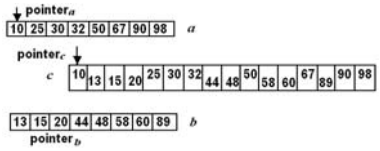
Algorithm MergeSort

- **John von Neumann** (*1945!*): a recursive divide-and-conquer approach
- Three basic steps:
  - If the number of items is 0 or 1, return
  - Recursively sort the first and the second halves separately
  - Merge two presorted halves into a sorted array
- Linear time merging $O(n)$ yields MergeSort time complexity $O(n \log n)$

1

$O(n)$ Merge of Sorted Arrays

if *a*[pointer*a*] < *b*[pointer*b*] **then** *c*[pointer*c*] ← *a*[pointer*a*];
pointer*a* ← pointer*a* + 1;
pointer*c* ← pointer*c* + 1
else *c*[pointer*c*] ← *b*[pointer*b*];
pointer*b* ← pointer*b* + 1;
pointer*c* ← pointer*c* + 1



2

Structure of MergeSort

begin MergeSort (an integer array a of size n)

1. Allocate a temporary array tmp of size n
2. **RecursiveMergeSort**(a, tmp, 0, $n-1$)

end MergeSort

Temporary array: to merge each successive pair of ordered subarrays a[left], …, a[centre] and a[centre+1, …, a[right] and copy the merged array back to a[left], …, a[right]

3

Recursive MergeSort

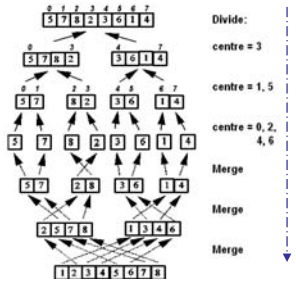
begin RecursiveMergeSort (an integer array a of size n; a temporary array tmp of size n; range: left, right)
  if left < right then centre ← $\lfloor (\text{left} + \text{right})/2 \rfloor$
  RecursiveMergeSort(a, tmp, left, centre);
  RecursiveMergeSort(a, tmp, centre + 1, right);
  Merge(a, tmp, left, centre + 1, right);
  end if
end RecursiveMergeSort

4

How MergeSort works

$2n$ comparisons for random data

n comparisons for sorted/reverse data



5

Analysis of mergeSort

+ $O(n \log n)$ best-, average-, and worst-case complexity because the merging is always linear
– Extra O(n) temporary array for merging data
– Extra work for copying to the temporary array and back
- Useful only for external sorting
- For internal sorting: QuickSort and HeapSort are much better

6

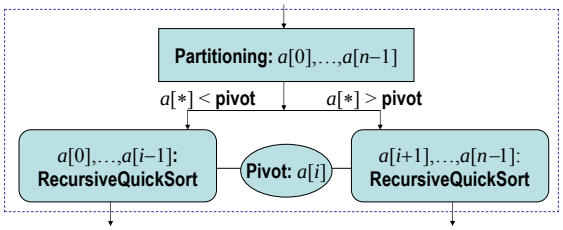
Algorithm QuickSort

- C. A. R. Hoare (1961): the divide-and-conquer approach
- Four basic steps:
  - If $n = 0$ or 1, return
  - Pick a **pivot** item
  - Partition the remaining items into the left and right groups with the items that are smaller or greater than the pivot, respectively
  - Return the **QuickSort** result for the left group, followed by the pivot, followed by the **QuickSort** result for the right group

7

Recursive QuickSort

- $T(n) = c \cdot n$ (pivot positioning) + $T(i)$ + $T(n - 1 - i)$



8

Analysis of QuickSort: the worst case $O(n^2)$

- If the pivot happens to be the largest (or smallest) item, then one group is always empty and the second group contains all the items but the pivot
- Time for partitioning an array: $c \cdot n$
- Running time for sorting: $T(n) = T(n - 1) + c \cdot n$
- "Telescoping" (recall the basic recurrences):

$$T(n) = c \cdot \frac{n(n+1)}{2}$$

9

Analysis of QuickSort: the average case $O(n \log n)$

- The left and right groups contain i and $n - 1 - i$ items, respectively; $i = 0, \ldots, n - 1$
- Time for partitioning an array: $c \cdot n$
- Average running time for sorting:

$$T(n) = \tfrac{2}{n}\big(T(0) + \ldots + T(n-2) + T(n-1)\big) + cn, \text{ or}$$

$$nT(n) = 2\big(T(0) + \ldots + T(n-2) + T(n-1)\big) + cn^2$$

$$(n-1)T(n-1) = 2\big(T(0) + \ldots + T(n-2)\big) + c(n-1)^2$$

10

Analysis of QuickSort: the average case $O(n \log n)$

$nT(n) - (n-1)T(n-1) \rightarrow nT(n) = (n+1)T(n-1) + 2cn$

"Telescoping": $\dfrac{T(n)}{n+1} \cong \dfrac{T(n-1)}{n} + \dfrac{2c}{n+1}$

Explicit form: $\dfrac{T(n)}{n+1} = \dfrac{T(0)}{1} + 2c\left(\dfrac{1}{2} + \dfrac{1}{3} + \dfrac{1}{4} + \ldots + \dfrac{1}{n+1}\right)$

$\approx 2cH_{n+1} \approx C \log n$

where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n + 0.577$

is the n^{th} harmonic number

11

Analysis of QuickSort: the choice of the pivot

- Never use the first a[low] or the last a[high] item!
- A reasonable choice is the middle item:

$$a\left[\text{middle} = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor\right]$$

where $\lfloor z \rfloor$ is an integer "floor" of the real value z

- **Good choice is the median of three:**

a[low], a[middle], a[high]

12

Pivot positioning in QuickSort: **low=0 , middle=4, high=9**

Data to be sorted										Description		
0	1	2	3	4	5	6	7	8	9	←Indices		
25	**8**	**2**	**91**	*70*	**50**	**20**	**31**	**15**	*65*	Initial array ***a***		
25	**8**	**2**	**91**	**65**	**50**	**20**	**31**	**15**	*70*	*i*=**MedianOfThree**(***a***,low,high) ; *p*=*a*[*i*]; **swap**(*i*, *a*[high−1])		
25	**8**	**2**	**91**	**15**	**50**	**20**	**31**	**65**	*70*	*i*	*j*	Condition
	8						**31**			1	7	*a*[*i*]< *p*> *a*[*j*]; *i*++
		2					**31**			2	7	*a*[*i*]< *p*> *a*[*j*]; *i*++

Pivot positioning in QuickSort: **low=0 , middle=4, high=9**

25	**8**	**2**	**91**	**15**	**50**	**20**	**31**	**65**	*70*	*i*	*j*	Condition
			91 **31**				**31** **91**	**65**		3	7	*a*[*i*] ≥ *p* > *a*[*j*]; swap; *i*++; *j*−−
				15		**20**		**65**		4	6	*a*[*i*]< *p* > *a*[*j*]; *i*++
					50	**20**		**65**		5	6	*a*[*i*]< *p* > *a*[*j*]; *i*++
						20		**65**		6	6	*a*[*i*]< *p* > *a*[*j*]; *i*++
								65		7	6	*i* > *j*; **break**
25	**8**	**2**	**31**	**15**	**50**	**20**	**65**	**91**	**70**	**swap**(*a*[*i*], *p* = *a*[high−1])		

Data selection: QuickSelect

- Goal: find the k-th smallest item of an array a of size n
- If k is fixed (e.g., the median), then selection should be faster than sorting
- Linear average-case time $O(n)$: by a small change of QuickSort
- Basic Recursive QuickSelect: to find the k-th smallest item in a subarray:

  (a[low], a[low + 1], …, a[high])

  such that $0 \le$ low $\le k-1 \le$ high $\le n-1$

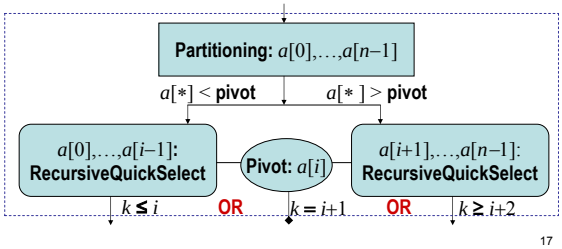
Recursive QuickSelect

- If high = low = $k-1$: return $a[k-1]$
- Pick a median-of-three pivot and split the remaining elements into two disjoint groups just as in QuickSort:

  a[low], …, $a[i-1] < a[i] =$ pivot $< a[i+1]$, …, a[high]
- Recursive calls:
  - $k \le i$: **RecursiveQuickSelect**(a, low, $i-1$, k)
  - $k = i+1$: **return** $a[i]$
  - $k \ge i+2$: **RecursiveQuickSelect**(a, $i+1$, high, k)

Recursive QuickSelect

The average running time $T(n) = cn$ (partitioning an array) + the average time for selecting among i or $(n-1-i)$ elements where i varies from 0 to $n-1$



QuickSelect: low=0, high=$n-1$

- $T(n) = c \cdot n$ (splitting the array) + { $T(i)$ OR $T(n-1-i)$ }
- Average running time:

$$T(n) = \frac{1}{n}\left(T(0) + \ldots + T(n-2) + T(n-1)\right) + cn$$

or $n\,T(n) = T(0) + \ldots + T(n-2) + T(n-1) + cn^2$

$(n-1)\,T(n-1) = T(0) + \ldots + T(n-2) + c(n-1)^2$

$nT(n) - (n-1)\,T(n-1) \to T(n) - T(n-1) \cong 2c$

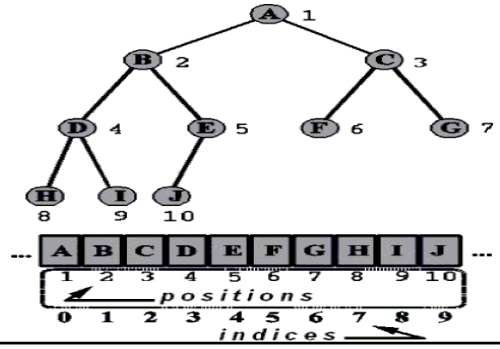
or $T(n)$ is $O(n)$

Algorithm HeapSort

- J. W. J. Williams (1964): a special binary tree called **heap** to obtain an $O(n \log n)$ worst-case sorting
- Basic steps:
  - Convert an array into a heap in linear time $O(n)$
  - Sort the heap in $O(n \log n)$ time by deleting n times the maximum item because each deletion takes the logarithmic time $O(\log n)$

19

Complete Binary Tree: linear array representation



20

Complete Binary Tree

- A complete binary tree of the height h contains between 2^h and $2^{h+1}-1$ nodes
- A complete binary tree with the n nodes has the height $\lfloor \log_2 n \rfloor$
- Node positions are specified by the level-order traversal (the root position is 1)
- If the node is in the position p then:
  - the parent node is in the position $\lfloor p/2 \rfloor$
  - the left child is in the position $2p$
  - the right child is in the position $2p + 1$

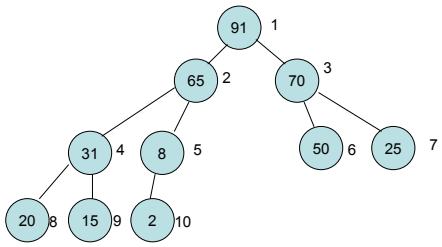
21

Binary Heap

- A heap consists of a complete binary tree of height h with numerical keys in the nodes
- **The defining feature of a heap**:
  the key of each parent node is **greater than** or **equal to** the key of any child node
- The root of the heap has the maximum key

22

Complete Binary Tree: linear array representation



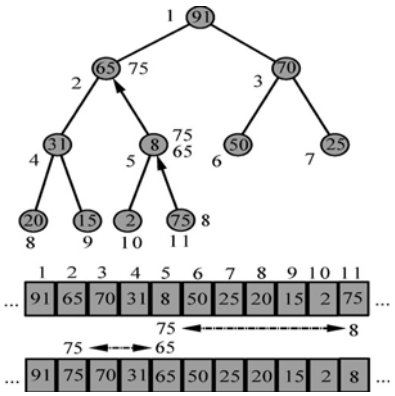
23

Binary Heap: insert a new key

- Heap of k keys → heap of $k + 1$ keys
- Logarithmic time $O(\log k)$ to insert a new key:
  - Create a new leaf position $k + 1$ in the heap
  - **Bubble** (or **percolate**) the new key up by swapping it with the parent if the latter one is smaller than the new key

24

Binary Heap:
an example of
inserting a key



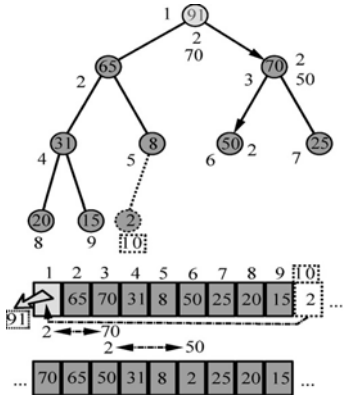
25

Binary Heap: delete the maximum key

- Heap of k keys → heap of $k-1$ keys
- Logarithmic time $O(\log k)$ to delete the root (or maximum) key:
  - Remove the root key
  - Delete the leaf position k and move its key into the root
  - Bubble (percolate) the root key down by swapping with the largest child if the latter one is greater

26

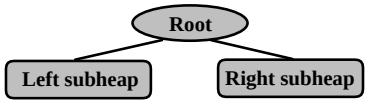
Binary Heap:
an example
of deleting the
maximum key



27

Linear Time Heap Construction

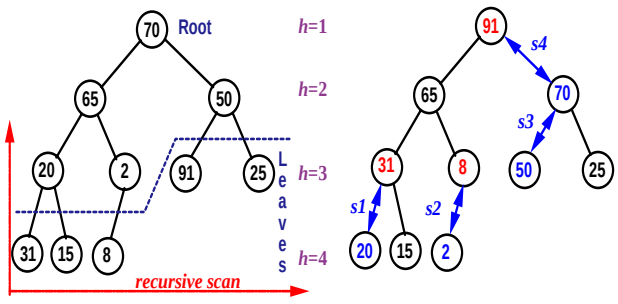
- n insertions take $O(n \log n)$ time.
- Alternative $O(n)$ procedure uses a recursively defined heap structure:



  - form recursively the left and right subheaps
  - percolate the root down to establish the heap order everywhere

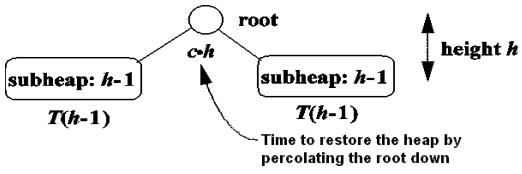
28

Heapifying Recursion



29

Time to build a heap



$$T(h) = 2T(h-1) + c \cdot h; \qquad T(0) = 0$$

$$\rightarrow \quad T(h) = c \cdot \left(2^{h+1} - h - 1\right)$$

30

Linear time heap construction: non-recursive procedure

- Nodes are percolated down in reverse level order
- When the node p is processed, its descendants will have been already processed.
- Leaves need not to be percolated down.
- Worst-case time $T(h)$ for building a heap of height h:

  $T(h) = 2T(h-1) + ch \rightarrow T(h) = O(2^h)$

  – Form two subheaps of height $h-1$

  – Percolate the root down a path of length at most h

31

Time to build a heap

- A heap of the height h has $n = 2^{h-1} \ldots 2^h - 1$ nodes so that the height of the heap with n items: $h = \lceil \log_2 n \rceil$
- Thus, $T(h) = O(2^h)$ yields the linear time $T(n) = O(n)$

$$\int_0^N (N-x)e^x dx = e^N - N - 1$$

$$\int_0^N xe^x dx = (N-1)e^N + 1$$

Two integral relationships helping to derive the above (see Slide 12) and the like discrete formulas

32

Time to build a heap

$$T(h) = 2T(h-1) + c \cdot h$$

$$2T(h-1) = 2^2 T(h-2) + 2c \cdot (h-1)$$

$$\ldots \ldots \ldots$$

$$2^{h-2} T(2) = 2^{h-1} T(1) + 2^{h-2} c \cdot 2$$

$$2^{h-1} T(1) = 2^h T(0) + 2^{h-1} c \cdot 1 = 2^{h-1} c \cdot 1$$

$$T(h) = c \cdot \left(1 \cdot 2^{h-1} + 2 \cdot 2^{h-2} + \ldots + (h-2) \cdot 2^2 + (h-1) \cdot 2^1 + h \cdot 2^0\right)$$
$$= c \cdot \left(2^{h+1} - h - 1\right)$$

33

Steps of HeapSort

p/i	$1/0$	$2/1$	$3/2$	$4/3$	$5/4$	$6/5$	$7/6$	$8/7$	$9/8$	$10/9$
a	70	65	50	20	2	91	25	31	15	8
HEAPIFY					8					2
				31				20		
			91			50				
	91		70							
h	**91**	**65**	**70**	**31**	**8**	**50**	**25**	**20**	**15**	**2**

34

Steps of HeapSort

a_1	2	65	70	31	8	50	25	20	15	91
Restore the heap (R.h.)	70		2							
			50			2				
H_9	70	65	50	31	8	2	25	20	15	
a_2	15	65	50	31	8	2	25	20	70	91
R.h.	65	15								
		31		15						
				20				15		
h_8	65	31	50	20	8	2	25	15		

35

Steps of HeapSort

a_3	15	31	50	20	8	2	25	85	70	91
R.h.	50		15							
			25				15			
h_7	50	31	25	20	8	2	15			
a_4	15	31	25	20	8	2	50	85	70	91
R.h.	31	15								
		20		15						
h_6	31	20	25	15	8	2				

36

Steps of HeapSort

a_5	2	20	25	15	8	31	50	65	70	81
R. h.	25		2							
h_5	25	20	2	15	8					
a_6	8	20	2	15	25	31	50	65	70	81
R. h.	20	8								
		15		8						
h_4	20	15	2	8						

37

Steps of HeapSort

a_7	8	15	2	20	25	31	50	65	70	81
R. h.	15	8								
h_3	15	8	2							
a_8	2	8	15	20	25	31	50	65	70	81
R. h.	8	2								
h_2	8	2								
a_9	2	8	15	20	25	31	50	65	70	81

sorted array

38

Lower Bound for Sorting Complexity

- Each algorithm that sorts by comparing only pairs of elements must use at least

$$\lceil \log_2(n!) \rceil \cong n \log_2 n - 1.44n$$

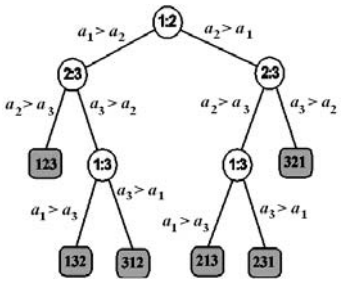
comparisons in the worst case (that is, for some "worst" input sequence) and in the average case.

- Stirling's approximation of the factorial ($n!$):

$$1 \cdot 2 \cdot \ldots \cdot n \equiv n! \geq \left(\frac{n}{e}\right)^n \sqrt{2\pi n} \approx 2.5 n^{n+0.5} e^{-n}$$

39

Decision Tree for Sorting n Items



Decision tree for n =3:

- i:j - a comparison of a_i and a_j
- ijk - a sorted array $(a_i a_j a_k)$
- $n!$ permutations → $n!$ leaves

Sorting in descending order of the numbers

40

Decision Tree for Sorting n Items

- Decision tree for n=3: an array $\boldsymbol{a} = \{a_1, a_2, a_3\}$
- Example: {35, 10, 17}
  - Comparison 1:2 $(35 > 10)$ → left branch $a_1 > a_2$
  - Comparison 2:3 $(10 < 17)$ → right branch $a_2 < a_3$
  - Comparison 1:3 $(35 > 17)$ → left branch $a_1 > a_3$
- Sorted array 132 → $\{a_1=35, a_3=17, a_2=10\}$

41

Decision Tree

- Decision tree of height h has $L_h \leq 2^h$ leaves
- Mathematical induction:
  - $h = 1$: the tree of height 1 has $L_1 \leq 2^h$ leaves
  - $h-1$ → h: let the tree of height $h-1$ have $L_{h-1} \leq 2^{h-1}$ leaves; the tree of height h consists of a root and two subtrees at most of height $h-1$. Thus,

$$L_h = L_{h-1} + L_{h-1} \leq 2^{h-1} + 2^{h-1} = 2^h$$

42

Worst-Case Complexity of Sorting

- The lower bound for the least height h of a decision tree for sorting by pairwise comparisons which provides $L_h = 2^h \geq n!$ leaves is

$$h \geq \log_2(n!) \cong n \log_2 n - 1.44\, n$$

- Thus, the worst-case complexity of the sorting is at least $O(n \log n)$

43

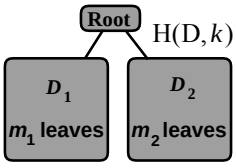
Average-Case Sorting Complexity

- Each n-element decision tree has an average height at least $\log(n!) \geq n \log n$
- Let $H(D,k)$ be the sum of heights for all k leaves of a tree D and $H(k) = \min_D H(D,k)$ denote the minimum sum of heights
- Math induction to prove that $H(k) \geq k \log k$
  - $k = 1$: Obviously, $H(1) = 0$
  - $k-1 \rightarrow k$: Let $H(m) \geq m \log m$, $m < k$

44

Average-Case Sorting Complexity

- The tree D with k leaves contains 2 subtrees, D_1 with $m_1 < k$ leaves and D_2 with $m_2 < k$ leaves just under the root ($m_1 + m_2 = k$):



$$H(D,k) = k + H(D_1,m_1) + H(D_2,m_2)$$

because the link to the root adds 1 to each leaf's height

45

Average-Case Sorting Complexity

The minimum height : $H(k) = k + \min_{m_1+m_2=k} \{H(m_1) + H(m_2)\}$

By the induction assumption :

$$H(k) \leq k + \min_{m_1+m_2=k} \{m_1 \log m_1 + m_2 \log m_2\}$$

The minimum is reached by symmetry for $m_1 = m_2 = \frac{k}{2}$:

$$H(k) \leq k + 2\left(\tfrac{k}{2}\right)\log\left(\tfrac{k}{2}\right) = k \log k$$

Thus the average height of the decision tree with at least $n!$ leaves is : $\left(H(n!)/n!\right) \leq \log_2 n! \cong n \log_2 n - 1.44\, n$

46

Data Search

- Data record → Specific key
- Goal: to find all records with keys matching a given search key
- Purpose:
  - to access information in the record for processing, or
  - to update information in the record, or
  - to insert a new record or to delete the record

47

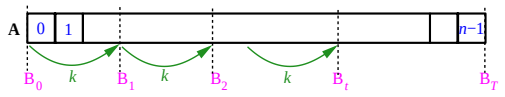
Types of Search

- **Static search**: stored data is not changed
  - Given an integer search key X, return either the position of X in an array A of records or an indication that it is not present without altering the array A
  - If X occurs more than once, return any occurrence
- **Dynamic search**: the data may be inserted or deleted

48

Sequential and Jump Search

- Sequential search is the only one for an **unsorted** array
- Successful / unsuccessful search: the $O(n)$ worst-case and average-case complexity
- Jump search $O(n^{0.5})$ in a **sorted** array **A** of size n: $T = \lceil n/k \rceil$ jumps of length k to the points $B_t = \min\{t \cdot k, n\}$ and the sequential search among k items in a t-th part such that $B_{t-1} \le$ **key** $\le B_t - 1$; $t = 1,\ldots,T$



Jump Search $O(n^{0.5})$

Worst-case complexity:

$$T(n,k) = \frac{n}{k} + k$$

Minimum complexity : for $k = \sqrt{n}$

$$T(n) = \min_k\{T(n,k)\} = 2\sqrt{n}$$

because $\frac{d}{dk} T(n,k) = -\frac{n}{k^2} + 1 = 0 \;\rightarrow\; n = k^2$

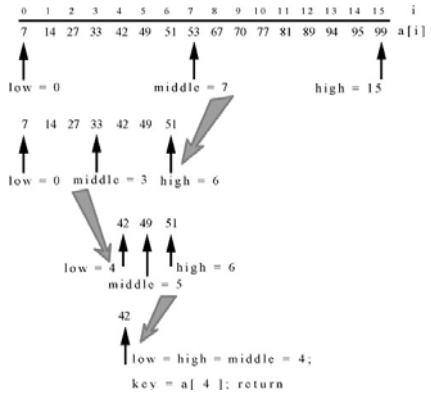
Binary Search $O(\log n)$

- Ordered array: $\mathrm{key}_0 < \mathrm{key}_1 < \ldots < \mathrm{key}_{n-1}$
- Compare the search key with the record key_i at the middle position $i = \lfloor (n-1)/2 \rfloor$
  - if key = key_i, return i
  - if key < key_i or key < key_i, then it must be in the 1st or in the 2nd half of the array, respectively
- Apply the previous two steps to the chosen half of the array iteratively (**repeating halving principle**)

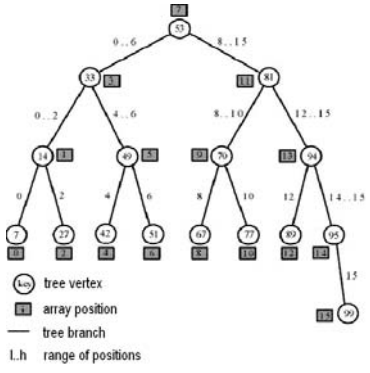
Implementation of Binary Search

```
begin BinarySearch (an integer array a of size n, a search key)
    low ← 0;   high ← n – 1
    while low ≤ high do
        middle ← ⌊( low + high ) / 2⌋
        if a[ middle ] < key then low ← middle + 1
        else if a[ middle ] > key then high← middle – 1
        else return middle end if
    end while
    return ItemNotFound
end BinarySearch
```


Binary search: detailed analysis



Comparison structure: the binary (search) tree



Worst-Case Complexity $O(\log n)$ of Binary Search

- Let $n = 2^k - 1; k = 1,2,\ldots,$ then the binary tree is complete (each internal node has 2 children)
- The tree height is $k-1$
- Each tree level l contains 2^l nodes for $l = 0$ (the root), $1, \ldots, k-2, k-1$ (the leaves)
- $l + 1$ comparisons to find a key of level l
- **The worst case**: $k = \log_2(n+1)$ comparisons

55

Average-Case Complexity $O(\log n)$ of Binary Search

- Let $C_i = l + 1$ be the number of comparisons to find key_i of level l; $i = 0, \ldots, n-1$; $l = 0, \ldots, k-1$
- Average number: $\overline{C} = \frac{1}{n}(C_0 + C_1 + \ldots + C_{n-1})$
- There are 2^l nodes at the level l, so that:

$$C_0 + C_1 + \ldots + C_{n-1} \equiv S_{k-1} = 1 \cdot 2^0 + \ldots + k \cdot 2^{k-1}$$

- By math induction: $S_{k-1} = 1 + (k-1)\,2^k$, so that

$$\overline{C} = \frac{1}{n}\left(1 + (k-1)2^k\right) = \frac{n+1}{n}\log_2(n+1) - 1$$

56