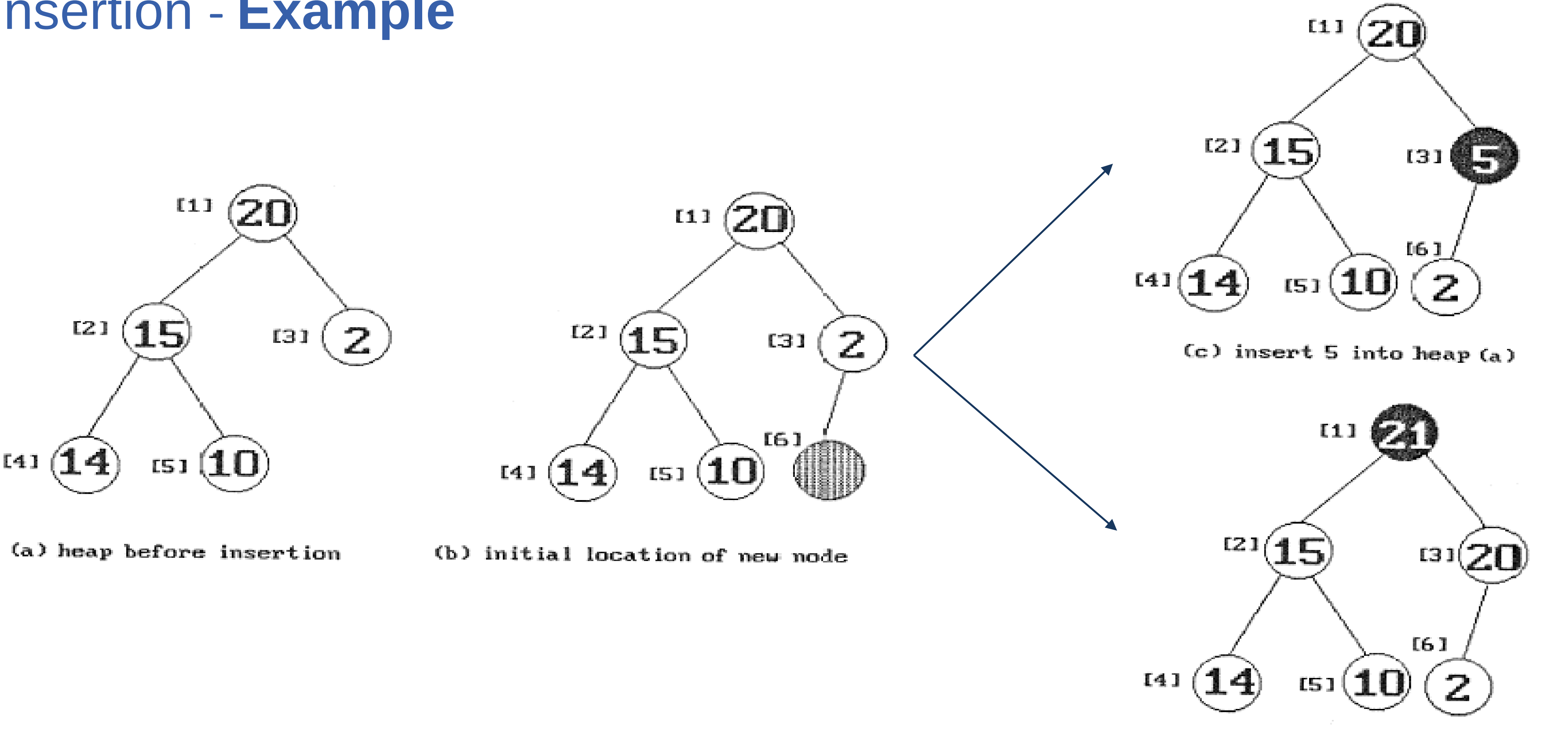
CONTENT

- Introduction
- **Basic Operations**
- Heap Sort

- **Insertion**
  - Find a proper place for the new element in the array implementation
  - The parent of node **i** is located at **i/2**
    - Step 1: Put the new element at the last entry of the array
    - Step 2: Exchange the new element with its parent,
      if the new element is greater
    - Step 3: Repeat Step 2 until no more exchange is necessary



- Insertion - **Example**



(a) heap before insertion

(b) initial location of new node

(c) insert 5 into heap (a)

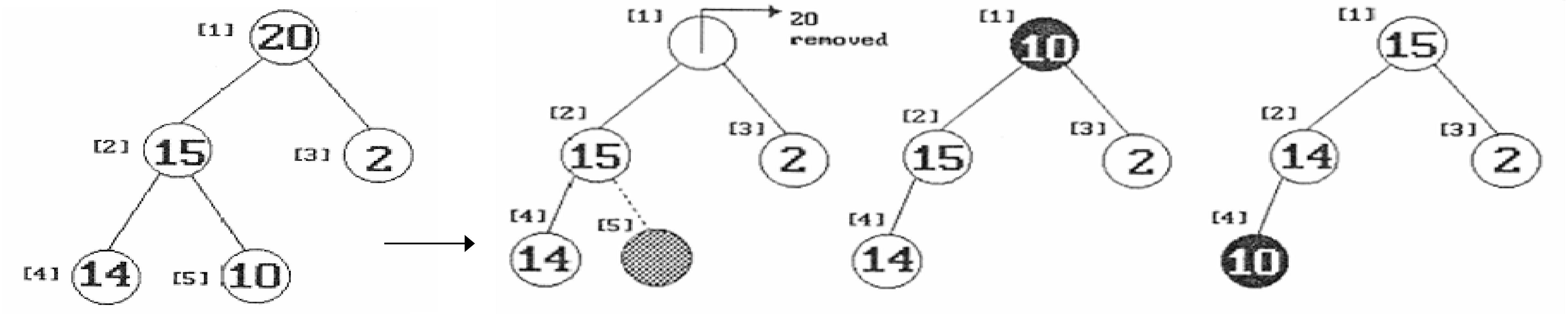
(d) insert 21 into heap (a)

- **Insertion**

```
void  insertMaxHeap(element item, int *n){
        int i;
        if (HEAP_FULL(*n))
                fprintf(stderr, “the heap is full.\n); exit(1);
        i = ++(*n);
        while ((i!=1) && (item.key>heap[i/2].key))
                heap[i] = heap[i/2];  i /= 2;
        heap[i] = item;
}
```

The height of n node heap = $\log_2(n+1)$
Time complexity = O (height) = O ($\log_2 n$)

- **Delete** - Delete the max (root) from a max heap
  - Step 1: Remove the root
  - Step 2: Replace the last element to the root
  - Step 3: Reestablish the heap (go down from root to leaf, exchange 2 elements as necessary)



(a) heap structure (b) 10 inserted at the root (c) final heap

- **Delete** - Delete the max (root) from a max heap

```
element   deleteMaxHeap(int *n){
   int parent, child;  element item, temp;
   if (HEAP_EMPTY(*n)) {
                        fprintf(stderr, "The heap is empty\n");        exit(1);
   }
   item = heap[1];                        /* save value of the element with the highest key */
   temp = heap[(*n)--];                   /* use last element in heap to adjust heap */
   parent = 1;  child = 2;
   while (child <= *n) {                  /* find the larger child of the current parent */
            if ((child < *n) && (heap[child].key<heap[child+1].key))          child++;
            if (temp.key >= heap[child].key)      break;
            heap[parent] = heap[child];           /* move to the next lower level */
            parent= child; child *= 2;
   }
   heap[parent] = temp;
   return  item;
}
```

CONTENT

- Introduction
- Basic Operations
- **Heap Sort**

- **Heap Sort**
  - Given *n* elements (in an array A[1..n]) to be sorted
  - Recall: max heap
    - An array is represented by a complete binary tree, in which the key value in each node is no smaller than the key values in its children (if any)
    - A[1] is the root (suppose the first element of the array is A[1])
    - A[i] is parent, so A[2i] is the left child and A[2i+1] is the right child (if A[0] is the root, so A[2i+1] and A[2i+2]) respectively
  - O(*n* log *n*) time

(1). Build a max heap

– Use function *adjust*(A, i, n)
  - both the left and the right sub-trees of A[i] are already max heaps
  - the element A[i] will be moved to one of its descendant so that the sub-tree rooted at A[i] becomes a max heap
– Function *adjust*() is invoked for the sub-trees rooted at A[n/2],A[n/2-1], …, A[1] in that order (i.e. all the non-leaf nodes)

(2). Sort by using the heap

a. A[1..n] is a heap, exchange A[1] & A[n] -> A[n] is rightly position
b. Rebuild a max heap for A[1..n-1]. Repeat steps (a) & (b) until array has only one element

- **Example**

6 5 3 1 8 7 2 4

- **(1).Build a max heap - Use function *adjust*(A, i, n)**
  - both the left and the right sub-trees of A[i] are already max heaps
  - the element A[i] will be moved to one of its descendant so that the sub-tree rooted at A[i] becomes a max heap

```
void adjust(int list[], int root, int n) {
        int child, rootkey; int temp;
        temp = list[root]; rootkey = list[root].key; child = 2*root;
        while (child <= n) {
                if ((child<n) && (list[child].key<list[child+1].key))     child++;
                if (rootkey > list[child].key)     break;
                else {    list[child/2] = list[child];        child *= 2; }
        }
        list[child/2] = temp;
}
```

- **(2).Sort by using heap**

```
void heap_sort(int list[], int n) {
    /* Initially data is in list[1.. n] */
    int i, j;
    /* build a max heap */
    for (i = n/2; i > 0; i--)   adjust(list, i, n);
    /* at this point we have a max heap */
    for (i = n-1; i > 0; i--) {
            SWAP(list[1], list[i+1]);  /* swap the root & element at pos. i+1*/
            adjust(list, 1, i);          /* rebuild list from elements 1 to i */
    }
}
```

- How to sort the list in descending order?

SUMMARY

- Introduction
- Basic Operations
- Heap Sort

Enjoy the Course…!