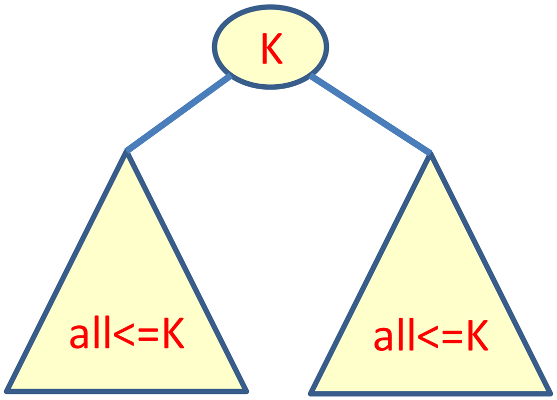
Chapter 8 - Heaps

- Binary Heap. Min-heap. Max-heap.
- Efficient implementation of heap ADT: use of array
- Basic heap algorithms: ReheapUp, ReheapDown, Insert Heap, Delete Heap, Built Heap
- d-heaps
- Heap Applications:
  - Select Algorithm
  - Priority Queues
  - Heap sort
- Advanced implementations of heaps: use of pointers
  - Leftist heap
  - Skew heap
  - Binomial queues

Binary Heaps

DEFINITION: A max-heap is a binary tree structure with the following properties:

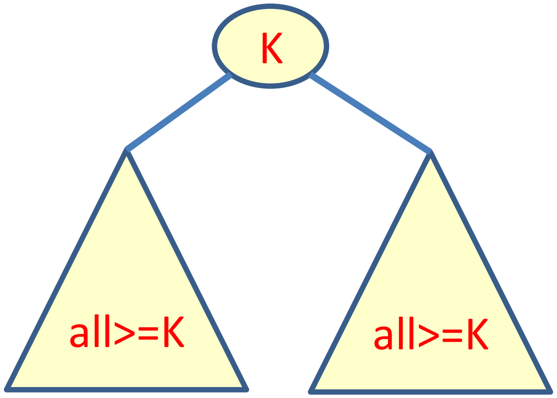
- The tree is complete or nearly complete.
- The key value of each node is greater than or equal to the key value



max-heap

DEFINITION: A min-heap is a binary tree structure with the following properties:

- The tree is complete or nearly complete.
- The key value of each node is less than or equal to the key value in each of its descendents.



min-heap

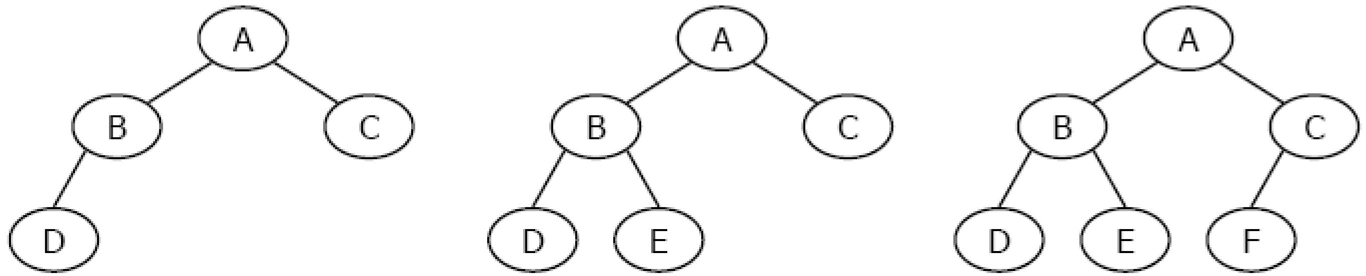
Properties of Binary Heaps

- Structure property of heaps
- Key value order of heaps

Properties of Binary Heaps

Structure property of heaps:

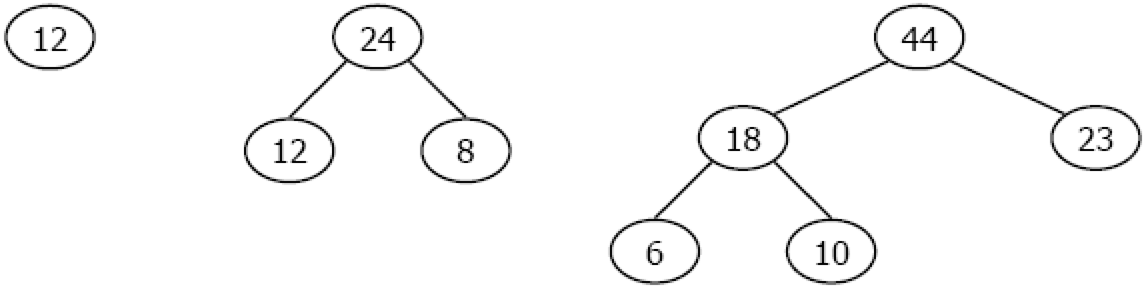
- A complete or nearly complete binary tree.
- If the height is h, the number of nodes n is between 2^{h-1} and $(2^h - 1)$
- Complete tree: $n = 2^h - 1$ when last level is full.
- Nearly complete: All nodes in the last level are on the left.



- $h = \lfloor \log_2 n \rfloor + 1$
- Can be represented in an array and no pointers are necessary.

Properties of Binary Heaps

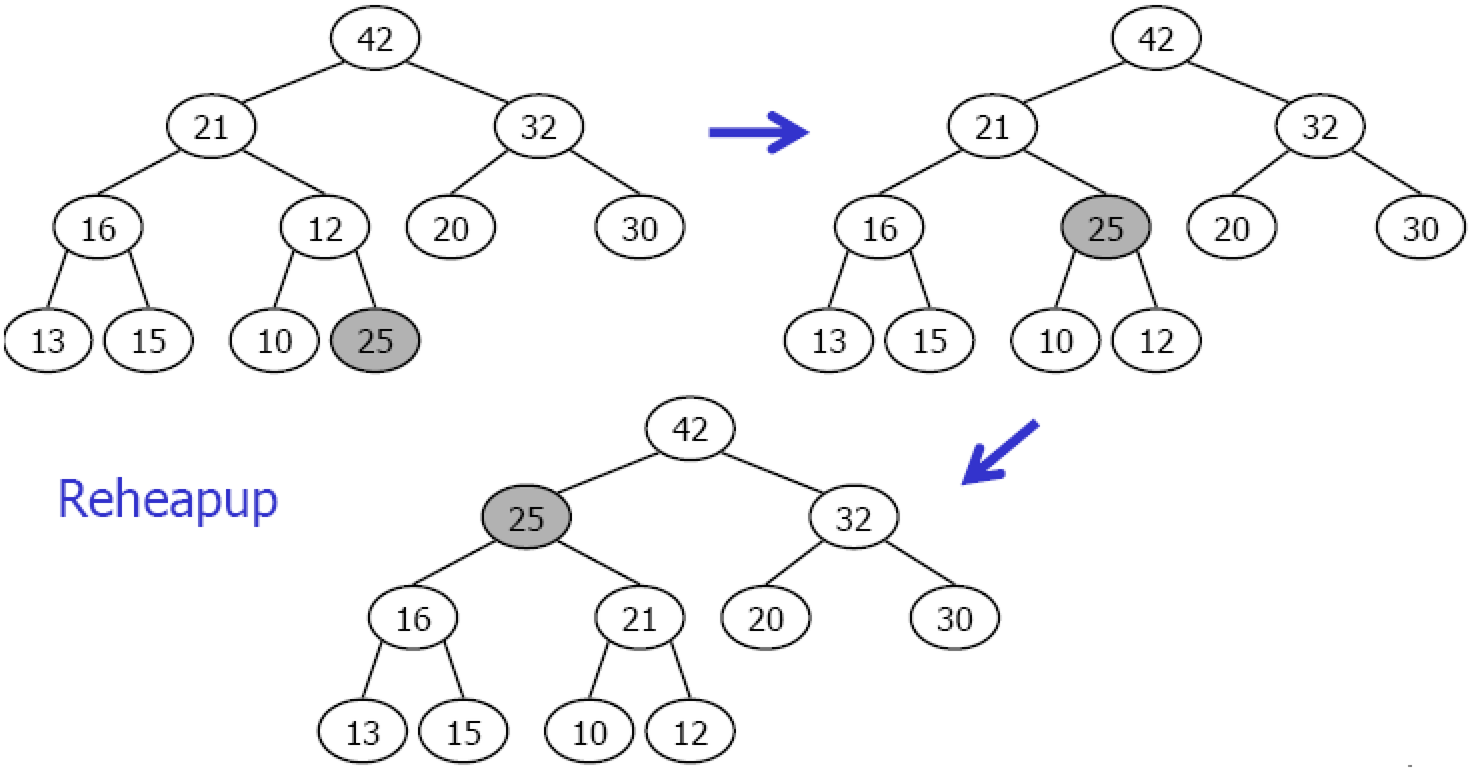
Key value order of max-heap:



(max-heap is often called as *heap*)

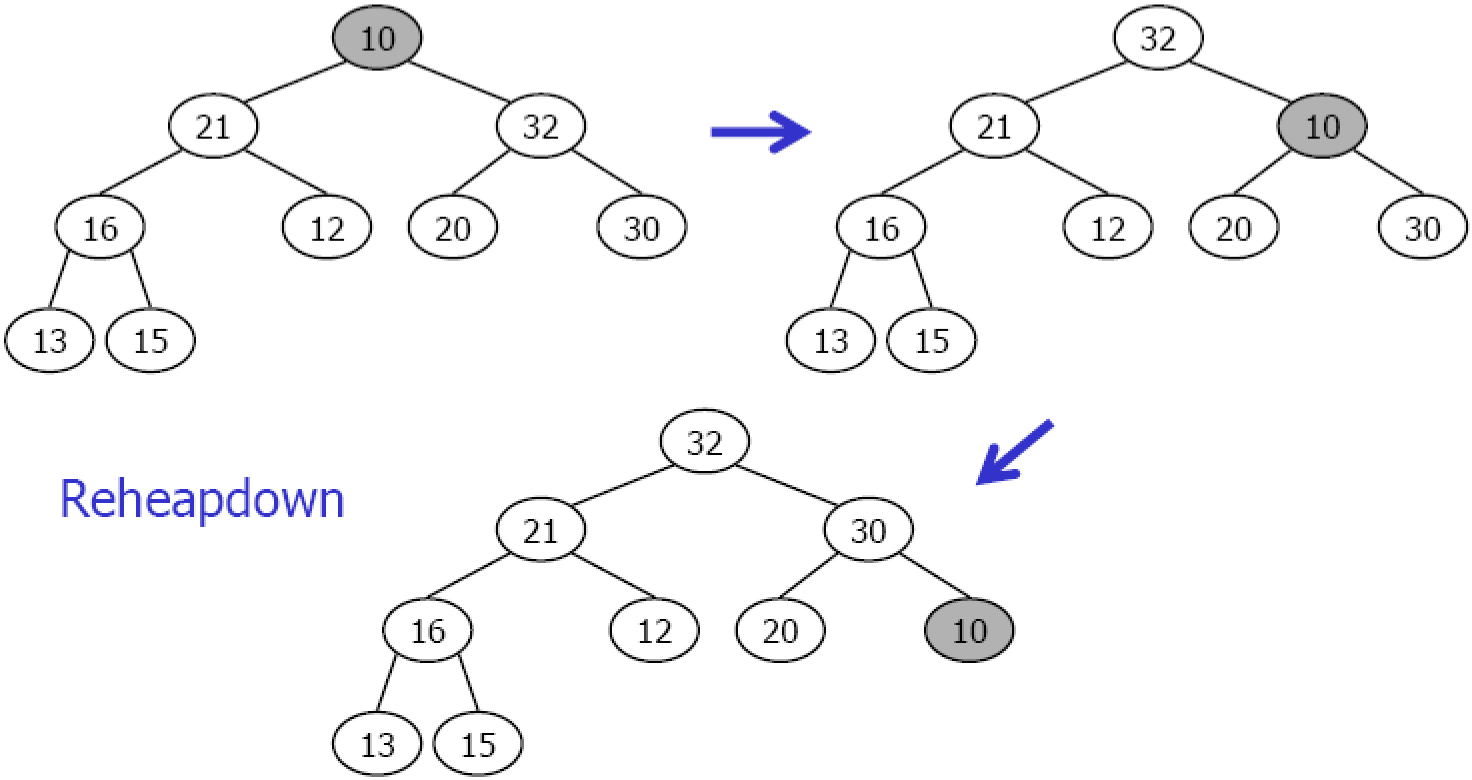
Basic heap algorithms

ReheapUp: repairs a "broken" heap by floating the last element up the tree until it is in its correct location.



Basic heap algorithms

ReheapDown: repairs a "broken" heap by pushing the root of the subtree down until it is in its correct location.



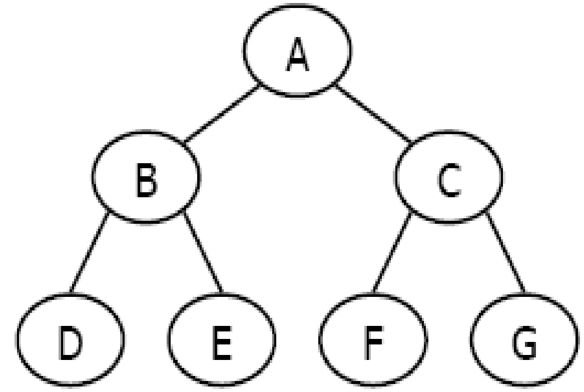
Contiguous Implementation of Heaps

Heap

data <Array of <DataType> >

count <int> *//number of elements in heap*

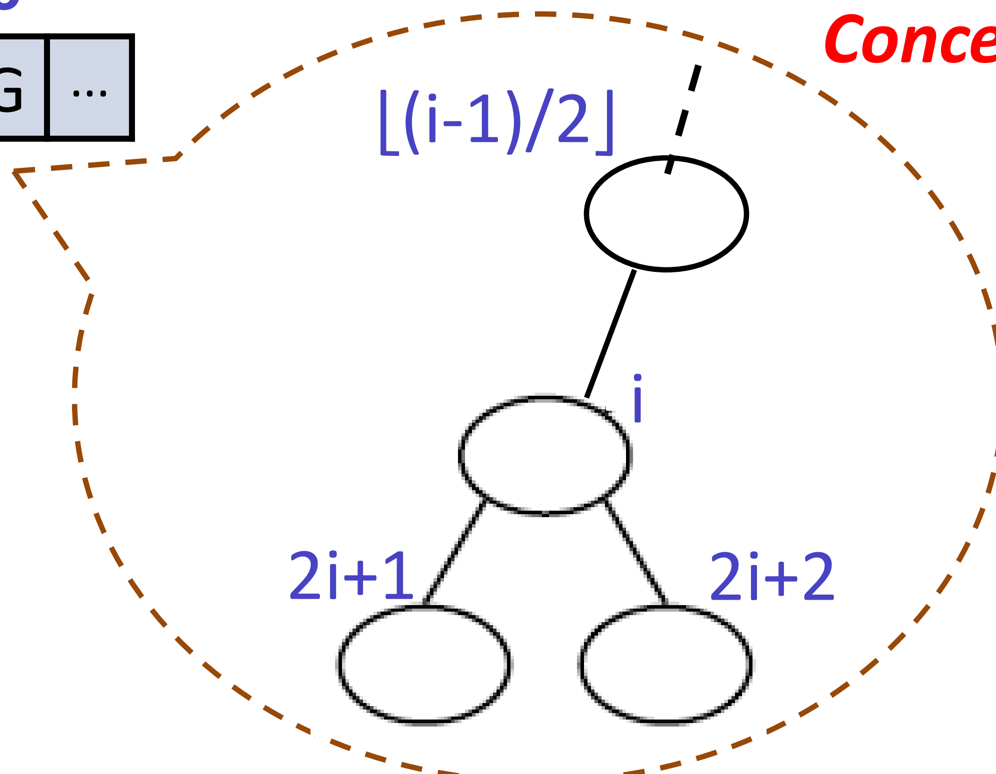
End Heap



0	1	2	3	4	5	6	
A	B	C	D	E	F	G	…

Physical

Conceptual



ReheapUp

Algorithm **ReheapUp** (val position <int>)

Reestablishes heap by moving data in position up to its correct location.

Pre All data in the heap above this position satisfy key value order of a heap, except the data in position.

Post Data in position has been moved up to its correct location.

Uses Recursive function ReheapUp.

```
1. if (position <> 0)            // the parent of position exists.
   1. parent = (position-1)/2
   2. if (data[position].key > data[parent].key)
      1. swap(position, parent) // swap data at position with data at parent.
      2. ReheapUp(parent)
2. return
End ReheapUp
```

ReheapDown

Algorithm **ReheapDown** (val position <int>, val lastPosition <int>)

Reestablishes heap by moving data in position down to its correct location.

Pre All data in the subtree of position satisfy key value order of a heap, except the data in position.

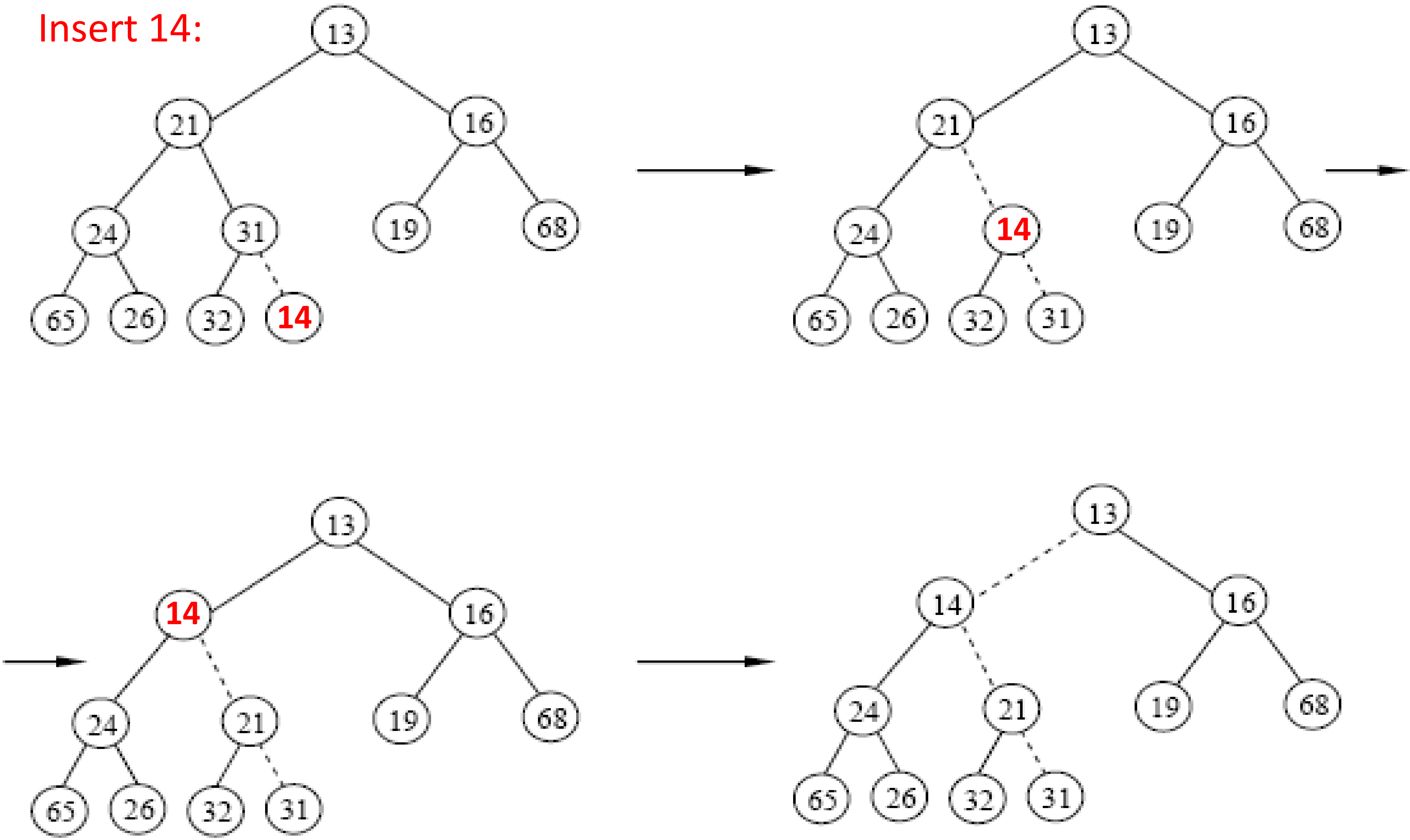
Post Data in position has been moved down to its correct location.

Uses Recursive function ReheapDown.

```
1. leftChild = position *2 + 1
2. rightChild = position *2 + 2
3. if ( leftChild  <= lastPosition )       // the left child of position exists.
   1. if ( rightChild  <= lastPosition)  AND  ( data[rightChild].key  >  data[leftChild].key )
         1. child = rightChild
   2. else
         1. child = leftChild     // choose larger child to compare with data in position
   3. if ( data[child].key  >  data[position].key )
         1. swap(child, position) // swap data at position with data at child.
         2. ReheapDown(child, lastPosition)
4. return
End ReheapDown
```

Insert new element into min-heap

Insert 14:



The new element is put to the last position, and ReheapUp is called for that position.

<ErrorCode> **InsertHeap** (val DataIn <DataType>) *// Recursive version.*

Inserts new data into the min-heap.

Post DataIn has been inserted into the heap and the heap order property is maintained.

Return *overflow* or *success*

Uses recursive function ReheapUp.

1. **if** (heap is full)
    1. return *overflow*

2. **else**
    1. data[count] = DataIn
    2. ReheapUp(count)
    3. count = count + 1
    4. return *success*

End InsertHeap

<ErrorCode> **InsertHeap** (val DataIn <DataType>) *// Iterative version*

Inserts new data into the min-heap.

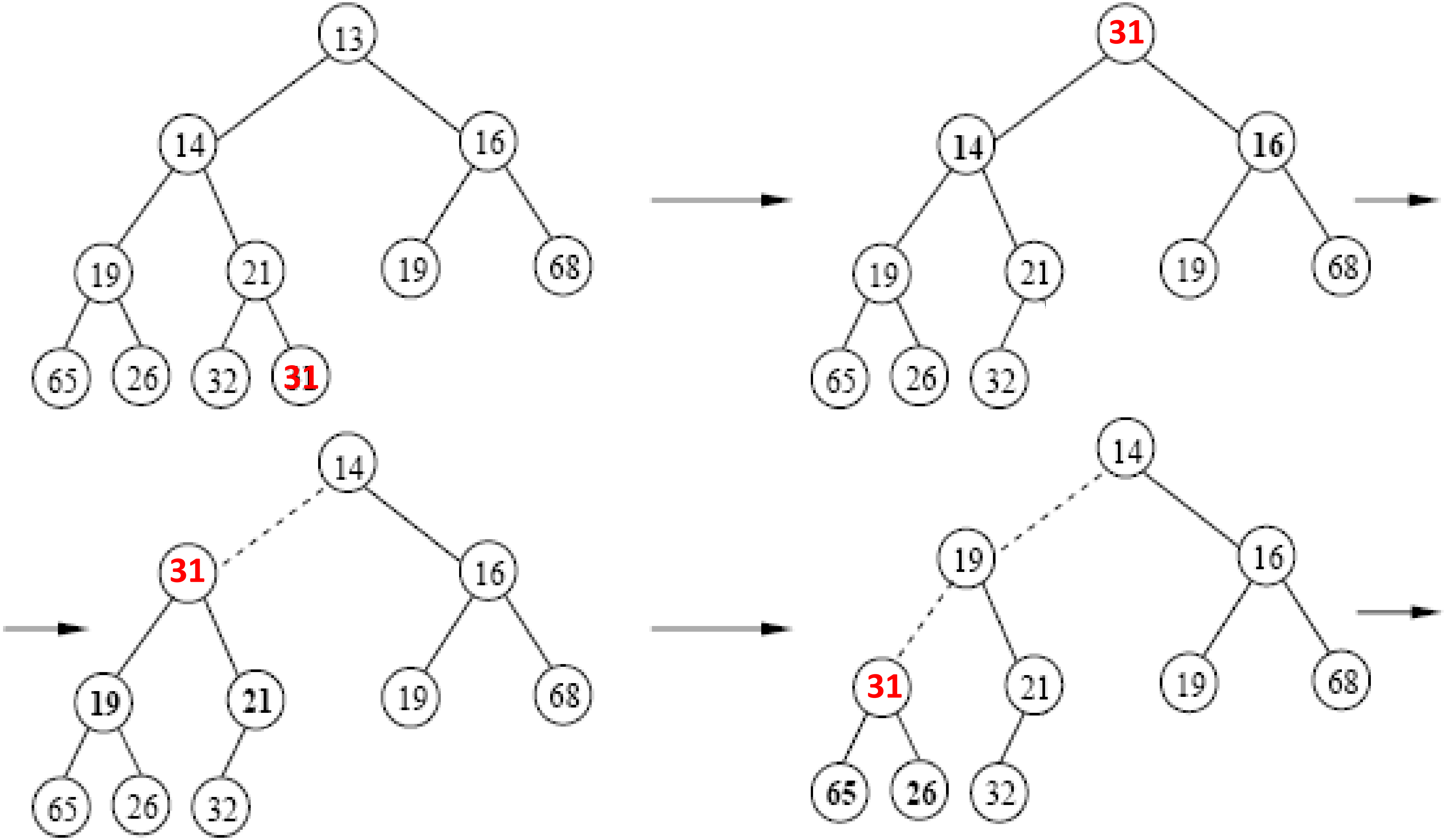
Post DataIn has been inserted into the heap and the heap order property is maintained.

Return *overflow* or *success*

1. **if** (heap is full)
    1. return *overflow*
2. **else**
    1. current_position = count - 1
    2. **loop** (the parent of the element at the current_position is exists) AND (parent.key > DataIn .key)
        1. data[current_position] = parent
        2. current_position = position of parent
    3. data[current_position] = DataIn
    4. count = count + 1
    5. return *success*

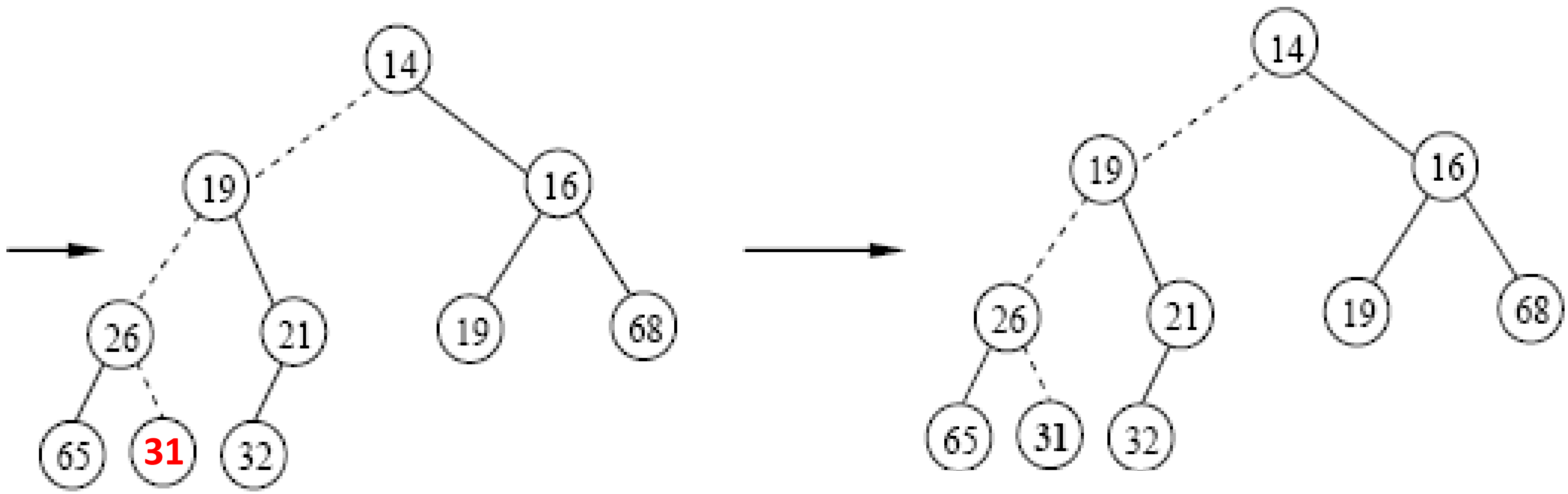
End InsertHeap

Delete minimum element from min-heap



The element in the last position is put to the position of the root, and ReheapDown is called for that position.

Delete minimum element from min-heap



The element in the last position is put to the position of the root, and ReheapDown is called for that position.

<ErrorCode> **DeleteHeap** (ref MinData <DataType>) *// Recursive version*

Removes the minimum element from the min-heap.

Post MinData receives the minimum data in the heap and this data has been removed. The heap has been rearranged.

Return *underflow* or *success*

Uses recursive function ReheapDown.

1. **if** (heap is empty)
    1. return *underflow*
2. **else**
    1. MinData = Data[0]
    2. Data[0] = Data[count -1]
    3. count = count - 1
    4. ReheapDown(0, count -1)
    5. return *success*

End DeleteHeap

<ErrorCode> **DeleteHeap** (ref MinData <DataType>) *// Iterative version*

Removes the minimum element from the min-heap.

Post MinData receives the minimum data in the heap and this data has been removed. The heap has been rearranged.

Return *underflow* or *success*

1. **if** (heap is empty)
   1. return *underflow*
2. **else**
   1. MinData = Data[0]
   2. lastElement = Data[count – 1] *// The number of elements in the*
      // heap is decreased so the last
      // element must be moved
      // somewhere in the heap.

```
// DeleteHeap(cont.)                                  // Iterative version
3.  current_position = 0
4.  continue = TRUE
5.  loop (the  element at the current_position has children) AND
                        (continue = TRUE)
    1.  Let child is the smaller of two children
    2.  if (lastElement.key  > child.key )
        1.  Data[current_position] = child
        2.  current_position = current_position of child
    3.  else
        1.  continue = FALSE
6.  Data[current_position] = lastElement
7.  count = count - 1
8.  return success
End DeleteHeap
```

Build heap

<ErrorCode> **BuildHeap** (val listOfData <List>)

Builds a heap from data from listOfData.

Pre listOfData contains data need to be inserted into an empty heap.

Post Heap has been built.

Return *overflow* or *success*

Uses Recursive function ReheapUp.

1. count = 0
2. **loop** (heap is not full) AND (more data in listOfData)
    1. listOfData.Retrieve(count, newData)
    2. data[count] = newData
    3. ReheapUp(count)
    4. count = count + 1
3. **if** (count < listOfData.Size())
    1. return *overflow*
4. **else**
    1. return *success*

End BuildHeap

Build heap

Algorithm **BuildHeap2** ()

Builds a heap from an array of random data.

Pre Array of count random data.

Post Array of data becames a heap.

Uses Recursive function ReheapDown.

1. position = count / 2 -1
2. **loop** (position >=0)
   1. ReheapDown(position, count-1)
   2. position = position - 1
3. return

End BuildHeap2

$O(n)$

$\frac{n}{2} \sim O(\log n) \approx O(n)$

Complexity of Binary Heap Operations

- ReheapUp: $O(\log_2 n)$
- ReheapDown: $O(\log_2 n)$
- BuildHeap: $O(n\log_2 n)$
- InsertHeap: $O(\log_2 n)$
- DeleteHeap: $O(\log_2 n)$

d-heaps

- d-heap is a simple generalization of a binary heap.
- In d-heap, all nodes have d children.
- d-heap improve the running time of InsertElement to $O(\log_d n)$.
- For large d, DeleteMin operation is more expensive: the minimum of d children must be found, which takes d-1 comparisons.
- The multiplications and divisions to find children and parents are now by d, which increases the running time. (If d=2, use of the bit shift is faster).
- d-heap is suitable for the applications where the number of Insertion is greater than the number of DeleteMin.

Heap Applications

- Select Algorithms.
- Priority Queues.
- Heap sort (*we will see in the Sorting Chapter*).

Select Algorithms

Determine the k^{th} largest element in an unsorted list

Algorithm 1a:

- Read the elements into an array, sort them.
- Return the appropriate element.

The running time of a simple sorting algorithm is $O(n^2)$

Select Algorithms

Determine the k^{th} largest element in an unsorted list

Algorithm 1b:

- Read k elements into an array, sort them.
- The smallest of these is in the k^{th} position.
- Process the remaining elements one by one.
- Compare the coming element with the k^{th} element in the array.
- If the coming element is large, the k^{th} element is removed, the new element is placed in the correct place.

The running time is $O(n^2)$

Select Algorithms

Determine the k^{th} largest element in an unsorted list

Algorithm 2a:

- Build a max-heap.
- Detele k-1 elements from the heap.
- The desired element will be at the top.

The running time is $O(n\log_2 n)$

Select Algorithms

Determine the k^{th} largest element in an unsorted list

Algorithm 2b:

- Build a min-heap of k elements.
- Process the remaining elements one by one.
- Compare the coming element with the minimum element in the heap (the element on the root of heap).
- If the coming element is large, the minimum element is removed, the new element is placed in the correct place (*reheapdown*).

The running time is $O(n\log_2 n)$

Priority Queue ADT

- Jobs are generally placed on a queue to wait for the services.
- In the multiuser environment, the operating system scheduler must decide which of several processes to run.
- Short jobs finish as fast as possible, so they should have precedence over other jobs.
- Otherwise, some jobs are still very important and should also have precedence.

These applications require a special kind of queue: a priority queue.

Priority Queue ADT

- Each element has a priority to be dequeued.
- Minimum value of key has highest priority order.

DEFINITION of Priority Queue ADT:

Elements are enqueued accordingly to their priorities.

Minimum element is dequeued first.

Basic Operations:

- *Create*
- *InsertElement*: Inserts new data to the position accordingly to its priority order in queue.
- *DeleteMin*: Removes the data with highest priority order.
- *RetrieveMin*: Retrieves the data with highest priority order.

Priority Queue ADT

Extended Operations:

- *Clear*
- *isEmpty*
- *isFull*
- *RetrieveMax*: Retrieves the data with lowest priority order.
- *IncreasePriority*
- *DecreasePriority*

Changes the priority of some data which has been inserted in queue.

- *DeleteElement*: Removes some data out of the queue.

Specifications for Priority Queue ADT

```
<ErrorCode> InsertElement (val DataIn <DataType>)
<ErrorCode> DeleteMin (ref MinData <DataType>)
<ErrorCode> RetrieveMin (ref MinData <DataType>)
<ErrorCode> RetrieveMax (ref MaxData <DataType>)
<ErrorCode> IncreasePriority (val position <int>,
                              val PriorityDelta <KeyType>)
<ErrorCode> DecreasePriority (val position <int>,
                              val PriorityDelta <KeyType>)
<ErrorCode> DeleteElement (val position <int>,
                           ref DataOut <DataType>)
<bool> isEmpty()
<bool> isFull()
<void> clear()
```

Implementations of Priority Queue

- **Use linked list:**
  - Simple linked list:
    - Insertion performs at the front, requires O(1).
    - DeleteMin requires O(n) for searching of the minimum data.
  - Sorted linked list:
    - Insertion requires O(n) for searching of the appropriate position.
    - DeleteMin requires O(1).

Implementations of Priority Queue

- Use BST:
  - Insertion requires $O(\log_2 n)$.
  - DeleteMin requires $O(\log_2 n)$.
  - But DeleteMin , repeatedly removing node in the left subtree, seem to hurt balance of the tree.

Implementations of Priority Queue

- Use min-heap:
  - Insertion requires $O(\log_2 n)$.
  - DeleteMin requires $O(\log_2 n)$.

Insert and Remove element into/from priority queue

<ErrorCode> **InsertElement** (val DataIn <DataType>):

InsertHeap Algorithm

<ErrorCode> **DeleteMin** (ref MinData <DataType>):

DeleteHeap Algorithm

Retrieve minimum element in priority queue

<ErrorCode> **RetrieveMin** (ref MinData <DataType>)

Retrieves the minimum element in the heap.

Post MinData receives the minimum data in the heap and the heap remains unchanged.

Return *underflow* or *success*

1. **if** (heap is empty)
    1. return *underflow*
2. **else**
    1. MinData = Data[0]
    2. return *success*

End RetrieveMin

Retrieve maximum element in priority queue

<ErrorCode> **RetrieveMax** (ref MaxData <DataType>)

Retrieves the maximum element in the heap.

Post MaxData receives the maximum data in the heap and the heap remains unchanged.

Return *underflow* or *success*

1. **if** (heap is empty)
    1. return *underflow*
2. **else**
    1. Sequential search the maximum data in the right half elements of the heap (the leaves of the heap). The first leaf is at the position count/2.
    2. return *success*

End RetrieveMax

Change the priority of an element in priority queue

<ErrorCode> **IncreasePriority** (val position <int>,
val PriorityDelta <KeyType>)

Increases priority of an element in the heap.

Post Element at position has its priority increased by PriorityDelta and has been moved to correct position.

Return *rangeError* or *success*

Uses ReheapDown.

Change the priority of an element in priority queue

<ErrorCode> **DecreasePriority** (val position <int>,

val PriorityDelta <KeyType>)

Decreases priority of an element in the heap.

Post Element at position has its priority decreased by PriorityDelta and has been moved to correct position.

Return *rangeError* or *success*

Uses ReheapUp.

Remove an element out of priority queue

<ErrorCode> **DeleteElement** (val position <int>,
ref DataOut <DataType>)

Removes an element out of the min-heap.

Post DataOut contains data in the element at position, this element has been removed. The heap has been rearranged.

Return *rangeError* or *success*

1. **if** (position>=count) OR (position <0)
    1. return *rangeError*
2. **else**
    1. DataOut = Data[position]
    2. DecreasePriority(position, VERY_LARGE_VALUE),
    3. DeleteMin(MinData)
    4. return *success*

End DeleteElement

Advanced implementations of heaps

- Advanced implementations of heaps: use of pointers
  - Leftist heap
  - Skew heap
  - Binomial queues

Use of pointers allows the merge operations (combine two heaps into one) to perform easily.