# Tables and Priority Queues

Initially prepared by Dr. İlyas Çiçekli; improved by various Bilkent CS202 instructors.

# Tables

- Appropriate for problems that must manage data by value.

- Some important operations of tables:
  - Inserting a data item containing the value x.
  - Delete a data item containing the value x.
  - Retrieve a data item containing the value x.

- Various table implementations are possible.
  - We have to analyze the possible implementations so that we can make an intelligent choice.
    - Some operations are implemented more efficiently in certain implementations.

**An ordinary table of cities**

| City | Country | Population |
|---|---|---|
| Athens | Greece | 2,500,000 |
| Barcelona | Spain | 1,800,000 |
| Cairo | Egypt | 9,500,000 |
| London | England | 9,400,000 |
| New York | U.S.A. | 7,300,000 |
| Paris | France | 2,200,000 |
| Rome | Italy | 2,800,000 |
| Toronto | Canada | 3,200,000 |
| Venice | Italy | 300,000 |

# Table Operations

- Some of the table operations are possible:

  - Create an empty table
  - Destroy a table
  - Determine whether a table is empty
  - Determine the number of items in the table
  - Insert a new item into a table
  - Delete the item with a given search key
  - Retrieve the item with a given search key
  - Traverse the table

- The client may need a subset of these operations or require more
- Are keys in the table are unique?
  - We will assume that keys in our tables are unique.
  - But, some other tables allow duplicate keys.

# Selecting an Implementation

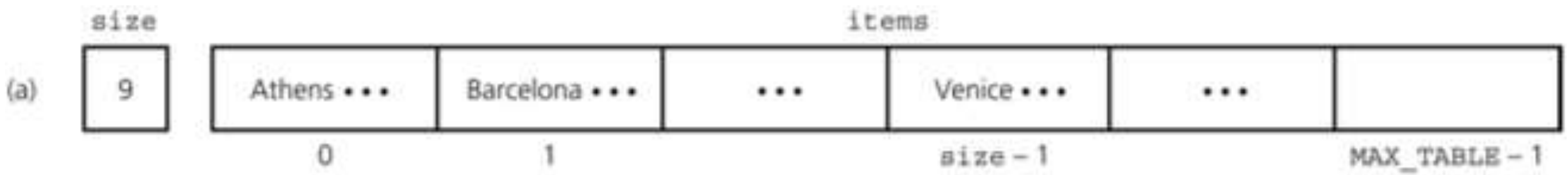
- Since an array or a linked list represents items one after another, these implementations are called **linear**.

- There are four categories of linear implementations:
  - Unsorted, array based (an unsorted array)
  - Unsorted, pointer based (a simple linked list)
  - Sorted (by search key), array based (a sorted array)
  - Sorted (by search key), pointer based (a sorted linked list).

- We have also **nonlinear** implementations such as binary search trees.
  - Binary search tree implementation offers several advantages over linear implementations.

# Sorted Linear Implementations

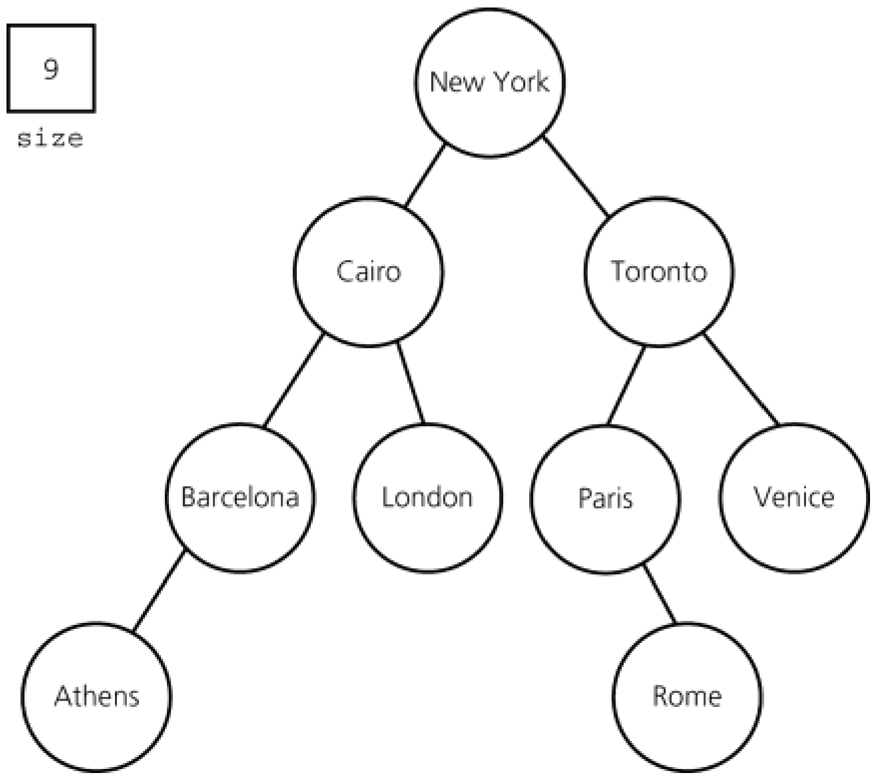
## Array-based implementation




## Pointer-based implementation

# A Nonlinear Implementation

## Binary search tree implementation



# Which Implementation?

- It depends on our application.

- Answer the following questions before selecting an implementation.

  1. **What operations are needed?**
     - Our application may not need all operations.
     - Some operations can be implemented more efficiently in one implementation, and some others in another implementation.

  2. **How often is each operation required?**
     - Some applications may require many occurrences of an operation, but other applications may not.
       – For example, some applications may perform **many retrievals**, but not so many insertions and deletions. On the other hand, other applications may perform **many insertions and deletions.**

# How to Select an Implementation – Scenario A

- **Scenario A:** Let us assume that we have an application:
  - Inserts data items into a table.
  - After all data items are inserted, traverses this table in no particular order.
  - Does not perform any retrieval and deletion operations.

- Which implementation is appropriate for this application?
  - Keeping the items in a sorted order provides no advantage for this application.
    - In fact, it will be more costly for this application.

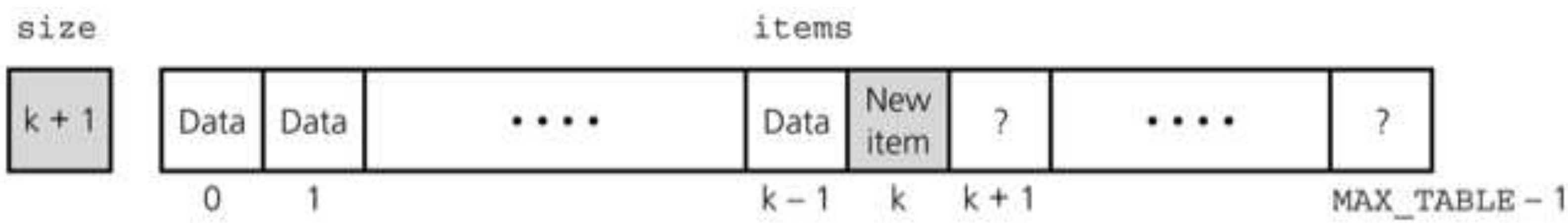
      ➔ ***Unsorted implementation is more appropriate.***

# How to Select an Implementation – Scenario A

- Which unsorted implementation (array-based, pointer-based)?
  - Do we know the maximum size of the table?
    - If we know the expected size is close to the maximum size of the table

➔ ***an array-based implementation is more appropriate***

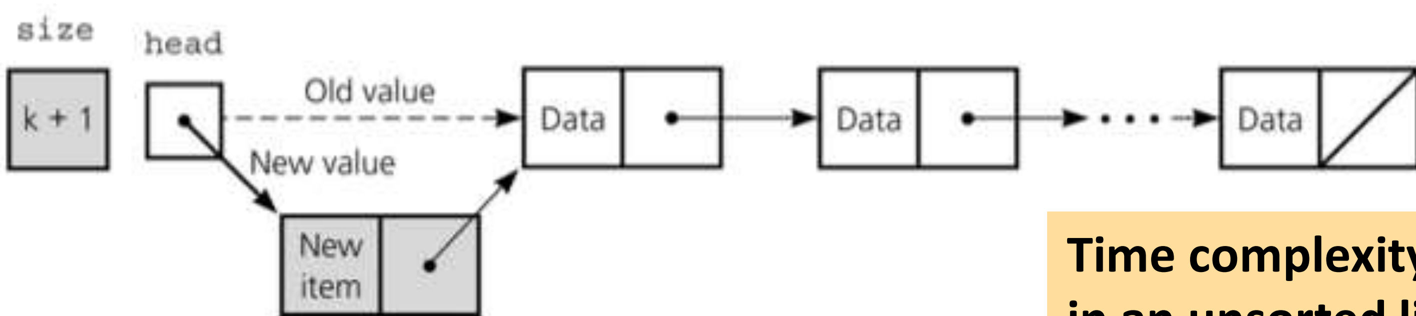
***(because a pointer-based implementation uses extra space for pointers)***




- Otherwise,

➔ ***a pointer-based implementation is more appropriate***

***(because too many entries will be empty in an array-based implementation)***




**Time complexity of insertion in an unsorted list: O(1)**

# How to Select an Implementation – Scenario B

- **Scenario B:** Let us assume that we have an application:
  - Performs many retrievals, but few insertions and deletions
    - E.g., a thesaurus (to look up synonyms of a word)

- For this application, ***a sorted implementation is more appropriate***
  - We can use binary search to access data, if we have sorted data.
  - A sorted linked-list implementation is not appropriate since binary search is not practical with linked lists.
    - If we know the maximum size of the table

    ➔ ***a sorted array-based implementation is more appropriate for frequent retrievals.***
    - Otherwise

    ➔ ***a binary search tree implementation is more appropriate for frequent retrievals.***

    ***(in fact, balanced binary search trees will be used)***

# How to Select an Implementation – Scenario C

- **Scenario C:** Let us assume that we have an application:
  - Performs many retrievals as well as many insertions and deletions.

? Sorted Array Implementation

- Retrievals are efficient.
- But insertions and deletions are not efficient. shifting data

➔ ***a sorted array-based implementation is not appropriate for this application.***

? Sorted Linked List Implementation

- Retrievals, insertions, and deletions are not efficient.

➔ ***a sorted linked-list implementation is not appropriate for this application.***

?Binary Search Tree Implementation

- Retrieval, insertion, and deletion are efficient in the average case.

➔ ***a binary search tree implementation is appropriate for this application.***
***(provided that the height of the BST is O(logn))***

# Which Implementation?

- Linear implementations of a table can be appropriate despite its difficulties.
  - Linear implementations are easy to understand, easy to implement.
  - For small tables, linear implementations can be appropriate.
  - For large tables, linear implementations may still be appropriate (e.g., for the case that has only insertions to an unsorted table--Scenario A)

- In general, a **binary search tree** implementation is a better choice.
  - Worst case: $O(n)$ for most table operations
  - Average case: $O(\log_2 n)$ for most table operations

- Balanced binary search trees increase the efficiency.

# Which Implementation?

**The average-case time complexities of the table operations**

| | Insertion | Deletion | Retrieval | Traversal |
|---|---|---|---|---|
| Unsorted array based | O(1) | O(n) | O(n) | O(n) |
| Unsorted pointer based | O(1) | O(n) | O(n) | O(n) |
| Sorted array based | O(n) | O(n) | O(log n) | O(n) |
| Sorted pointer based | O(n) | O(n) | O(n) | O(n) |
| Binary search tree | O(log n) | O(log n) | O(log n) | O(n) |

# Binary Search Tree Implementation – TableB.h

```
#include "BST.h"// Binary search tree operations
typedef TreeItemType TableItemType;

class Table {
public:
   Table();      // default constructor
   // copy constructor and destructor are supplied by the compiler

   bool tableIsEmpty() const;
   int  tableLength() const;
   void tableInsert(const TableItemType& newItem) throw(TableException);
   void tableDelete(KeyType searchKey) throw(TableException);
   void tableRetrieve(KeyType searchKey, TableItemType& tableItem) const
                                                    throw(TableException);
   void traverseTable(FunctionType visit);

protected:
   void setSize(int newSize);
private:
   BinarySearchTree bst;                 // BST that contains the table's items
   int size;                                     // Number of items in the table
}
```

# Binary Search Tree Implementation – tableInsert

```
#include "TableB.h"// header file

void Table::tableInsert(const TableItemType& newItem) throw(TableException) {
   try {
        bst.searchTreeInsert(newItem);
        ++size;
   }
   catch (TreeException e){
        throw TableException("Cannot insert item");
   }
}
```

# The Priority Queue

**Priority queue** is a variation of the table.

- Each data item in a priority queue has a priority value.
- Using a priority queue we prioritize a list of tasks:
  - Job scheduling

Major operations:

- **Insert** an item with a priority value into its proper position in the priority queue.
- **Deletion is not the same** as the deletion in the table. We delete the item with the **highest priority**.

# Priority Queue Operations

create – creates an empty priority queue.

destroy – destroys a priority queue.

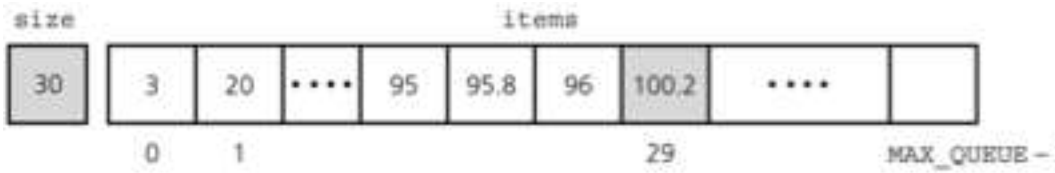
isEmpty – determines whether a priority queue is empty or not.

insert – inserts a new item (with a priority value) into a priority queue.

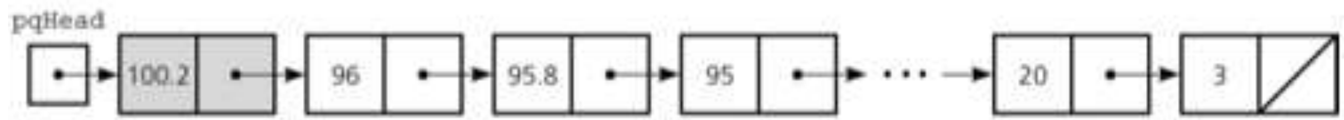
delete – retrieves the item in a priority queue with the highest priority value, and deletes that item from the priority queue.

# Which Implementations?

1. Array-based implementation
   - Insertion will be O(n)

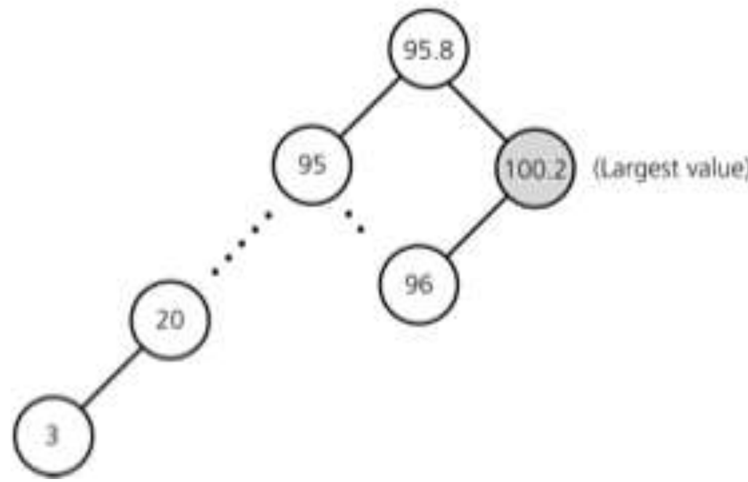



2. Linked-list implementation
   - Insertion will be O(n)




3. BST implementation
   - Insertion is $O(\log_2 n)$ in average but O(n) in the worst case.

**We need a balanced BST so that we can get better performance [O(logn) in the worst case] ➔ HEAP**



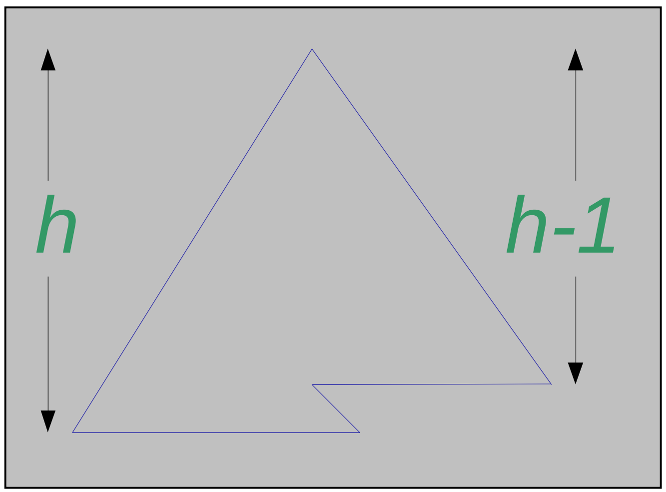
# Heaps

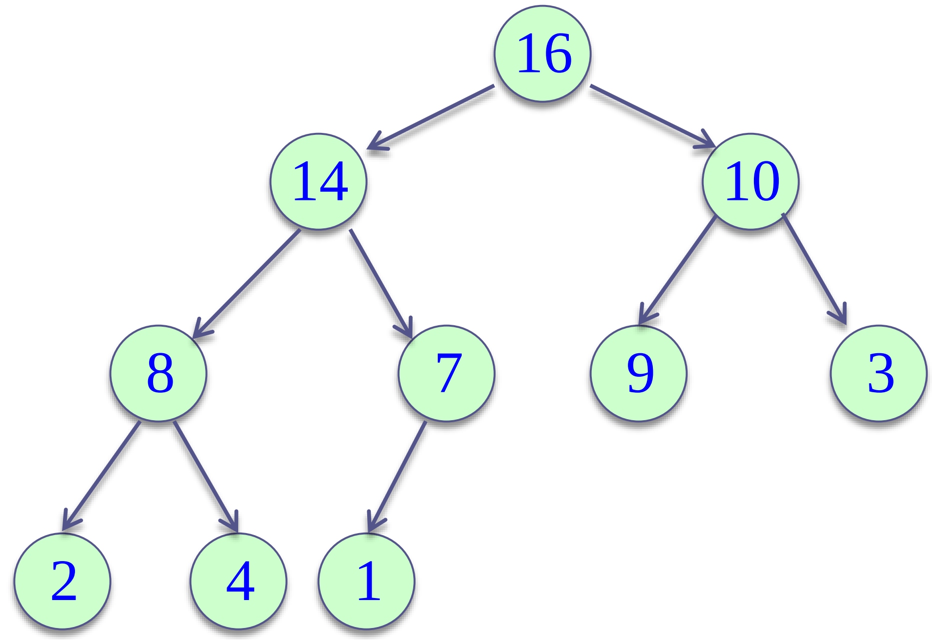
***Definition:*** A **heap** is a complete binary tree such that

- It is empty, or
- Its root contains a search key greater than or equal to the search key in each of its children, and each of its children is also a heap.

- Since the root contains the item with the largest search key, heap in this definition is also known as ***maxheap***.
- On the other hand, a heap which places the smallest search key in its root is know as ***minheap***.
- We will talk about maxheap as heap in the rest of our discussions.

# Heap Data Structure

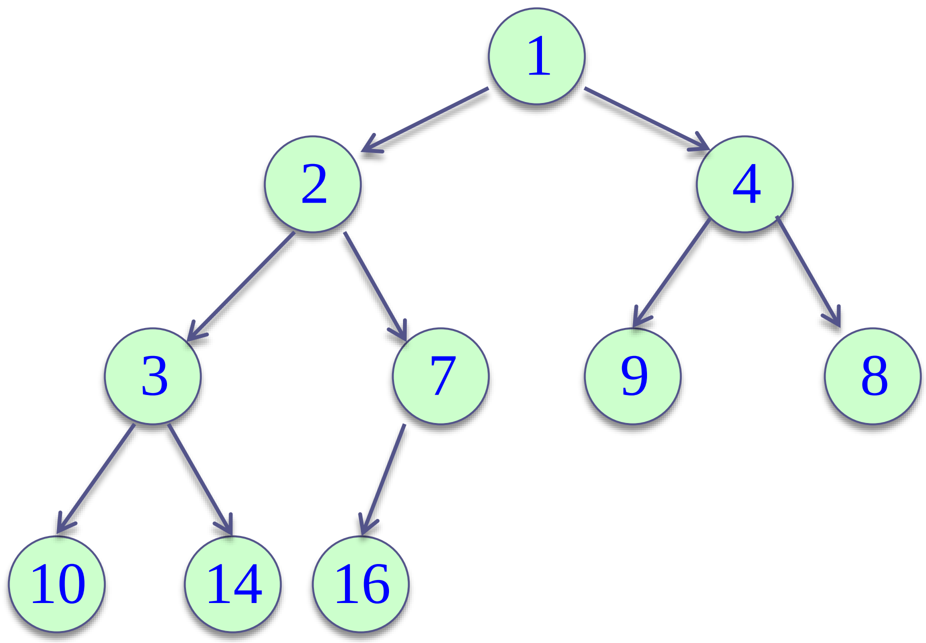



Complete binary tree

- Completely filled on all levels except possibly the lowest level
- The lowest level is filled from left to right

# Heap Property: Min-Heap

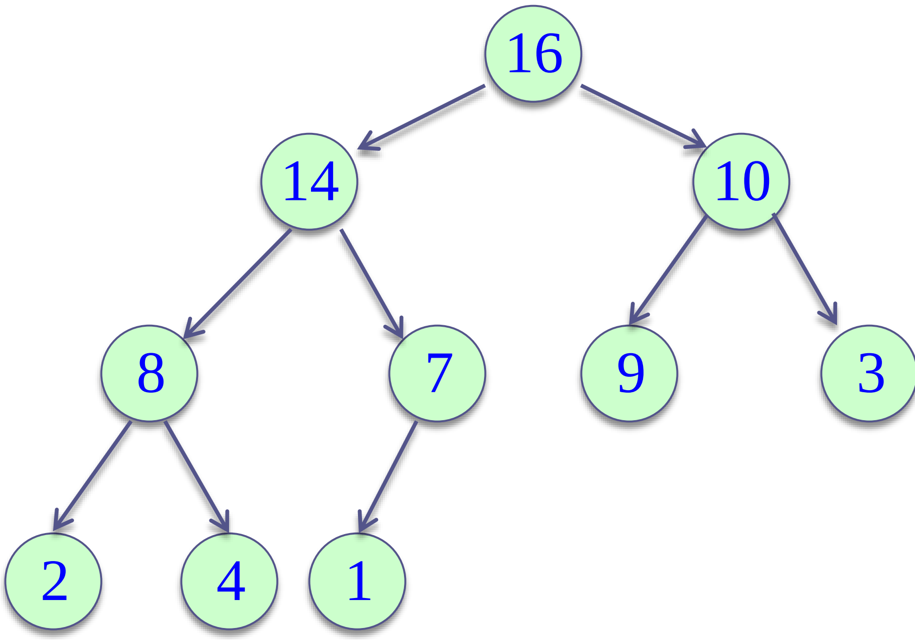



*The smallest element in any subtree is the root element in a min-heap*

*Min heap*: For every node i other than root, $A[parent(i)] \leq A[i]$

➔ Parent node is always smaller than the child nodes

# Heap Property: Max-Heap




*The largest element in any subtree is the root element in a max-heap*

*We will focus on max-heaps*

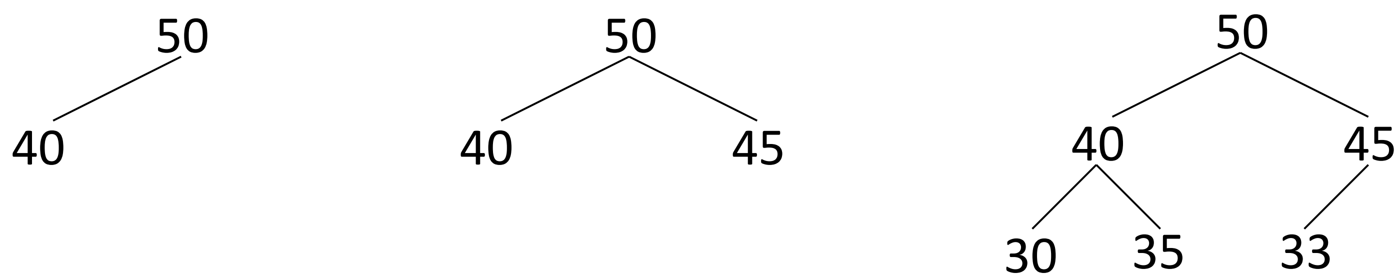
*Max heap*: For every node i other than root, A[parent(i)] ≥ A[i]

➔ Parent node is always larger than the child nodes

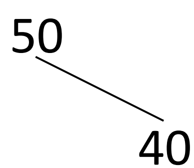
# Differences between a Heap and a BST

- A heap is **NOT** a binary search tree.
  1. A BST can be seen as sorted, but a heap is ordered in much weaker sense.
     - Although it is not sorted, the order of a heap is sufficient for the efficient implementation of priority queue operations.
  2. A BST has different shapes, but a heap is always complete binary tree.

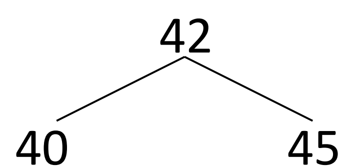
**HEAPS**

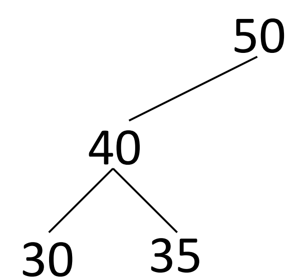



**NOT HEAPS**

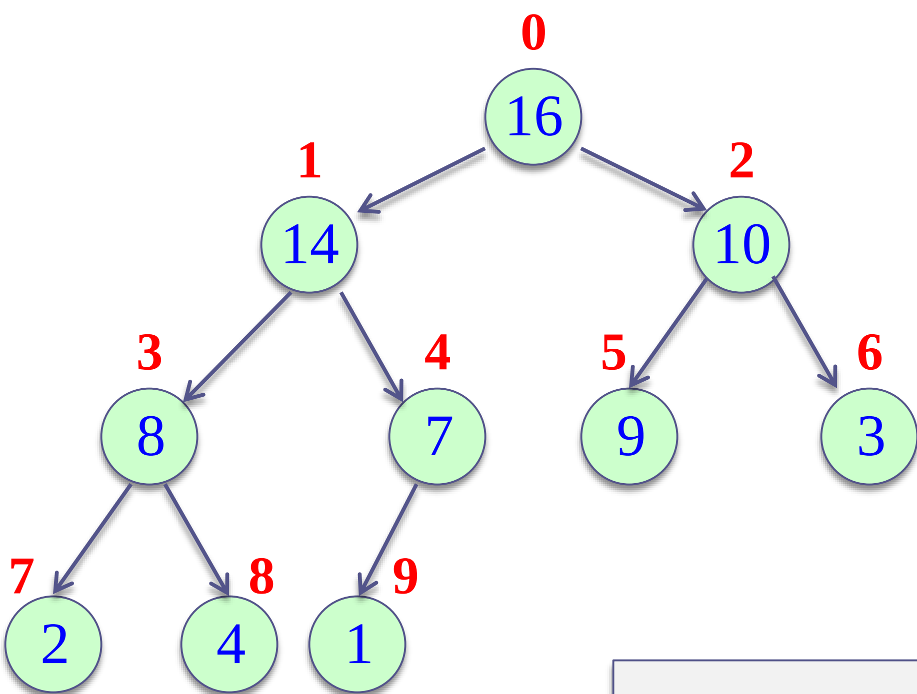



not complete binary tree



# Heap Data Structure

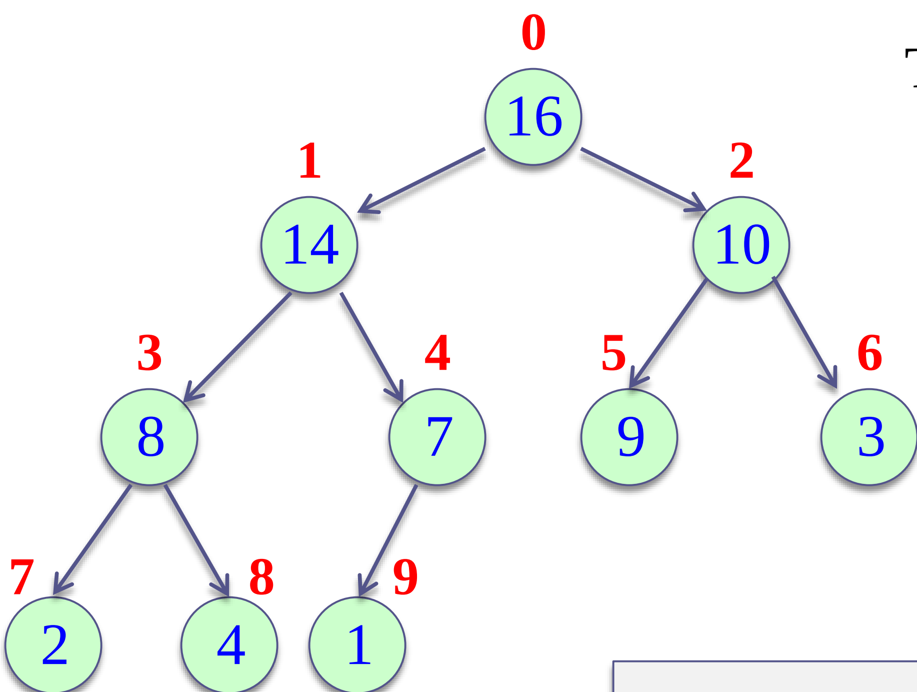



Heap can be stored in a linear array

Storage

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| items | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

# Heap Data Structure




The links in the heap are implicit:

$$left(i) = 2i + 1$$

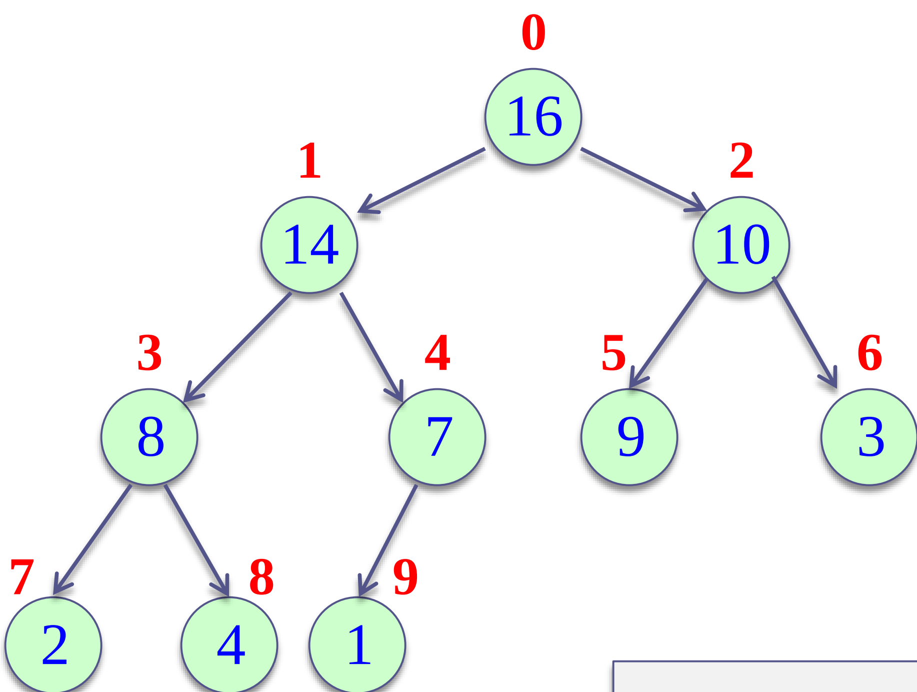
$$right(i) = 2i + 2$$

$$parent(i) = \lfloor (i-1)/2 \rfloor$$

Storage

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| items | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

# Heap Data Structure




$$left(i) = 2i + 1$$

e.g. Left child of node 3 has index 7

$$right(i) = 2i + 2$$

e.g. Right child of node 1 has index 4

$$parent(i) = \lfloor (i-1)/2 \rfloor$$

e.g. Parent of node 6 has index 2

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| items | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

# Heap Data Structure

- items[0] is always the root element

- Array items has two attributes:
  - MAX_SIZE: Size of the memory allocated for array items
  - size: The number elements in heap at a given time

$$\text{size} \leq \text{MAX\_SIZE}$$

# Major Heap Operations

- Two major heap operations are insertion and deletion.

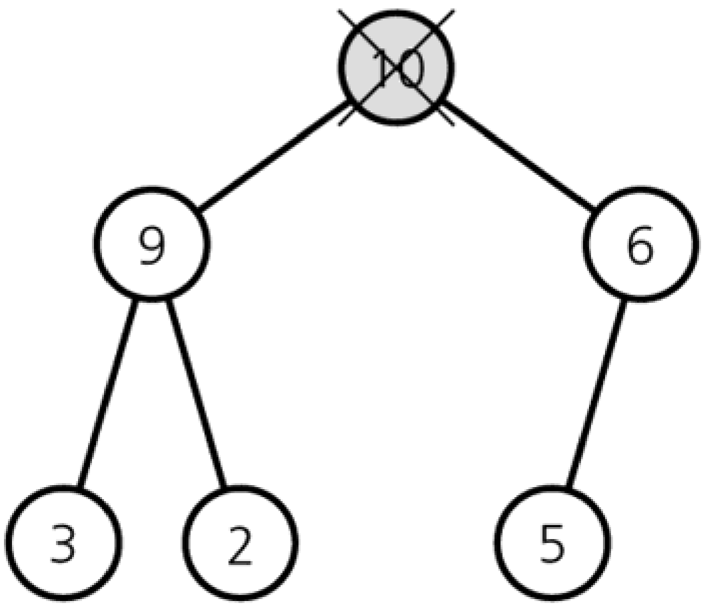
**Insertion**

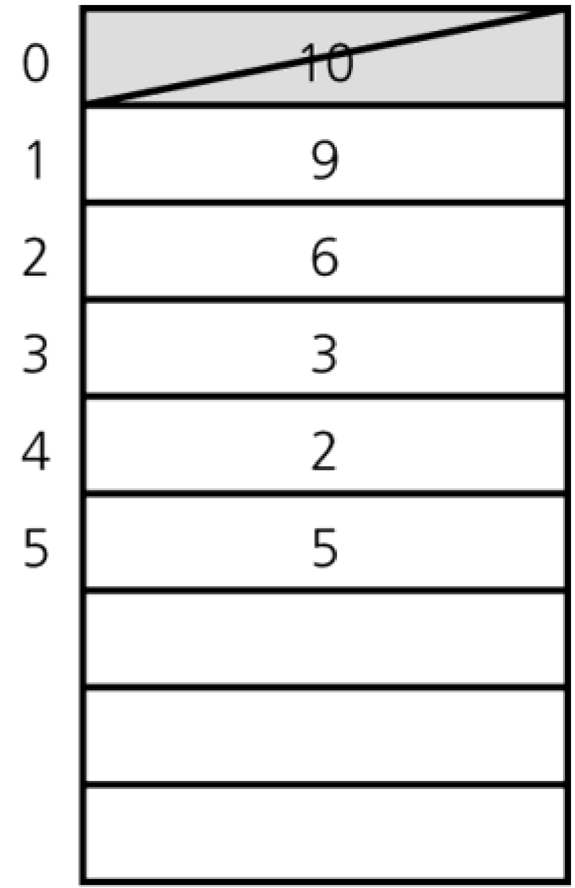
- Inserts a new item into a heap.
- After the insertion, the heap must satisfy the heap properties.

**Deletion**

- Retrieves and deletes the root of the heap.
- After the deletion, the heap must satisfy the heap properties.

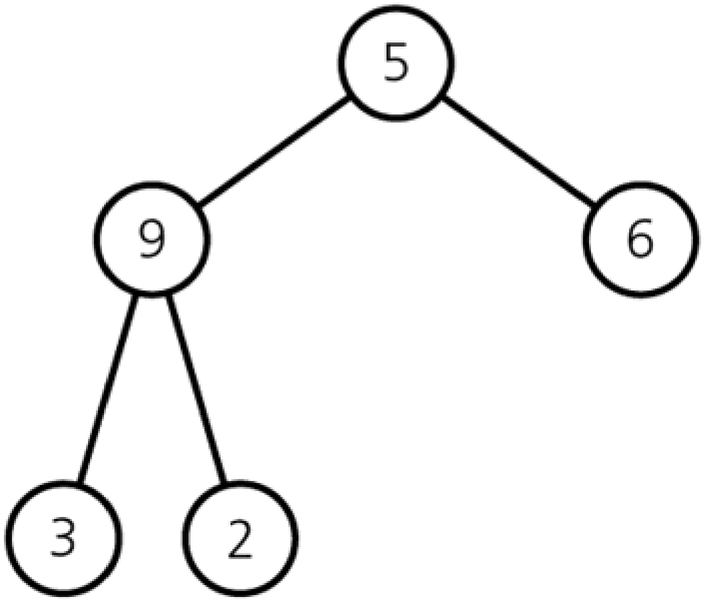
# Heap Delete – First Step

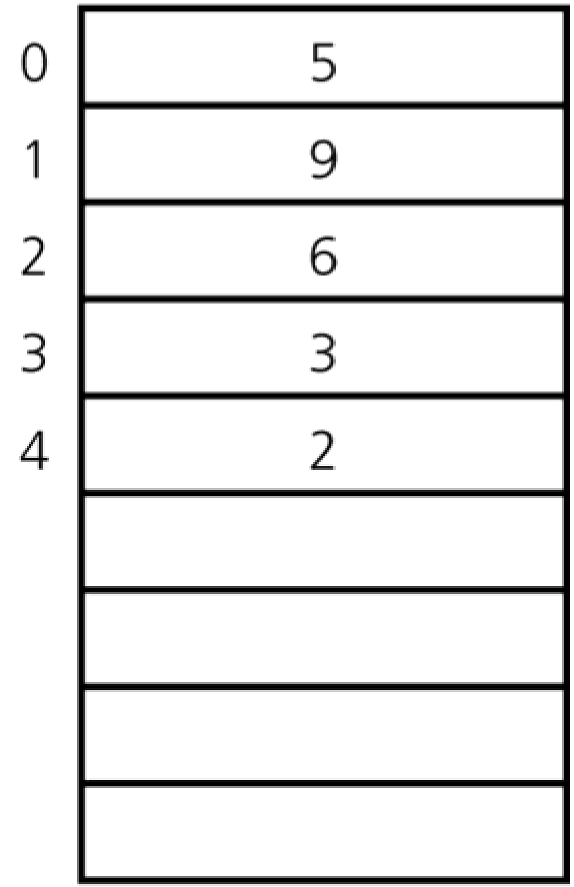



- The first step of **heapDelete** is to retrieve and delete the root.
  - This creates two disjoint heaps.

# Heap Delete – Second Step

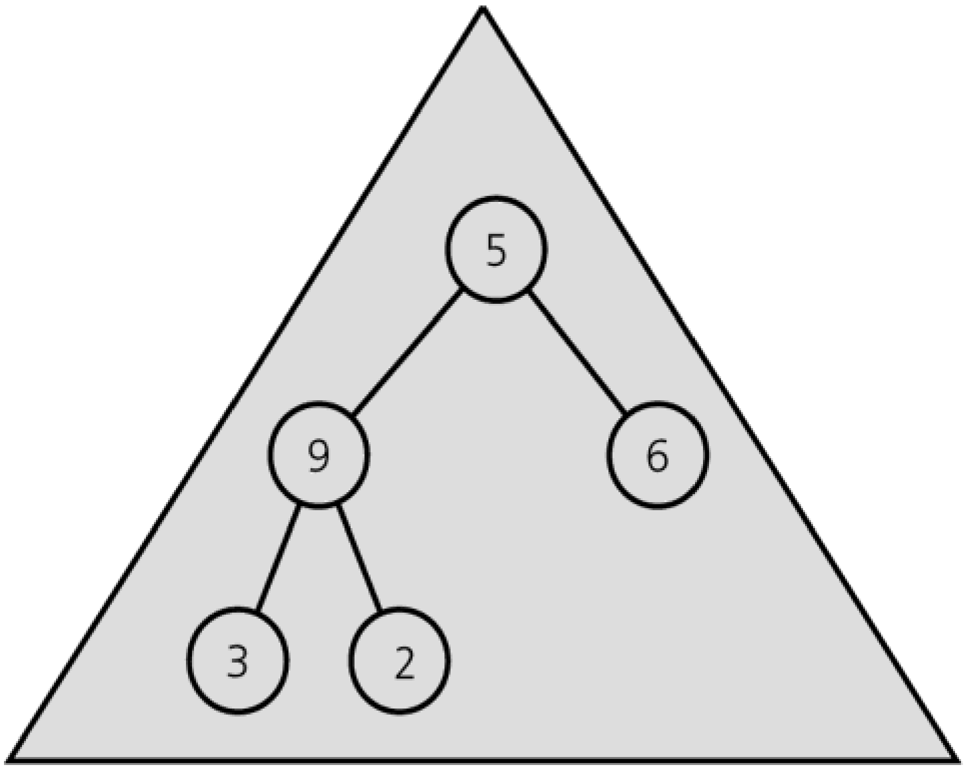


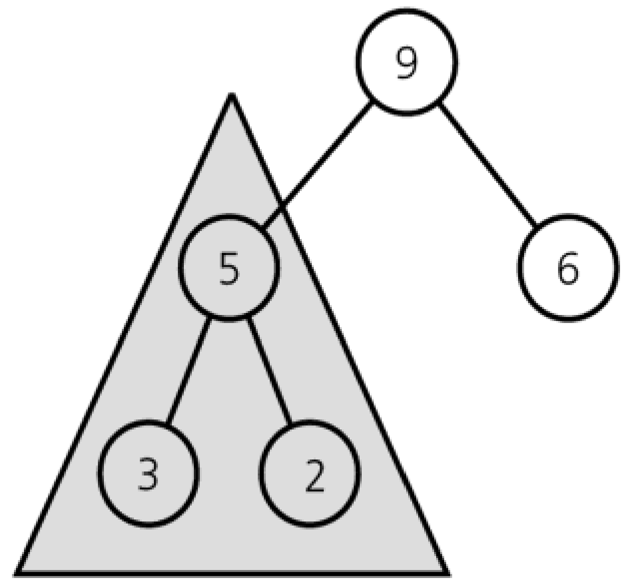

- Move the last item into the root.
- The resulting structure may not be heap; it is called as **semiheap**.

# Heap Delete – Last Step

The last step of *heapDelete* transforms the semiheap into a heap.




First semiheap passed to `heapRebuild`




Second semiheap passed to `heapRebuild`

Recursive calls to **heapRebuild**

# Heap Delete

**<u>heapDelete (items, *size*)</u>**

max ← items[0]

items[0] ← items[*size-1*]

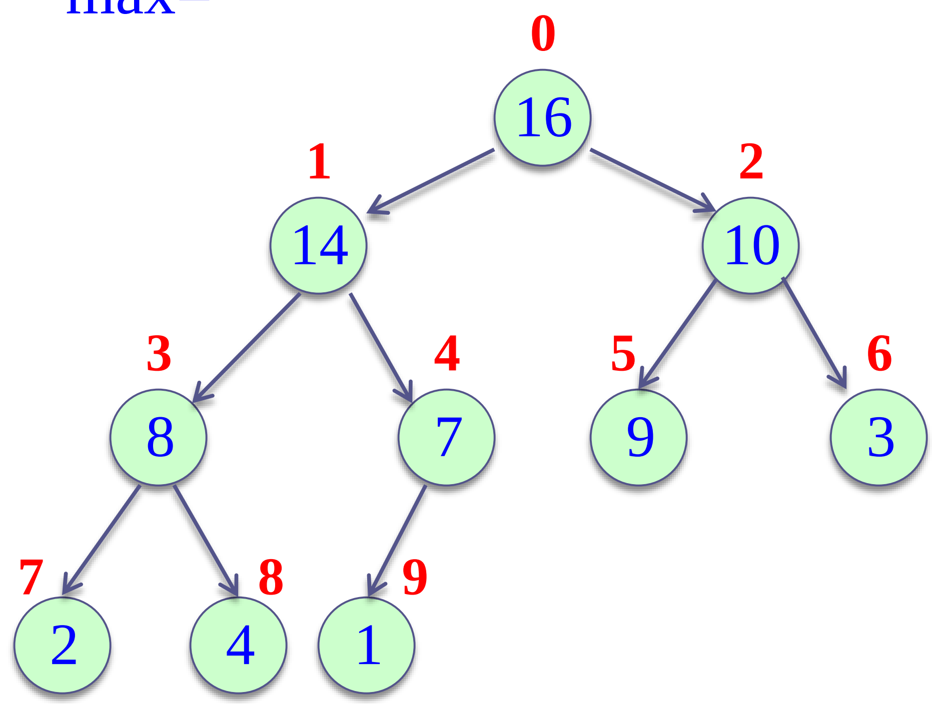
*size* ← *size* – 1

<u>heapRebuild</u>(items, 0, *size*)

**return** max

*Return the max element, and reorganize the heap to maintain heap property*

max=




only decrement size in the array does not actually delete the node

# Rebuild Heap

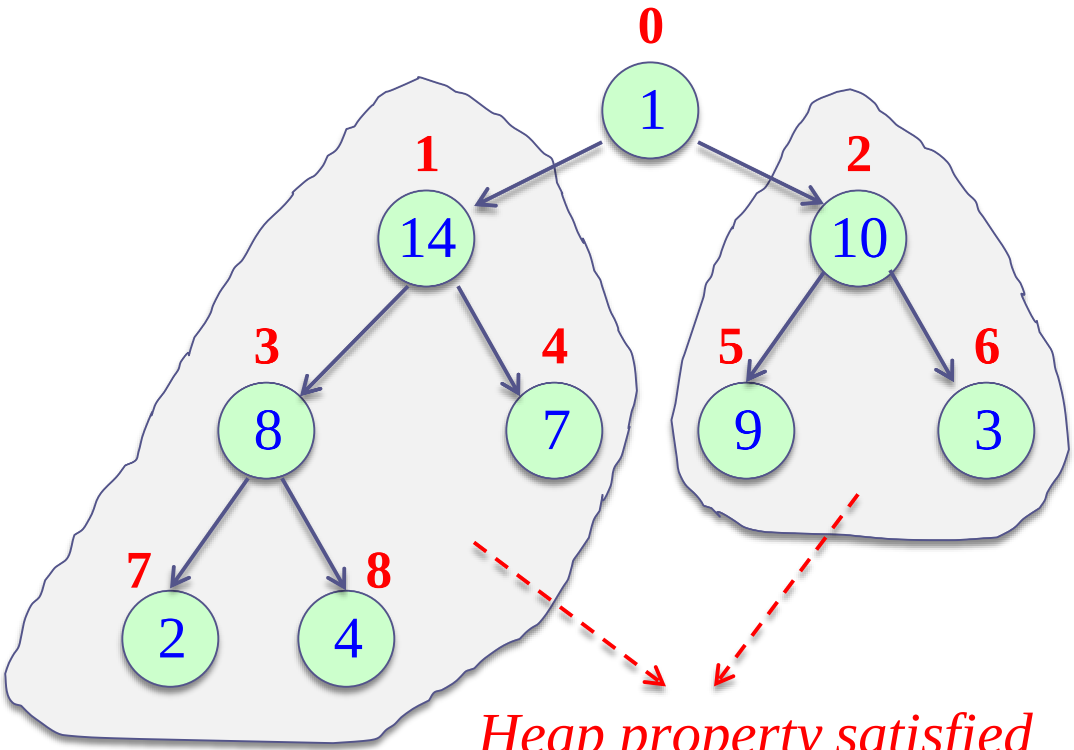
***Maintaining heap property:***

Subtrees rooted at left($i$) and right($i$) are already heaps.

But, items[$i$] may violate the heap property (i.e., may be smaller than its children)

***Idea***: Float down the value at items[$i$] in the heap so that subtree rooted at $i$ becomes a heap.

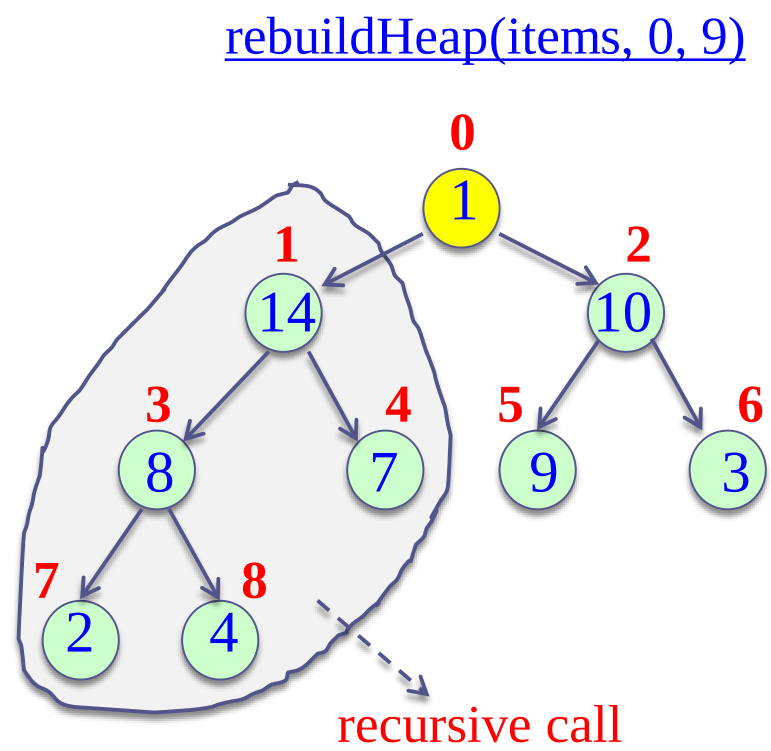
*Heap property violated at the root*




*Heap property satisfied for left and right subtrees*

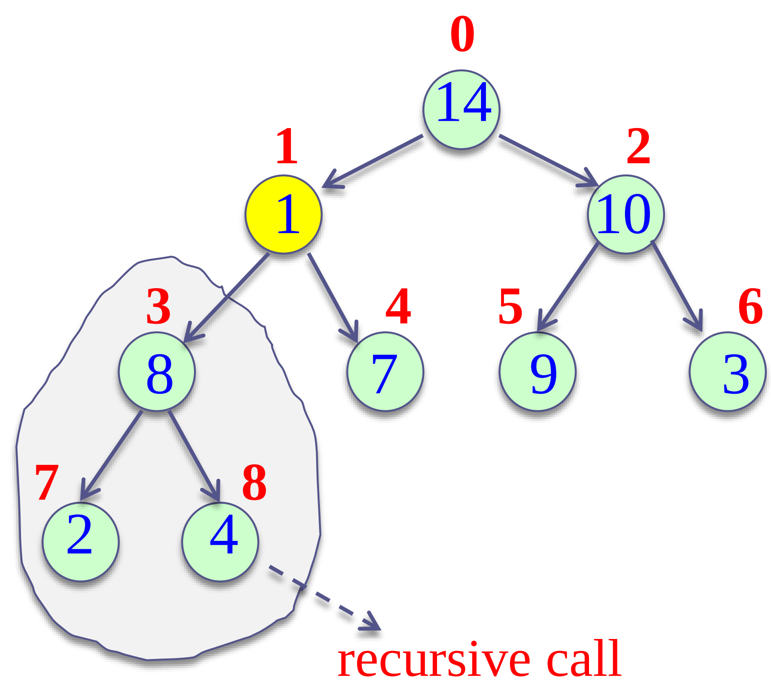
# Rebuild Heap

select larger child



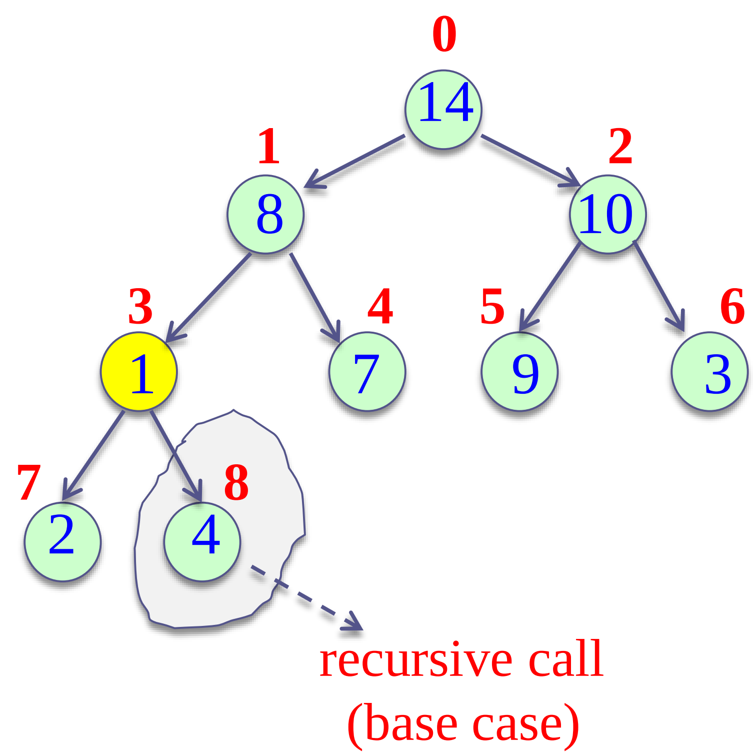
# Rebuild Heap

recursive call:
rebuildHeap(items, 1, 9)



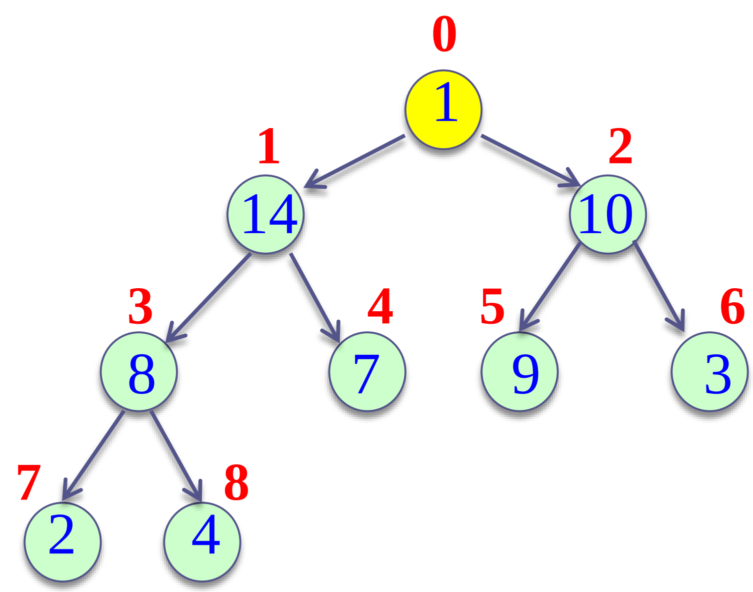
# Rebuild Heap

recursive call:
rebuildHeap(items, 3, 9)



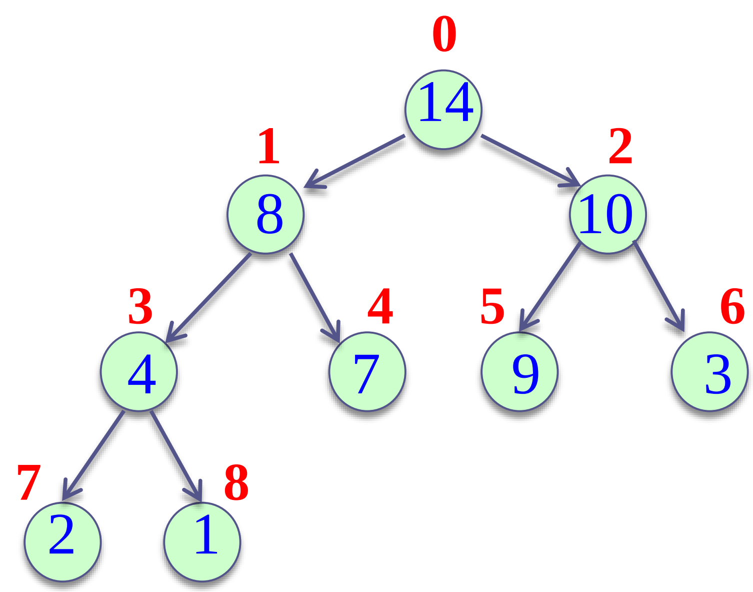
# Rebuild Heap: Summary (Floating Down the Value)

rebuildHeap(items, 0, 9)

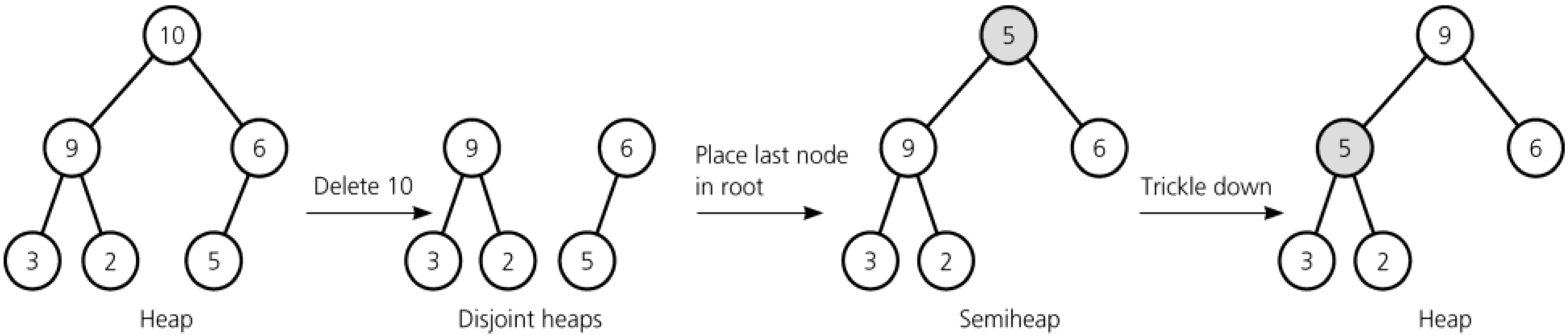


# Heap Operations: Rebuild Heap

after rebuildHeap:



# Heap Delete



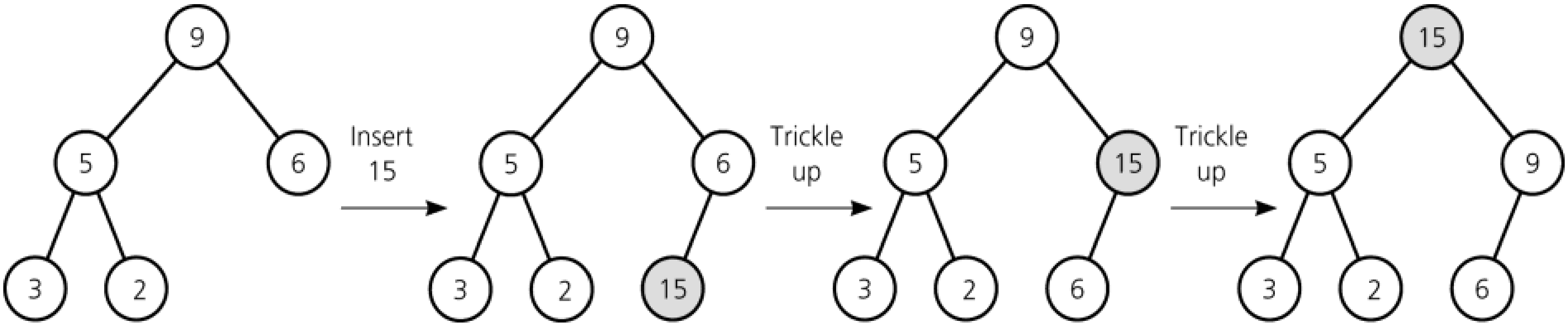

## ANALYSIS

- Since the height of a complete binary tree with n nodes is always $\lceil \log_2(n+1) \rceil$

  ➔ **heapDelete** is **$O(\log_2 n)$** worst case and average case

# Heap Insert

A new item is inserted at the bottom of the tree, and it trickles up to its proper place




## ANALYSIS

- Since the height of a complete binary tree with n nodes is always $\lceil \log_2(n+1) \rceil$

➔ **heapInsert** is **$O(\log_2 n)$** worst case and average case

# Heap Implementation

```
constint MAX_HEAP = maximum-size-of-heap;
#include "KeyedItem.h"// definition of KeyedItem
typedef KeyedItem HeapItemType;

class Heap {
public:
   Heap();                              // default constructor
   // copy constructor and destructor are supplied by the compiler

   bool heapIsEmpty() const;
   void heapInsert(const HeapItemType& newItem) throw(HeapException);
   void heapDelete(HeapItemType& rootItem) throw(HeapException);

protected:
   void heapRebuild(int root);          // Converts the semiheap rooted at
                                        // index root into a heap
private:
   HeapItemType items[MAX_HEAP];        // array of heap items
   int          size;                   // number of heap items
};
```

# Heap Implementation

```
// Default constructor
Heap::Heap() : size(0) {

}



boolHeap::heapIsEmpty() const {
   return (size == 0);
}
```

# Heap Implementation -- *heapInsert*

```
void Heap::heapInsert(constHeapItemType&newItem) throw(HeapException) {

   if (size >= MAX_HEAP)
        throw HeapException("HeapException: Heap full");

   // Place the new item at the end of the heap
   items[size] = newItem;

   // Trickle new item up to its proper position
   int place = size;
   int parent = (place - 1)/2;
   while ( (place > 0) && (items[place].getKey() > items[parent].getKey()) ) {
        HeapItemType temp = items[parent];
        items[parent] = items[place];
        items[place] = temp;

        place = parent;
        parent = (place - 1)/2;
   }
   ++size;
}
```

can be recursive
heapRebuild(lastIndex)
modify heapRebuild to trickle up
instead of root to leaf

integer division, root index no matter left or right child

# Heap Implementation -- *heapDelete*

```
Void Heap::heapDelete(HeapItemType&rootItem) throw(HeapException) {
   if (heapIsEmpty())
         throwHeapException("HeapException: Heap empty");
   else {
         rootItem = items[0];
         items[0] = items[--size];
         heapRebuild(0);
   }
}
```

# Heap Implementation -- *heapRebuild*

```
voidHeap::heapRebuild(int root) {
   int child = 2 * root + 1;    // index of root's left child, if any
   if ( child < size ) {
        // root is not a leaf so that it has a left child
        int rightChild = child + 1;     // index of a right child, if any

        // If root has right child, find larger child
        if ( (rightChild < size) &&
             (items[rightChild].getKey() >items[child].getKey()) )
                child = rightChild;     // index of larger child

        // If root’s item is smaller than larger child, swap values
        if ( items[root].getKey() < items[child].getKey() ) {
                HeapItemType temp = items[root];
                items[root] = items[child];
                items[child] = temp;

                // transform the new subtree into a heap
                heapRebuild(child);
        }
}
```

# Heap Implementation of *PriorityQueue*

- The heap implementation of the priority queue is straightforward
  - Since the heap operations and the priority queue operations are the same.

- When we use the heap,
  - Insertion and deletion operations of the priority queue will be $O(\log_2 n)$.

# Heap Implementation of *PriorityQueue*

```
#include "Heap.h"// ADT heap operations
typedef HeapItemType PQItemType;

class PriorityQueue {
public:
   // default constructor, copy constructor, and destructor
   // are supplied by the compiler

   // priority-queue operations:
   bool pqIsEmpty() const;
   void pqInsert(const PQItemType& newItem) throw (PQException);
   void pqDelete(PQItemType& priorityItem) throw (PQException);

private:
   Heap h;
};
```

# Heap Implementation of *PriorityQueue*

```
bool PriorityQueue::pqIsEmpty() const {
   return h.heapIsEmpty();
}

void PriorityQueue::pqInsert(const PQItemType& newItem) throw (PQException){
   try {
        h.heapInsert(newItem);
   }
   catch (HeapException e) {
        throw PQueueException("Priority queue is full");
   }
}

void PriorityQueue::pqDelete(PQItemType& priorityItem) throw (PQException) {
   try {
        h.heapDelete(priorityItem);
   }
   catch (HeapException e) {
        throw PQueueException("Priority queue is empty");
   }
}
```

# Heap or Binary Search Tree?

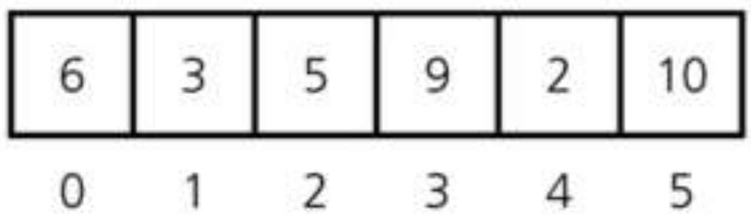
# Heapsort

We can make use of a heap to sort an array:

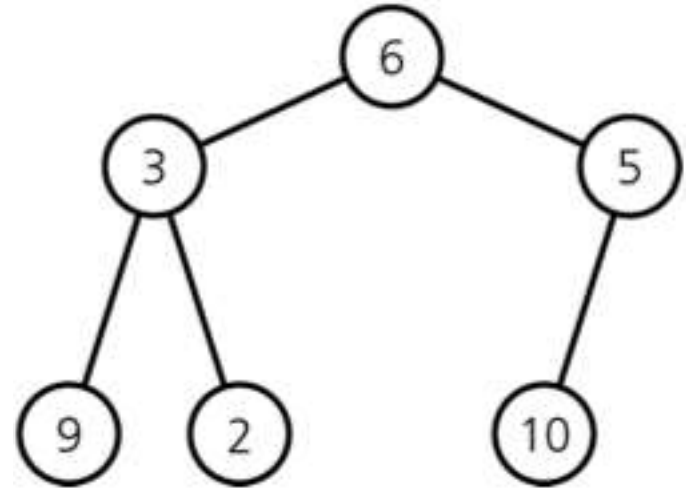
1. Create a heap from the given initial array with n items.
2. Swap the root of the heap with the last element in the heap.
3. Now, we have a semiheap with n-1 items, and a sorted array with one item.
4. Using heapRebuild convert this semiheap into a heap. Now we will have a heap with n-1 items.
5. Repeat the steps 2-4 as long as the number of items in the heap is more than 1.

# Heapsort -- Building a heap from an array

The initial contents of *anArray*

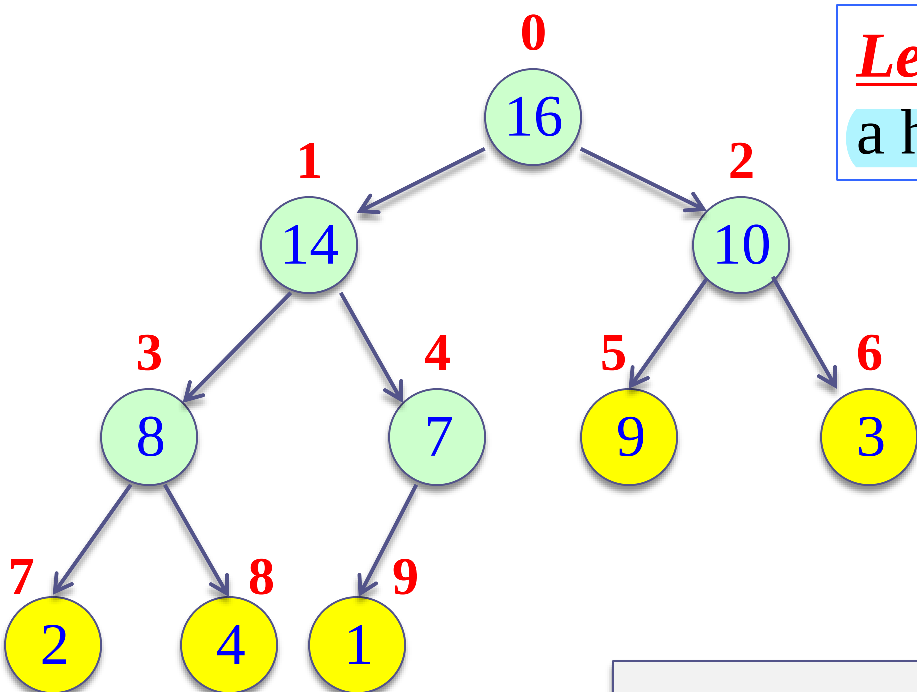



A heap corresponding to *anArray*




```
for (index = n - 1 ;  index >= 0 ;  index--) {
   // Invariant: the tree rooted at index is a semiheap
   heapRebuild(anArray, index, n)
   // Assertion: the tree rooted at index is a heap.
}
```

# Where are the leaves stored?




***Lemma***: The last $\lceil n/2 \rceil$ nodes of a heap are *all leaves*

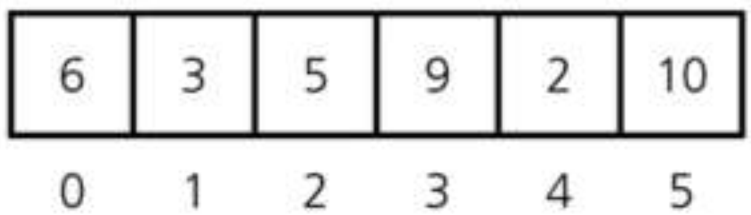
it is pointless to call heapRebuild after the index 4

Storage

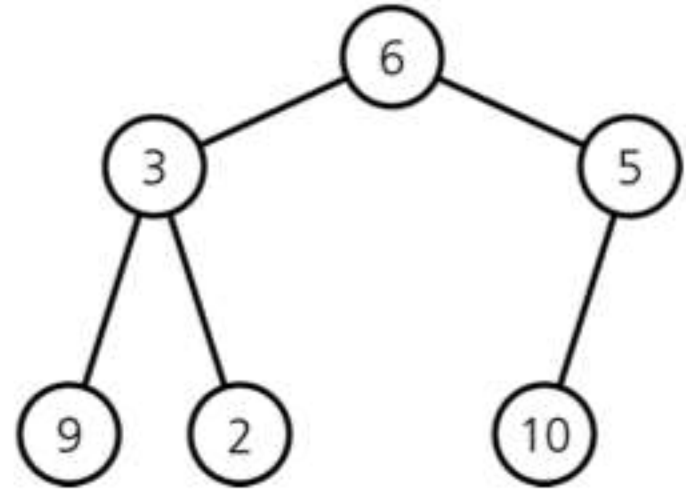
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

# Heapsort -- Building a heap from an array

The initial contents of *anArray*

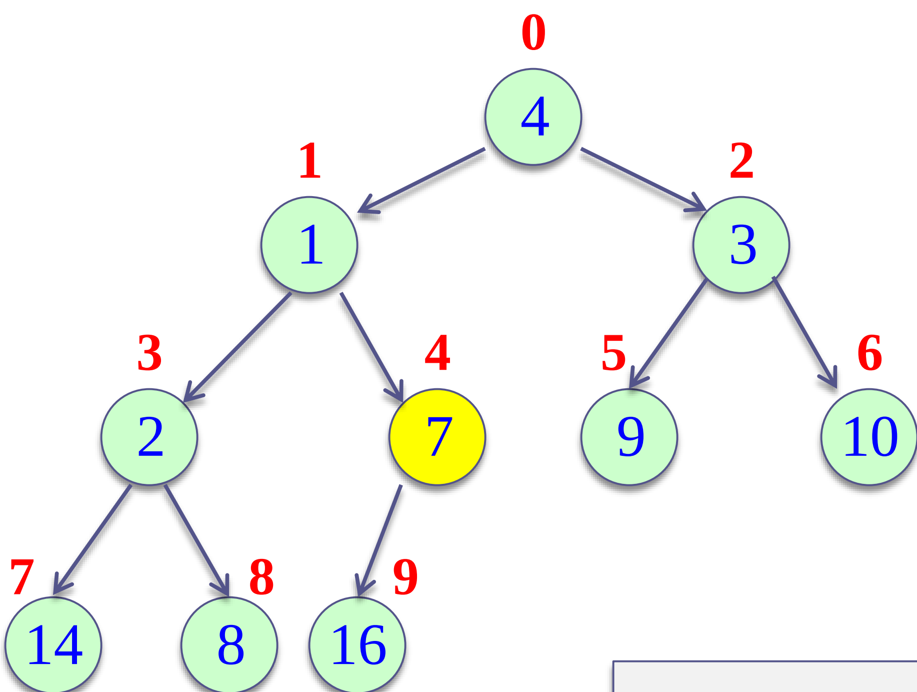



A heap corresponding to *anArray*




```
for (index = (n/2) - 1 ;  index >= 0 ;  index--) {
   → MORE EFFICIENT
   // Invariant: the tree rooted at index is a semiheap
   heapRebuild(anArray, index, n)
   // Assertion: the tree rooted at index is a heap.
}
```

# Build-Heap: Example

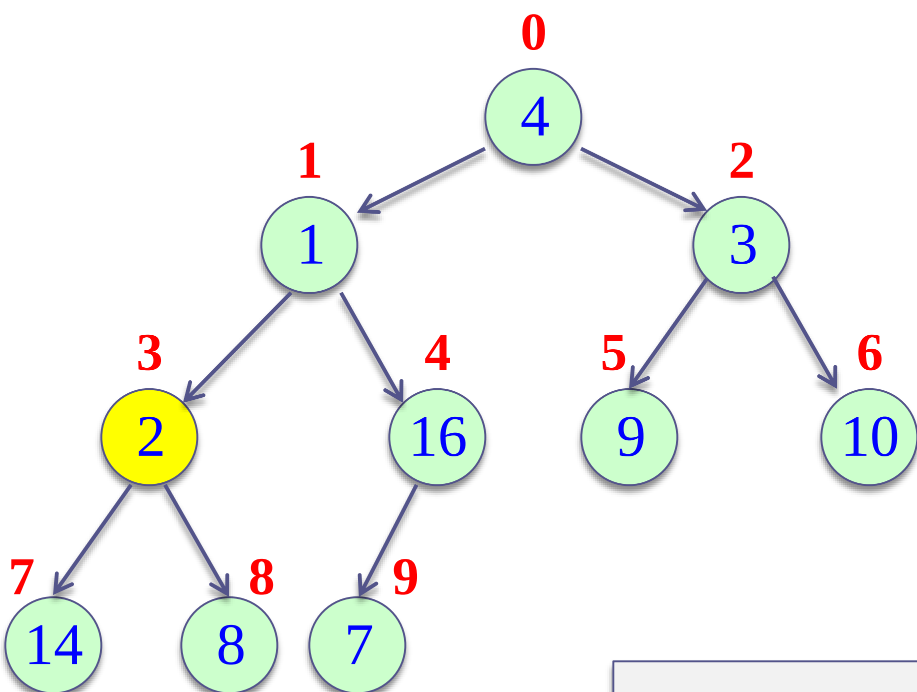



*index=4*

heapRebuild(A, 4, 10)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 4 | 1 | 3 | 2 | 7 | 9 | 10 | 14 | 8 | 16 |

# Build-Heap: Example

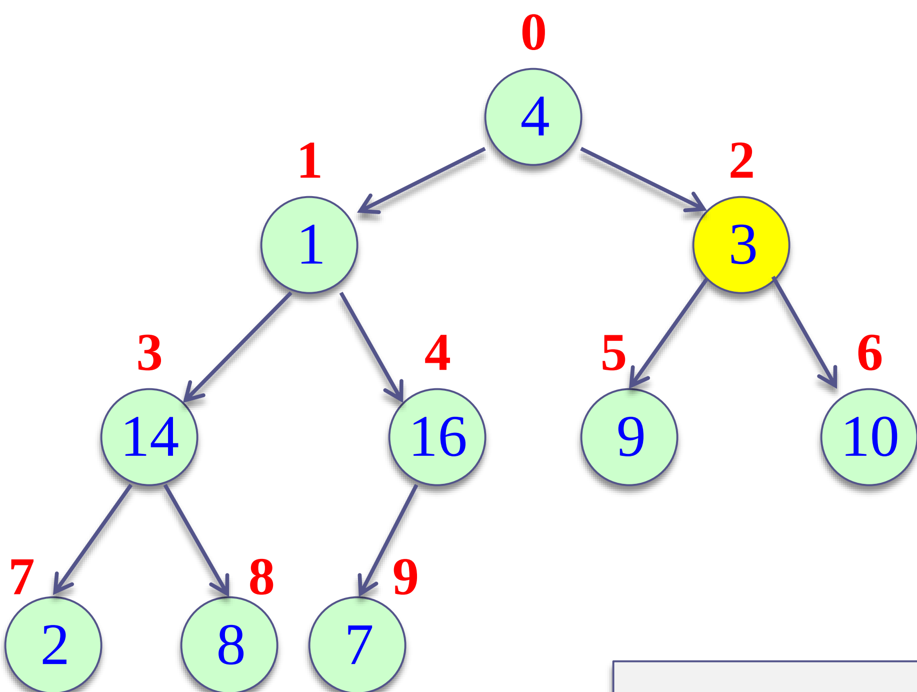



*index=3*

heapRebuild(A, 3, 10)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |

# Build-Heap: Example

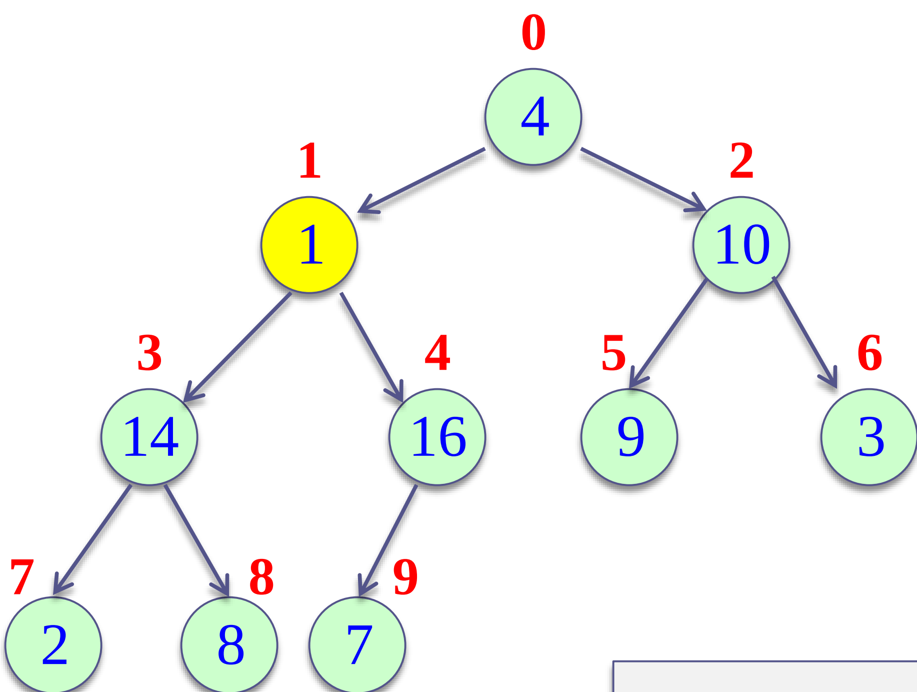



*index=2*

heapRebuild(A, 2, 10)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 4 | 1 | 3 | 14 | 16 | 9 | 10 | 2 | 8 | 7 |

# Build-Heap: Example

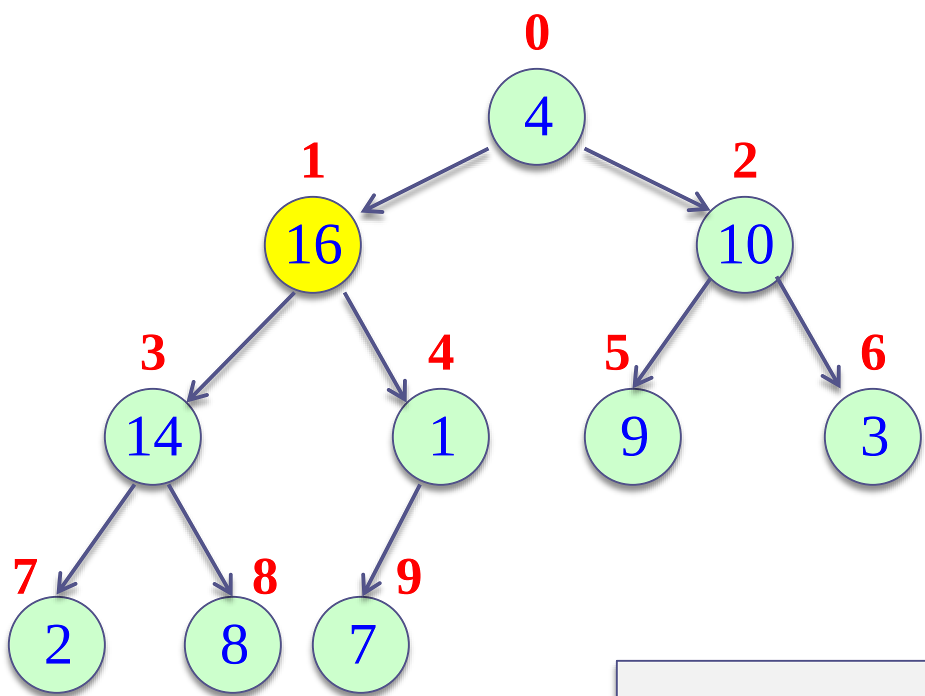



*index=1*

heapRebuild(A, 1, 10)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 4 | 1 | 10 | 14 | 16 | 9 | 3 | 2 | 8 | 7 |

# Build-Heap: Example

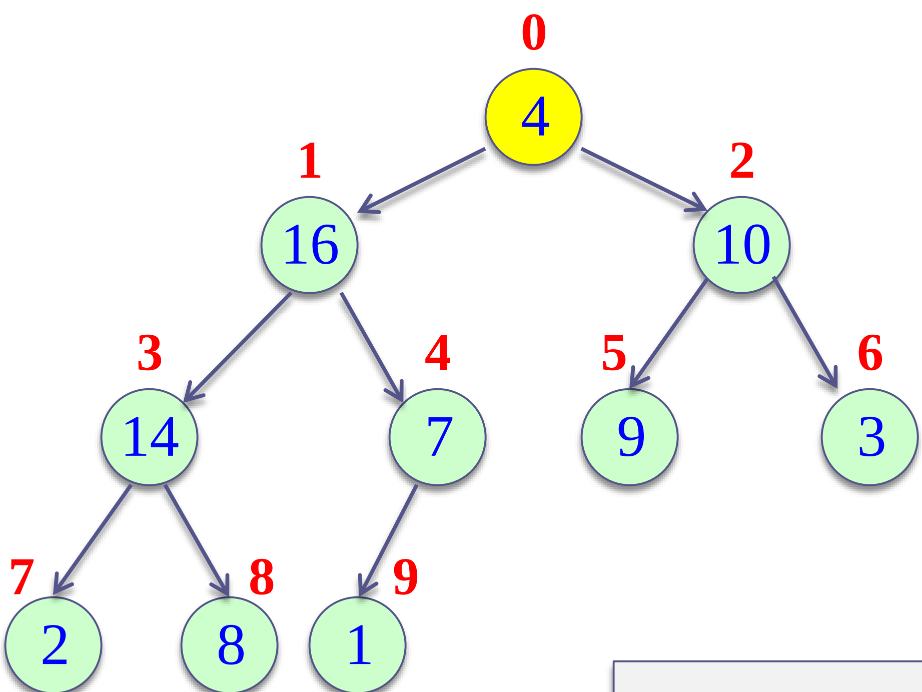



*index=1 (cont'd)*

heapRebuild(A, 1, 10)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 4 | 16 | 10 | 14 | 1 | 9 | 3 | 2 | 8 | 7 |

# Build-Heap: Example

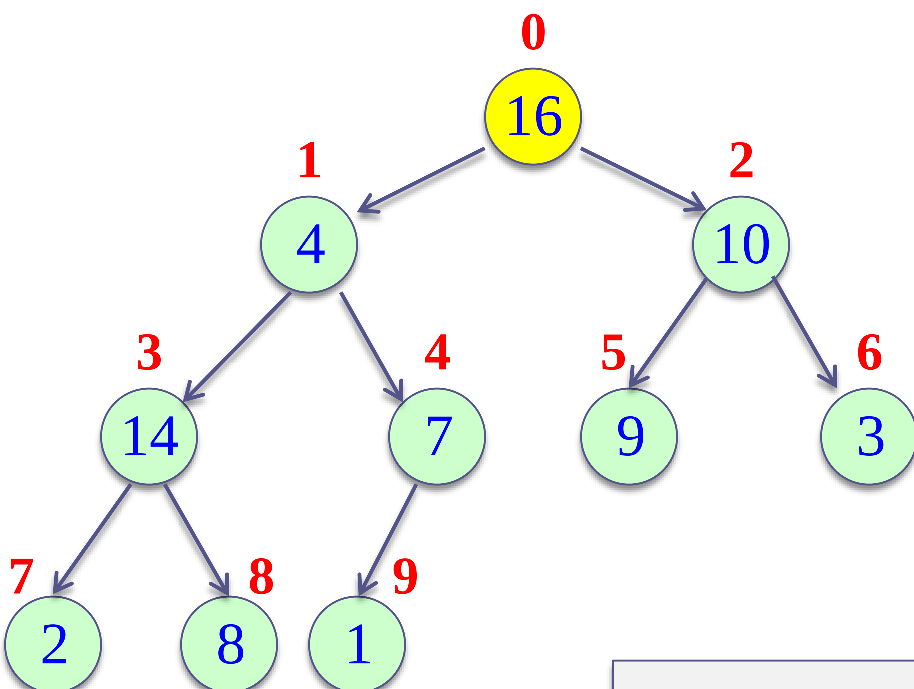



*index=0*

heapRebuild(A, 0, 10)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 4 | 16 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |

# Build-Heap: Example

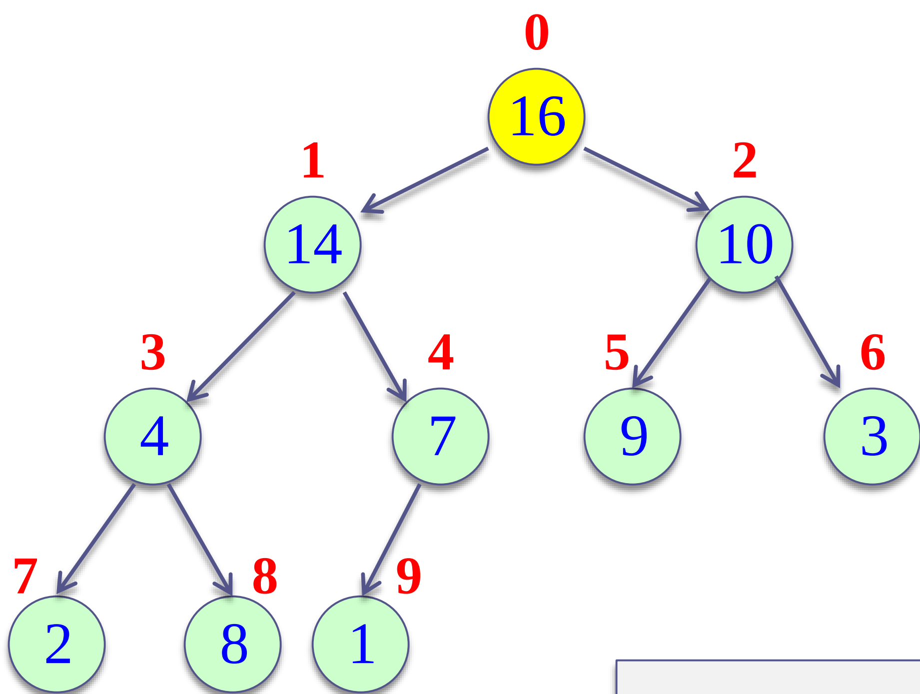



*index=0 (cont'd)*

heapRebuild(A, 0, 10)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 16 | 4 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |

# Build-Heap: Example



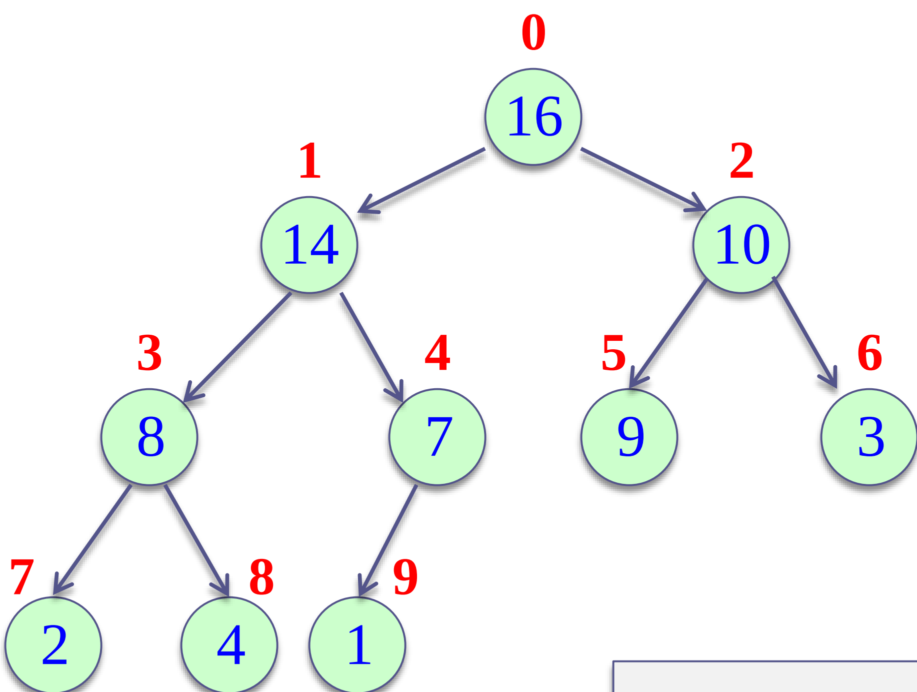

*index=0 (cont'd)*

heapRebuild(A, 0, 10)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 16 | 14 | 10 | 4 | 7 | 9 | 3 | 2 | 8 | 1 |

# Build-Heap: Example

*After Build-Heap*

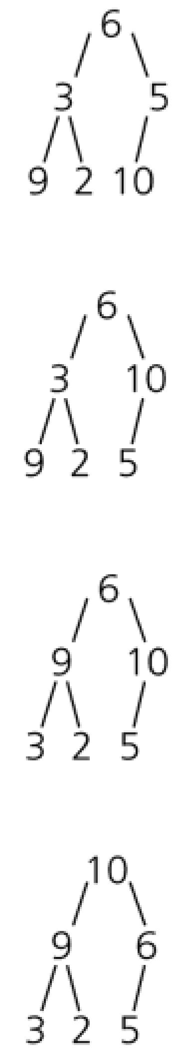



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

# Heapsort -- Building a heap from an array

| | Array `anArray` | | | | | |
|---|---|---|---|---|---|---|
| Original `anArray` | 6 | 3 | 5 | 9 | 2 | 10 |
| After `heapRebuild(anArray, 2, 6)` | 6 | 3 | 10 | 9 | 2 | 5 |
| After `heapRebuild(anArray, 1, 6)` | 6 | 9 | 10 | 3 | 2 | 5 |
| After `heapRebuild(anArray, 0, 6)` | 10 | 9 | 6 | 3 | 2 | 5 |

Tree representation of `anArray`



# Heapsort

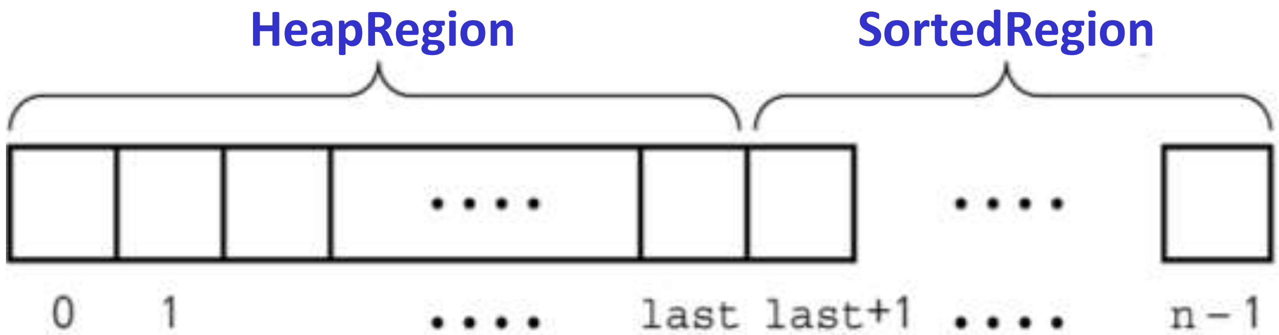
```
heapSort(inout anArray:ArrayType, in n:integer) {

   // build an initial heap
   for (index = (n/2) - 1 ;  index >= 0 ;  index--)
        heapRebuild(anArray, index, n)

   for (last = n-1 ;  last >0 ;  last--) {
        // invariant: anArray[0..last] is a heap,
        // anArray[last+1..n-1] is sorted and
        // contains the largest items of anArray.

        swap anArray[0] and anArray[last]

        // make the heap region a heap again
        heapRebuild(anArray, 0, last)
   }
}
```

O(n)

O(nlogn)

# Heapsort

- Heapsort partitions an array into two regions.



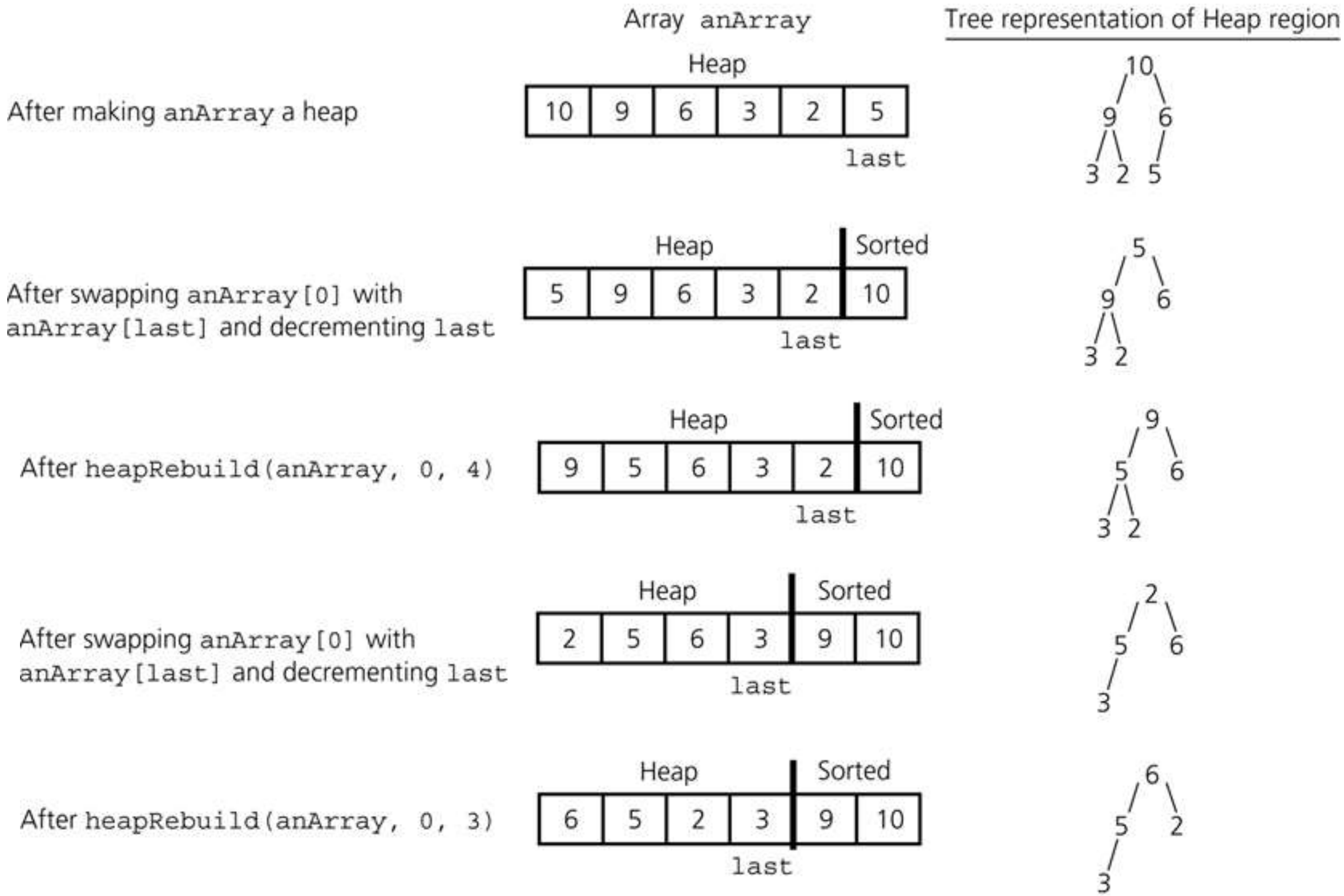

- Each step moves an item from the *HeapRegion* to *SortedRegion*.
- The invariant of the heapsort algorithm is:

  After the *k*th step,

  – The *SortedRegion* contains the *k* largest value and they are in sorted order.

  – The items in the *HeapRegion* form a heap.

# Heapsort -- Trace



# Heapsort -- Trace

After swapping anArray[0] with anArray[Last] and decrementing last

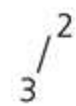
| Heap | | | Sorted | | |
|---|---|---|---|---|---|
| 3 | 5 | 2 | 6 | 9 | 10 |
| | | last | | | |


After rebuildHeap(anArray, 0, 2)

| Heap | | | Sorted | | |
|---|---|---|---|---|---|
| 5 | 3 | 2 | 6 | 9 | 10 |
| | | last | | | |

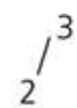

After swapping anArray[0] with anArray[last] and decrementing last

| Heap | | Sorted | | | |
|---|---|---|---|---|---|
| 2 | 3 | 5 | 6 | 9 | 10 |
| | last | | | | |




After heapRebuild(anArray, 0, 1)

| Heap | | Sorted | | | |
|---|---|---|---|---|---|
| 3 | 2 | 5 | 6 | 9 | 10 |
| | last | | | | |




After swapping anArray[0] with anArray[last] and decrementing last

| Heap | Sorted | | | | |
|---|---|---|---|---|---|
| 2 | 3 | 5 | 6 | 9 | 10 |
| last | | | | | |

Array is sorted

# Heapsort -- Analysis

- Heapsort is

  O(n log n) at the average case

  O(n log n) at the worst case

- Compared against *quicksort,*
  - *Heapsort* usually takes more time at the average case
  - But its worst case is also O(n log n).