Data Structures and Algorithms

Week 5

Binary Search Trees

- Will discuss binary search tree as an underlying storage of a sorted map.
- Each internal position *p* in a binary search tree stores (*k, v*) pair.

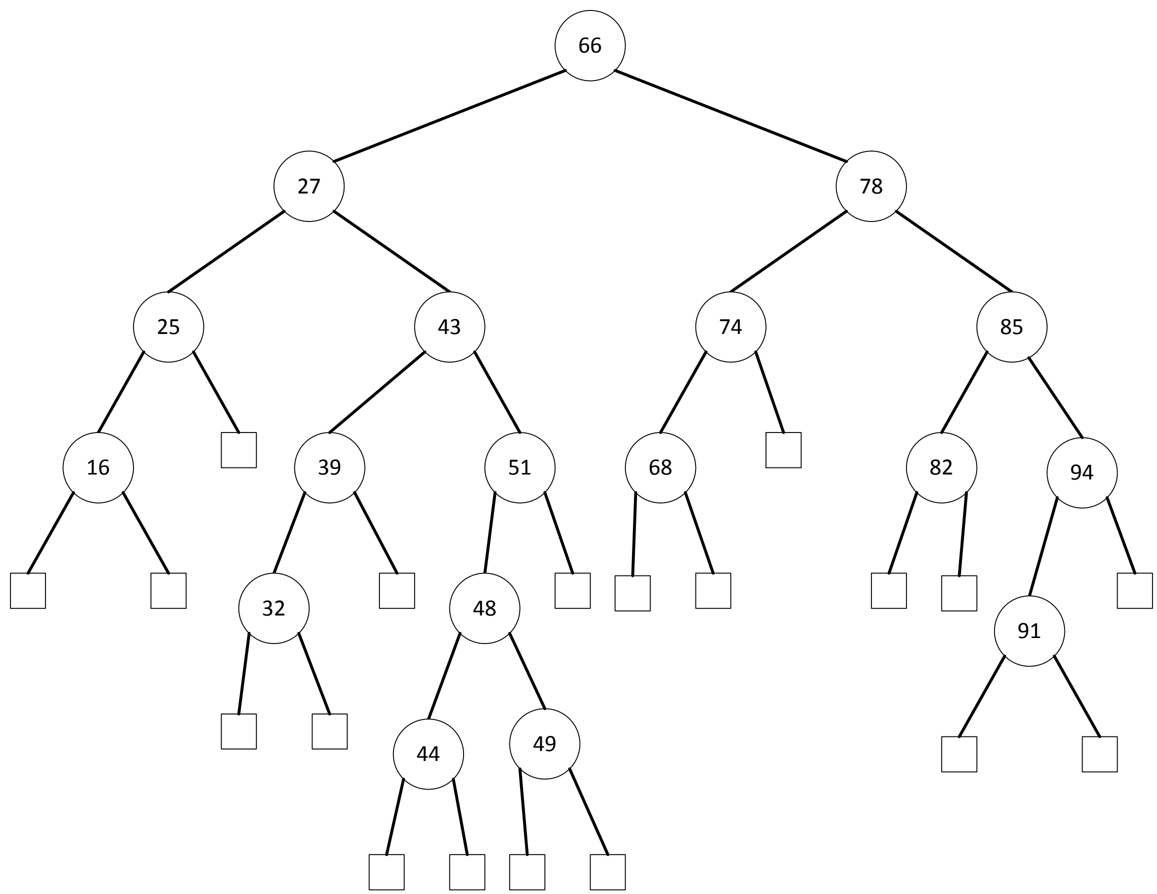
- Binary search tree is a *proper binary tree* with the following properties:

For each internal position *p* with entry (*k*, *v*) pair,

- Keys stored in the left subtree of *p* are less than *k*.
- Keys stored in the right subtree of *p* are greater than *k*.

Binary Search Trees

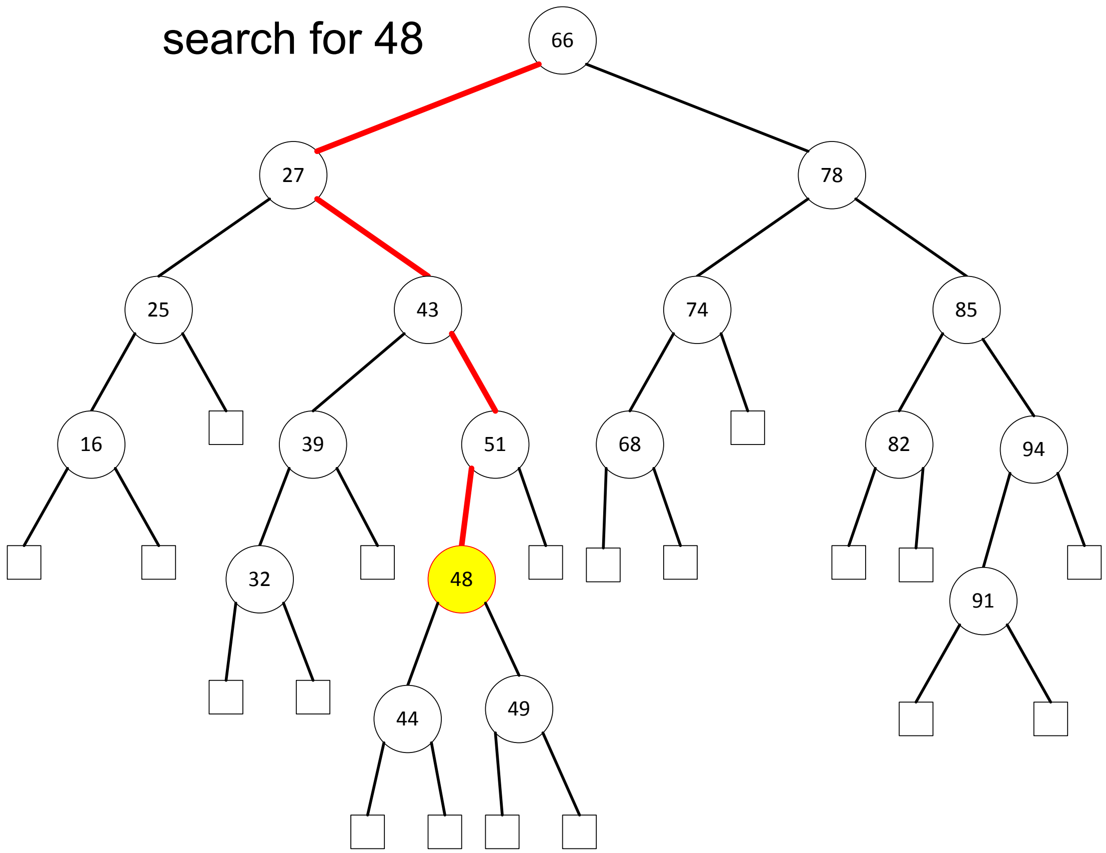
- Example (only keys are shown):



Binary Search Trees

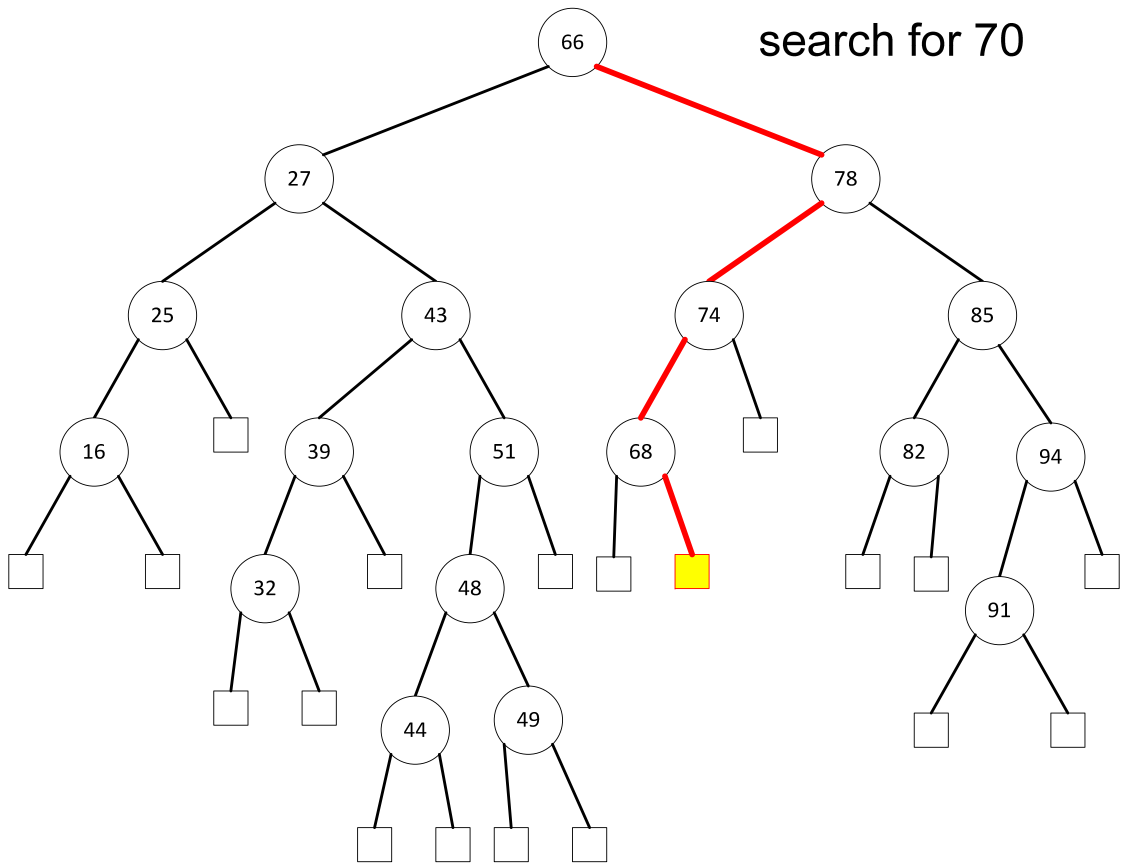
- Search (successful search)

search for 48



Binary Search Trees

- Search (unsuccessful search)



Binary Search Trees

- Search pseudocode

```
Algorithm TreeSearch(p, k)
if p is external then               // unsuccessful search
  return p
else if k == key(p)                 // successful search
  return p
else if k < key(p)
  return TreeSearch(left(p), k)     // recurse on left subtree
else
  return TreeSearch(right(p), k)    // recurse on right subtree
```

- Running time: $O(h)$

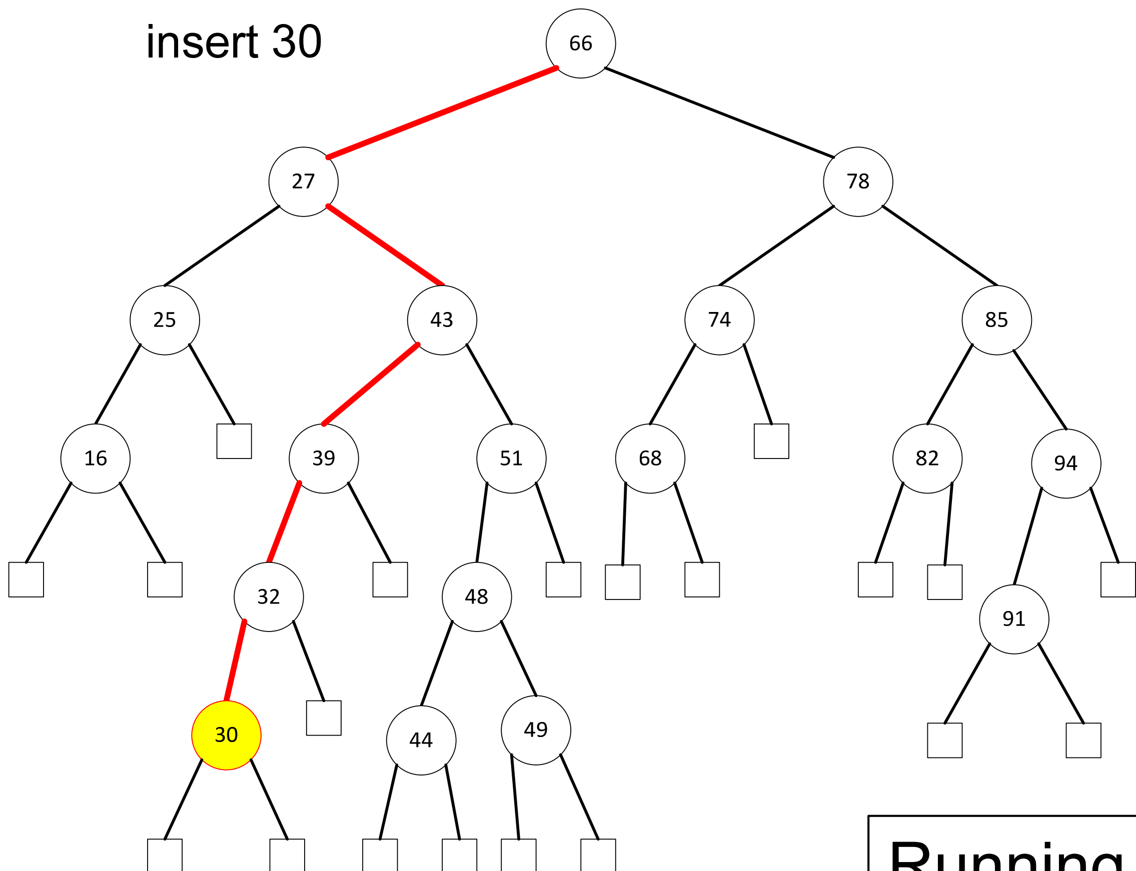
Binary Search Trees

- Inserting an entry with (k, v)
  - Perform a search operation.
  - If an entry with key *k* is found (i.e., successful search), the existing value is replaced with the new value *v*.
  - If there is no entry with key *k*, then we add an entry at the leaf node where the unsuccessful search ended up.

Binary Search Trees

- Insert illustration

insert 30



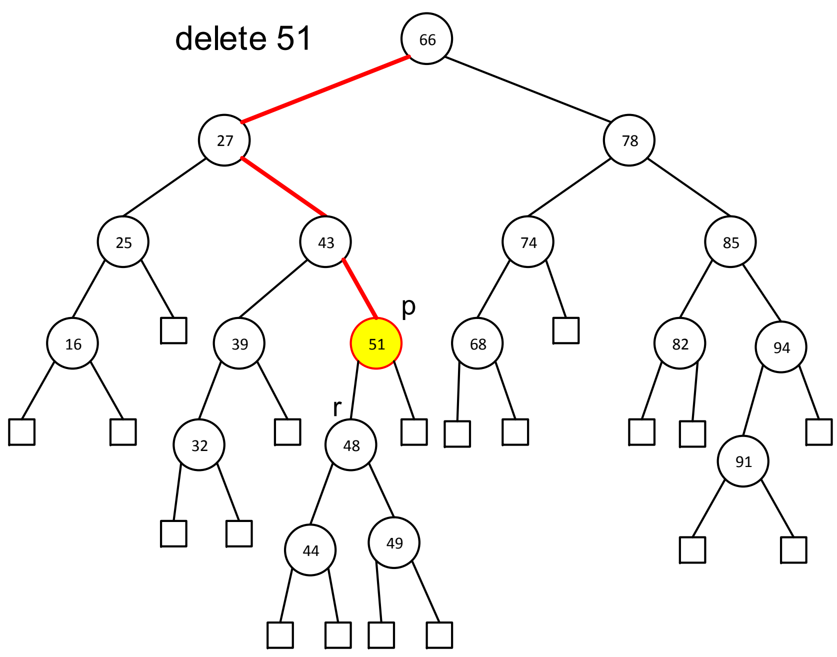
Running time: O(h)

Binary Search Trees

- Deleting an entry with (k, v)
  - Slightly more complex
  - Perform search
    - If we reach a leaf node, do nothing
    - If we find the entry at position *p*
      - Case 1: at most one child of *p* is an internal node
      - Case 2: *p* has two children, both of which are internal

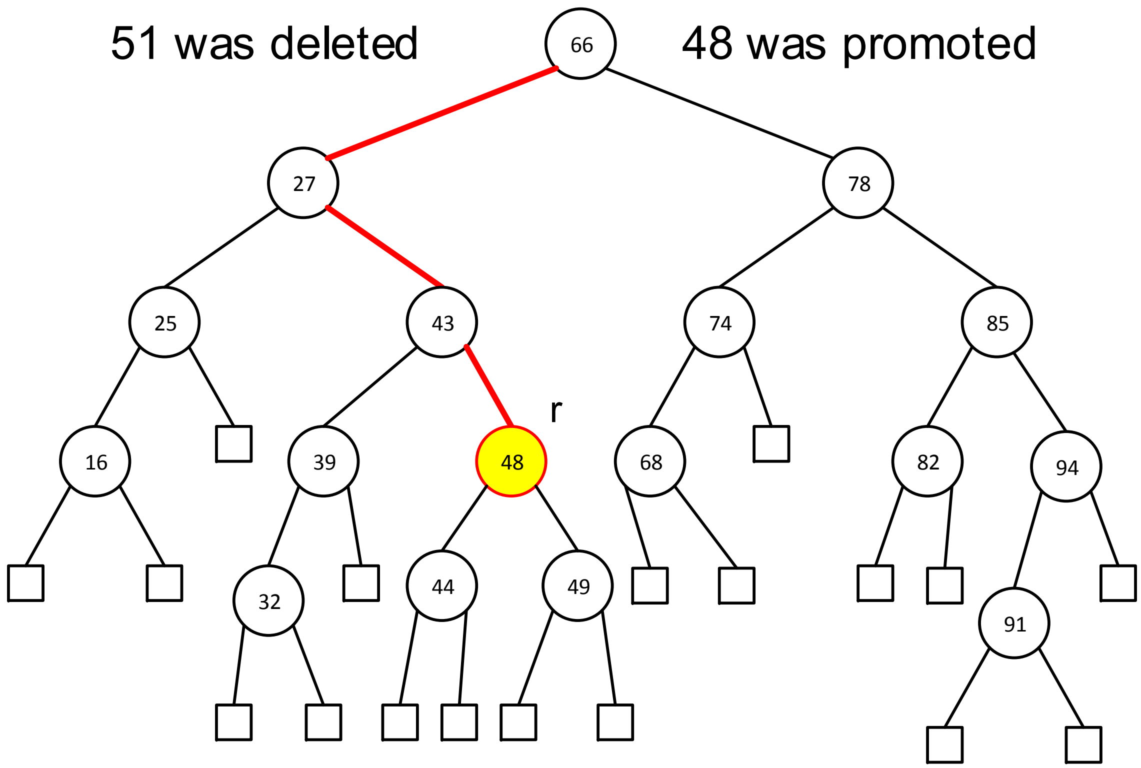
Binary Search Trees

- Deletion Case 1
  - If both children are leaf nodes, then *p* is replaced with a leaf node.
  - If *p* has one internal-node child, then that child node replaces *p*



Binary Search Trees

- Deletion Case 1
  - If *p* has one internal-node child (continued)



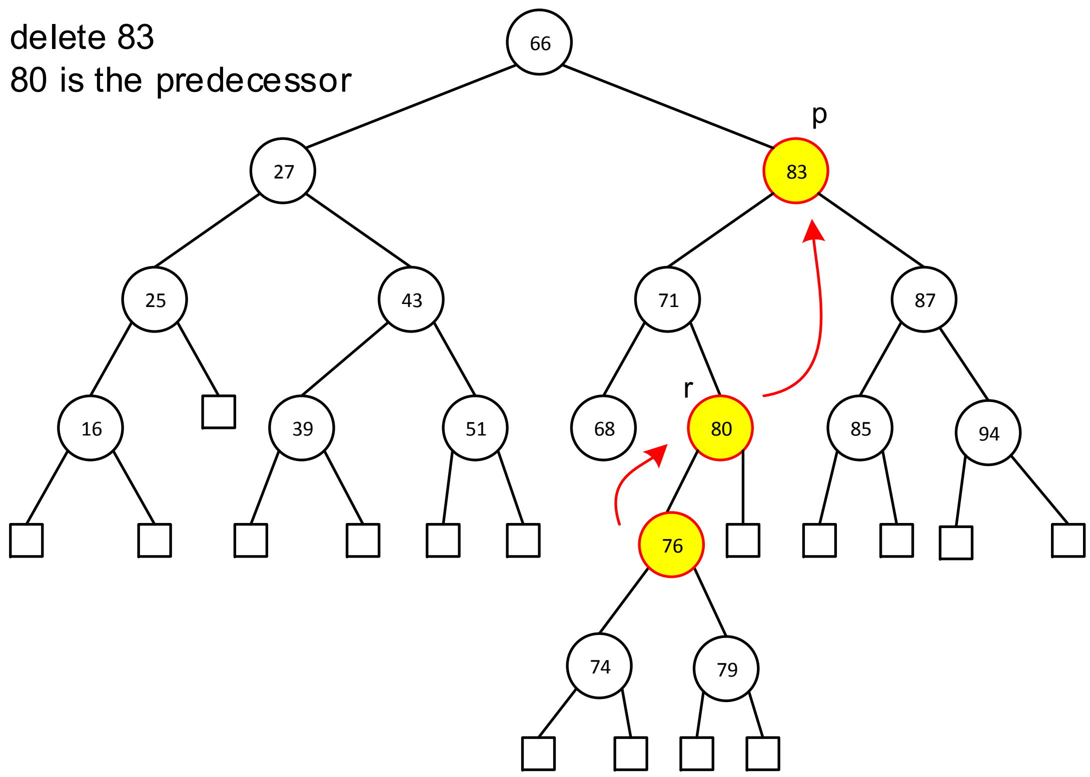
Binary Search Trees

- Deletion Case 2
  - First, we find the node *r* that has the largest key that is strictly less than *p*’s key. This node is called the *predecessor* of *p* in the ordering of keys, which is the rightmost node in *p*’s left subtree.
  - We let *r* replace *p*.
  - Since *r* is the rightmost node in *p*’s left subtree, it does not have a right child. It has only a left child.
  - The node *r* is removed and the subtree rooted at *r*’s left child is promoted to *r*’s position.

Binary Search Trees

- Deletion Case 2

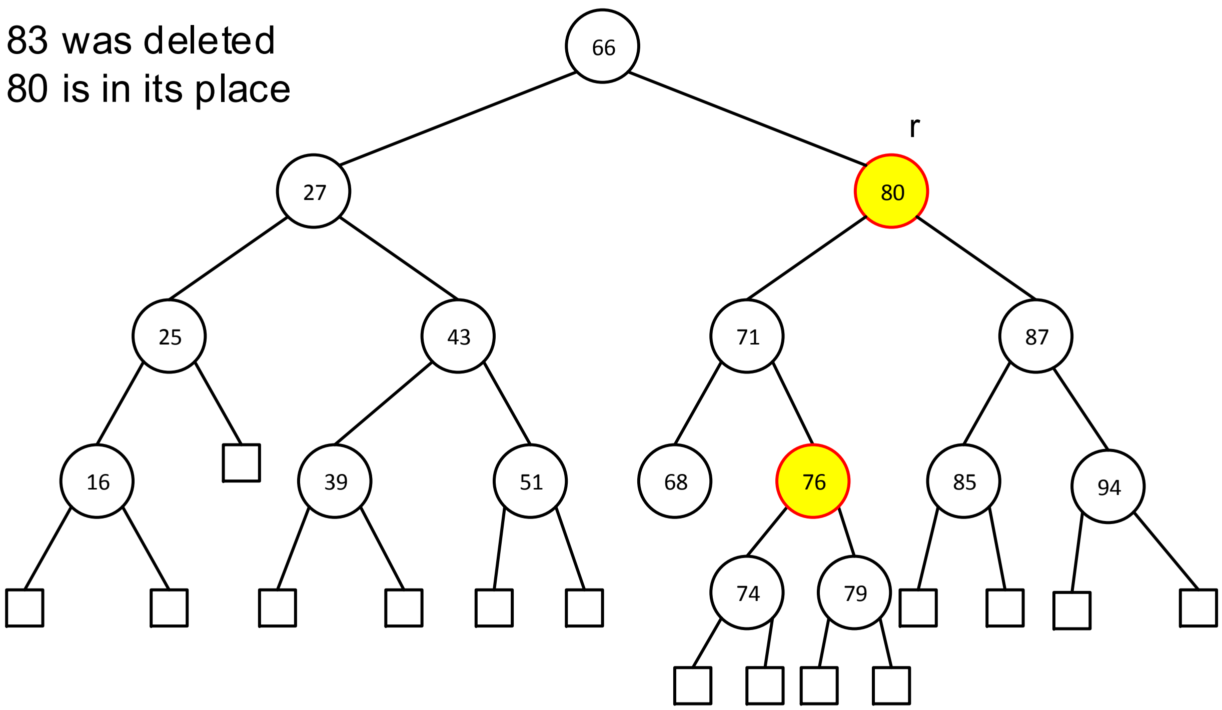
delete 83
80 is the predecessor



Binary Search Trees

- Deletion Case 2

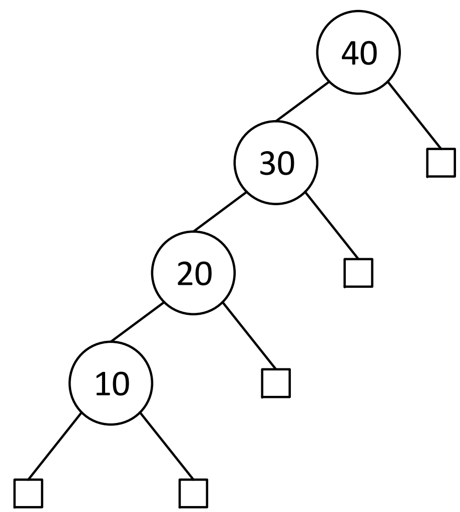
83 was deleted
80 is in its place



- Running time: O(h)

Binary Search Trees

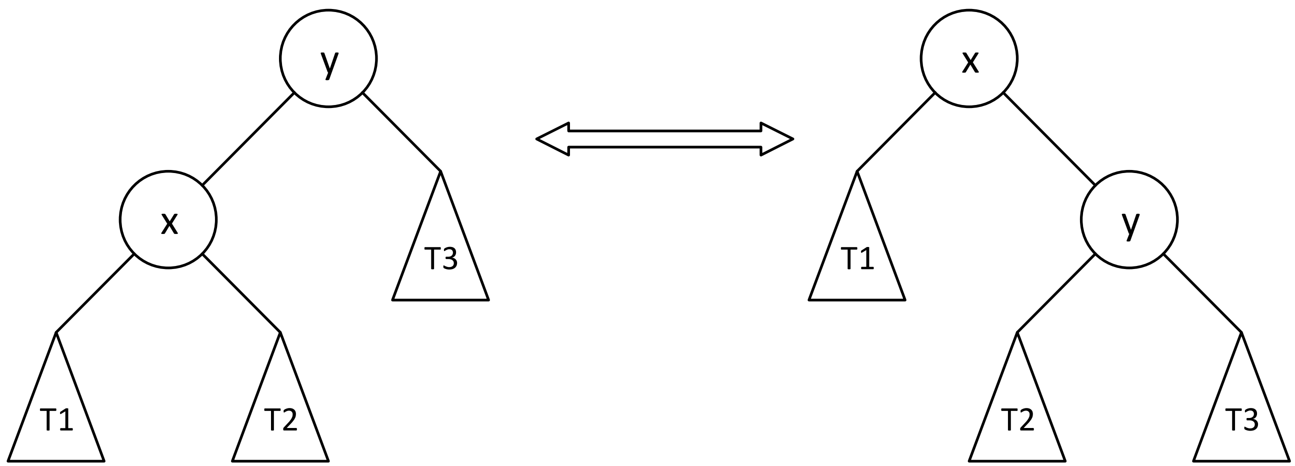
- Most binary search tree operations run in *O*(h).
- In the worst case, a tree is just a linked list. In this case, running times are *O*(n).



- To guarantee *O*(h), a tree needs to be balanced.

Balanced Search Trees

- When a binary search tree is unbalanced, it is necessary to *rebalance* the tree.
- Primary operation for rebalancing a binary search tree is *rotation*.



- Can rotate in either direction.
- Binary search tree property is maintained after rotation.

Balanced Search Trees

- A *trinode restructuring* performs a broader rebalancing.
- It involves three positions: *x*, *y*, and *z*
- *y* is the parent of *x* and *z* is the grandparent of *x*.
- Goal: Restructure the subtree rooted at *z* to reduce the path length from *z* to *x* and its subtrees.

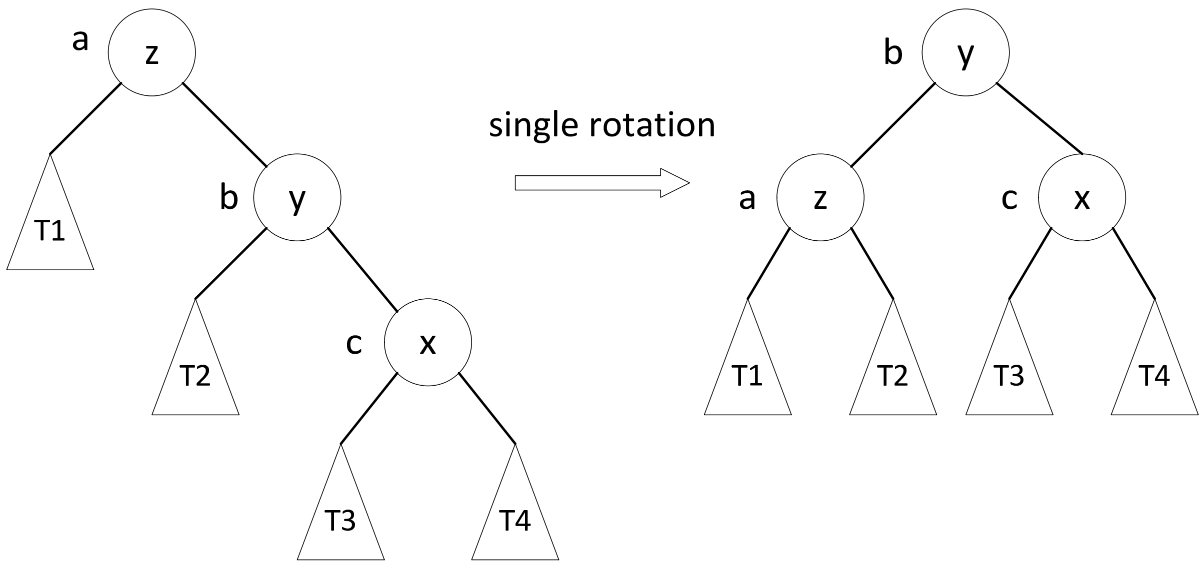
- Use secondary labels, *a*, *b*, and *c*, for the three positions such that *a* comes before *b* and *b* comes before *c* in an inorder tree traversal of the tree.
- There are four different configurations. This secondary labels allow us to describe the trinode restruring operations in a uniform way.

Balanced Search Trees

- Outline of the algorithm:
  - (T_1, T_2, T_3, T_4) are left-to-right listing of subtrees of *x*, *y*, and *z*.
  - The subtree rooted at *z* is replaced with the subtree rooted at *b*.
  - Make *a* the left child of *b*.
  - Make T_1 and T_2 the left and right subtree of *a*, respectively.
  - Make *c* the right child of *b*.
  - Make T_3 and T_4 the left and right subtree of *c*, respectively.

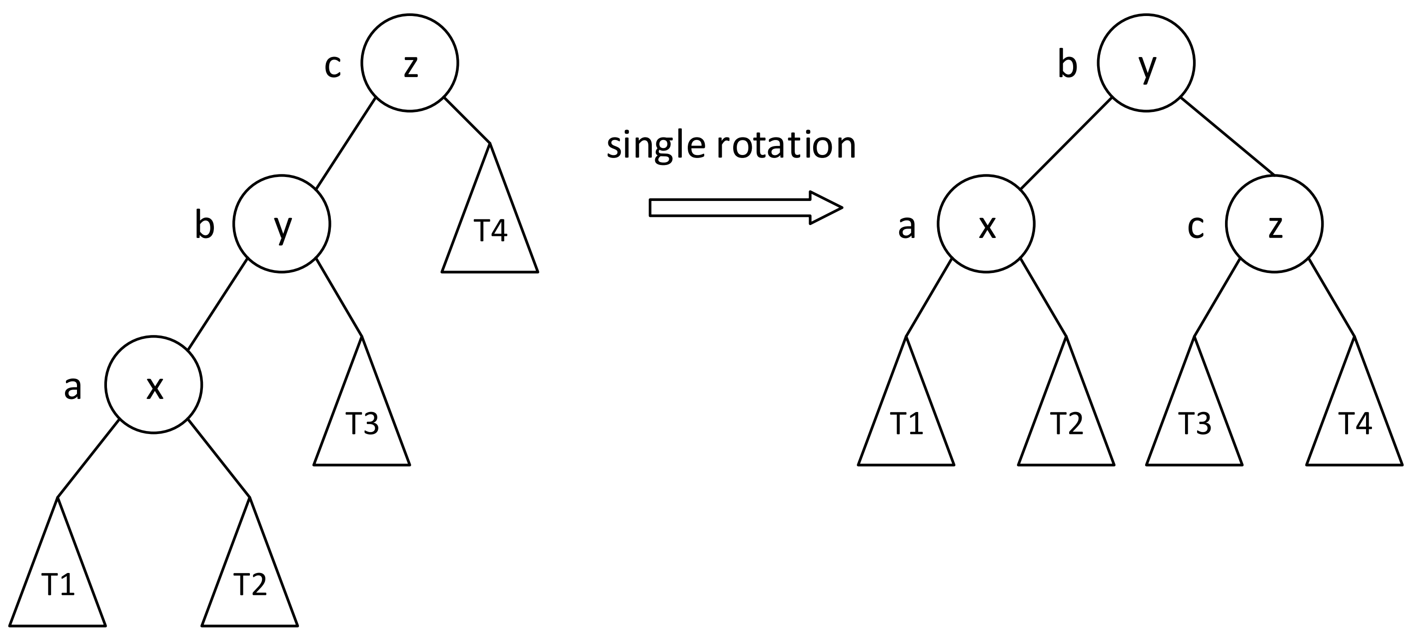
Balanced Search Trees

- Trinode restructuring: single rotation 1



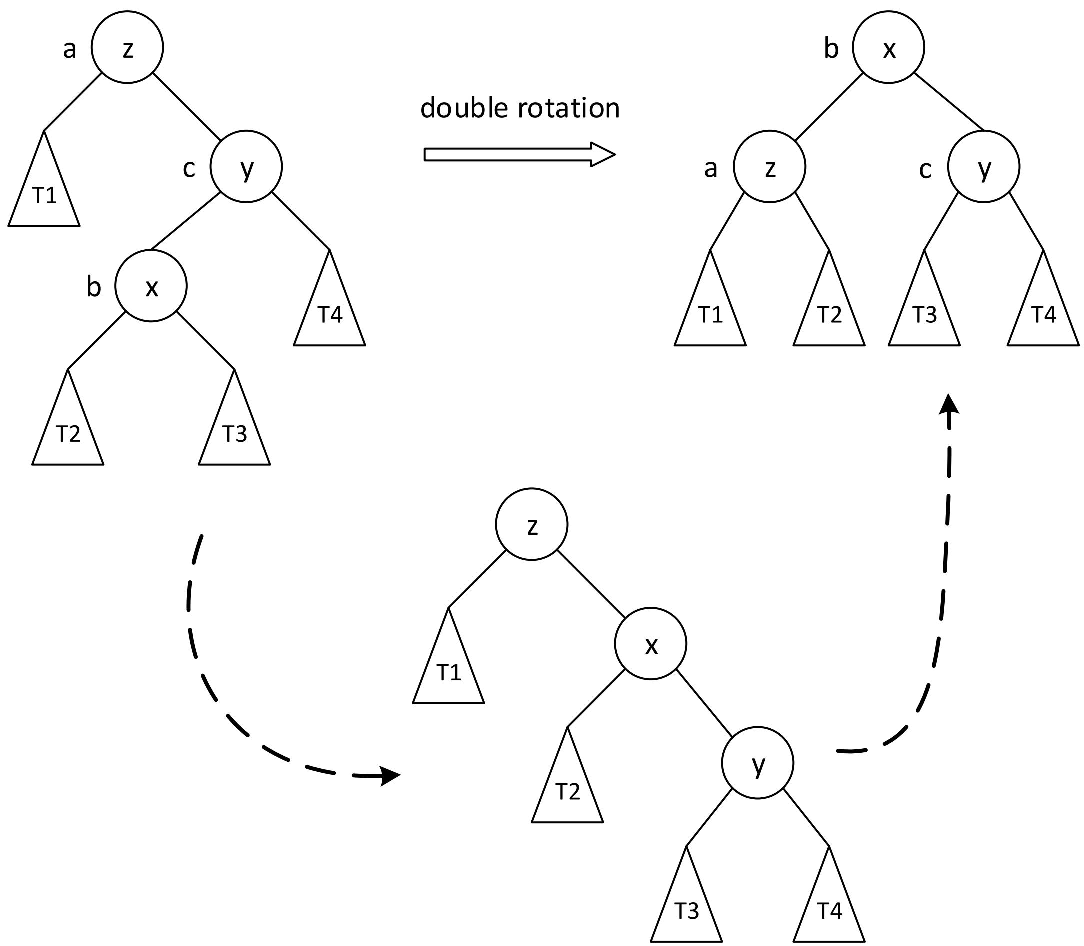
Balanced Search Trees

- Trinode restructuring: single rotation 2



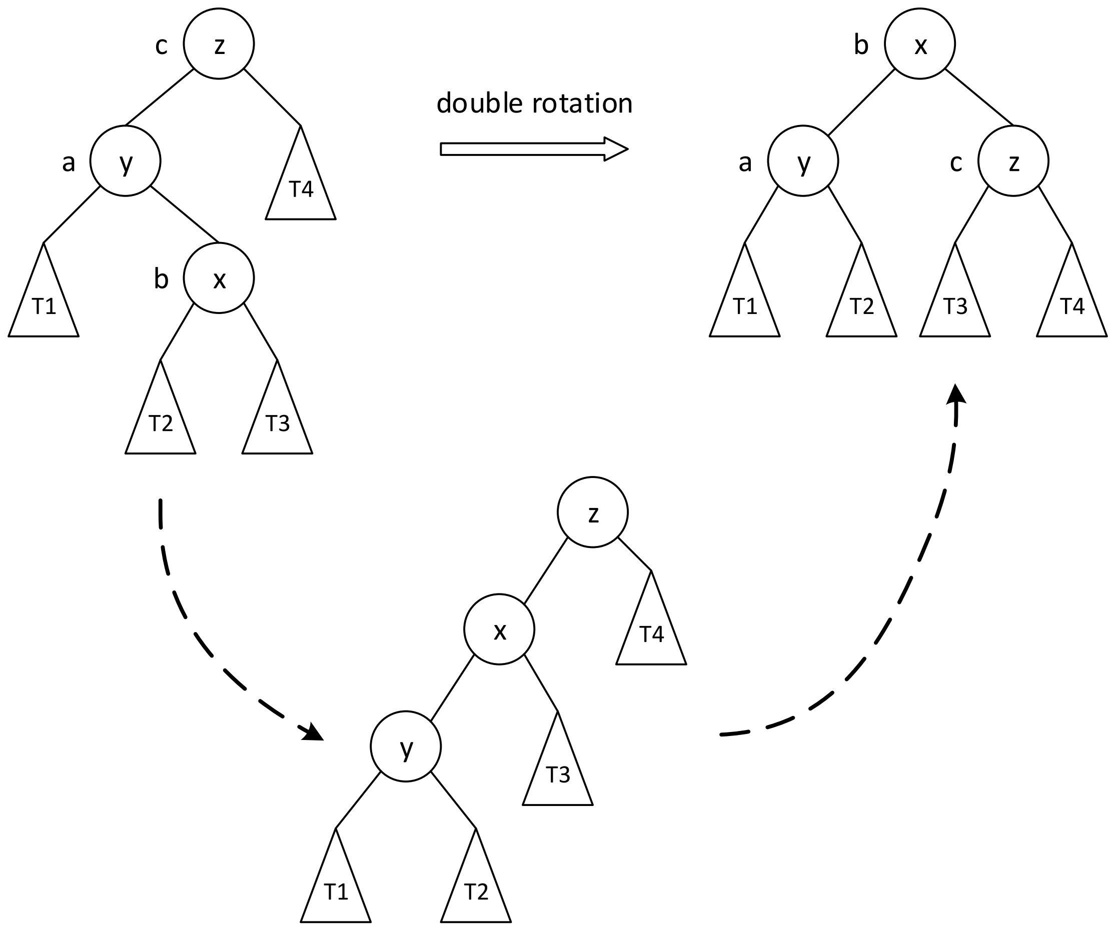
Balanced Search Trees

- Trinode restructuring: double rotation 1



Balanced Search Trees

- Trinode restructuring: double rotation 2



AVL Trees

- Recall
  - The *height of a node* is the number of edges on the longest path from that node to a leaf node.
  - The *height of a tree* (or a subtree) is the height of the root of the tree (or a subtree).
  - The height of a leaf node is zero.

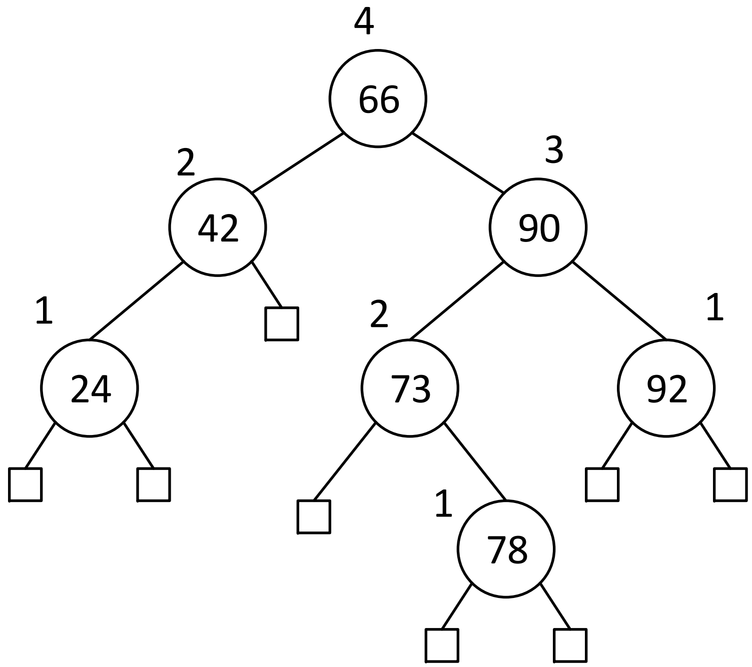
- An AVL tree is a binary search tree that satisfies the following *height-balance property*:

  For every internal node p of T, the heights of the children of p differ by at most one.

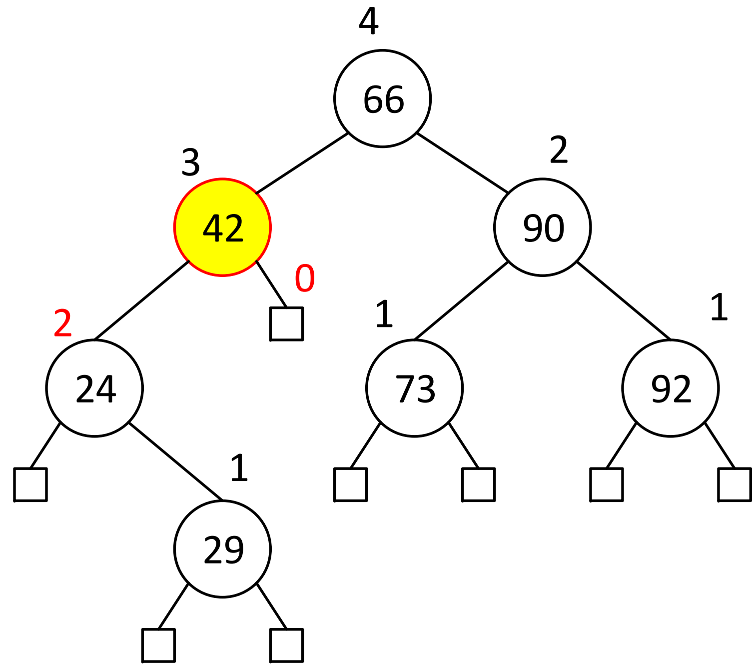
AVL Trees

- AVL tree example:

AVL tree



Not an AVL tree



AVL Trees

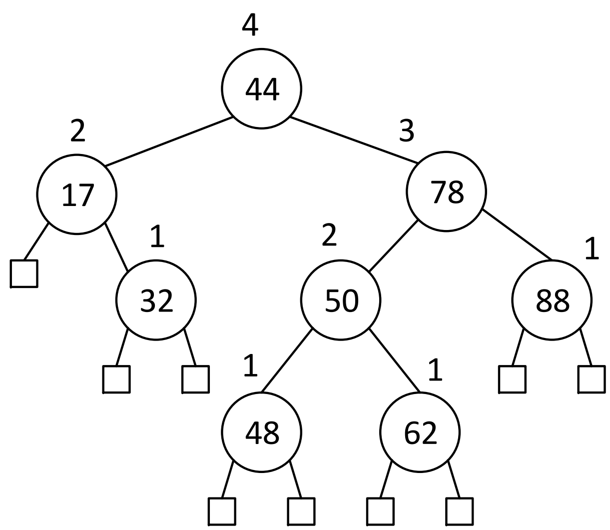
- Updating an AVL tree
  - A node *p* in a binary search tree is said to be *balanced* if the heights of *p*'s children differ by at most one.
  - Otherwise, a node is said to be *unbalanced.*
  - Therefore, every node in an AVL tree is balanced.

  - When we insert a node to an AVL tree or remove a node from an AVL tree, the resulting tree may violate the height-balance property.
  - So, we need to perform *post-processing*.

  - We will discuss only insertion.

AVL Trees

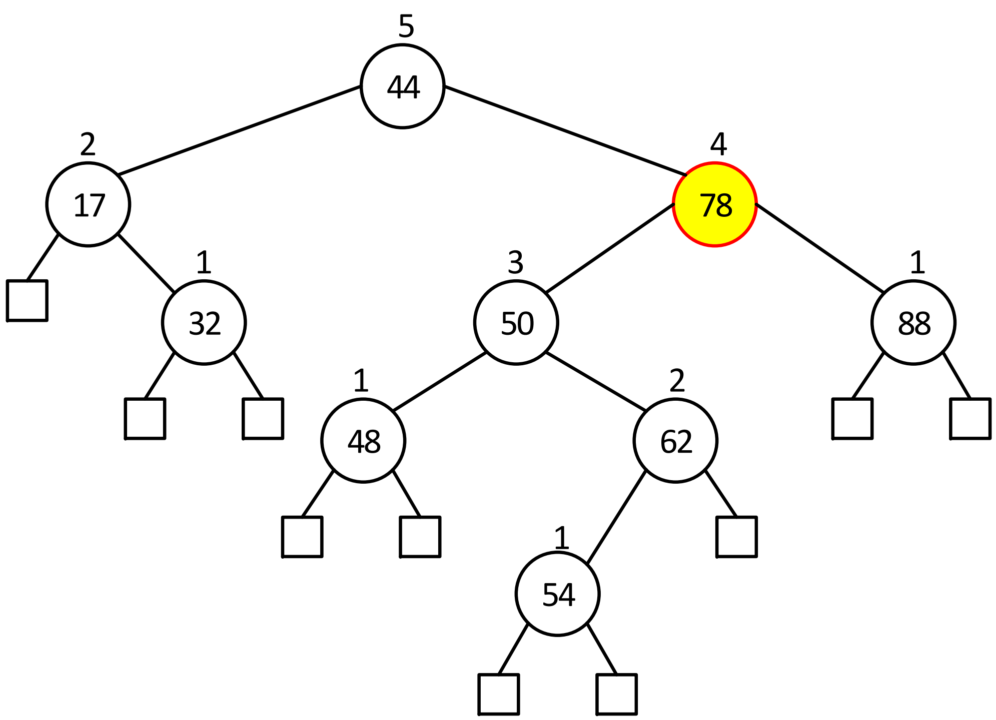
- When a node is inserted, the leaf node *p* where the new node is inserted becomes an internal node (with the entry of the new node).
- So, ancestors of *p* may be unbalanced.
- Restructuring is necessary.
- Consider the following tree:



AVL Trees

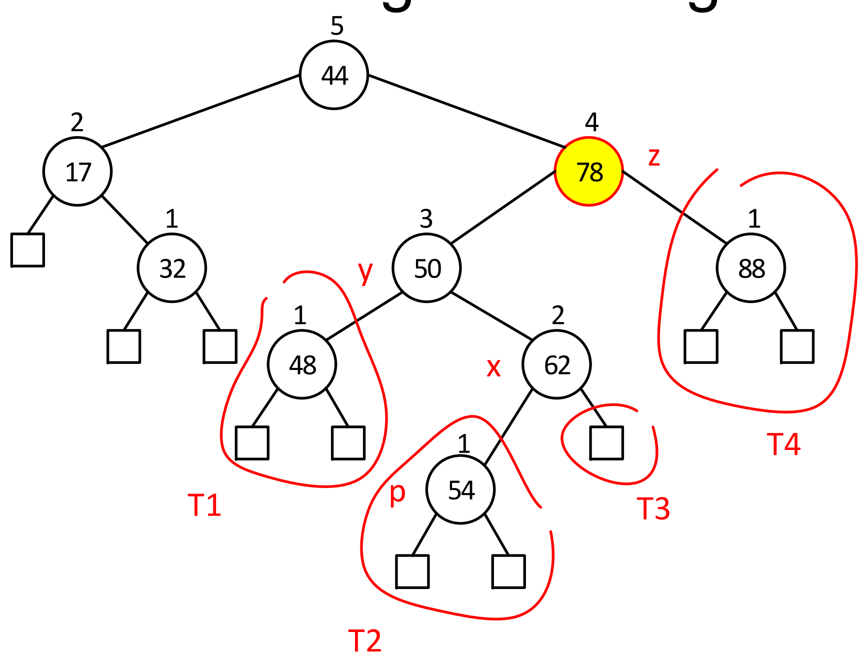
- After inserting 54, the node with 78 is unbalanced

After insertion, before rebalancing



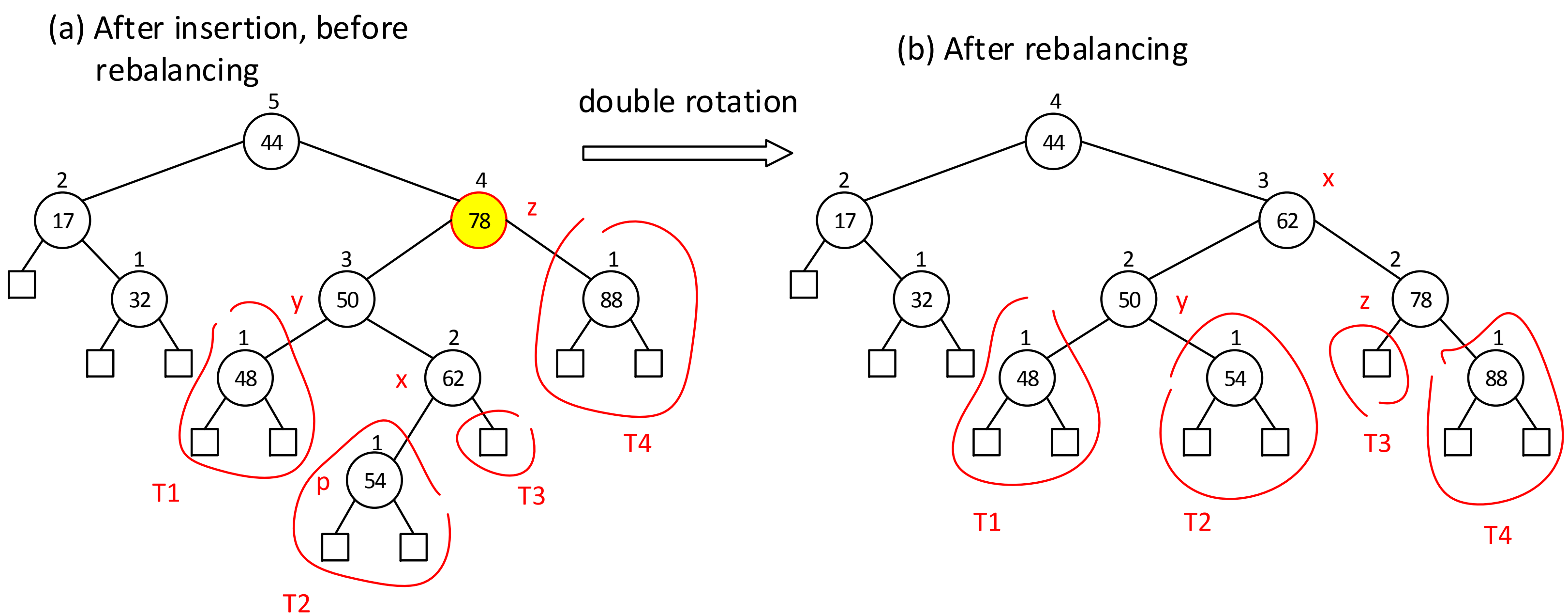
AVL Trees

- Post-processing
  - Search-and-repair strategy
  - Search a node *z* that is the lowest ancestor of *p* that is unbalanced.
  - *y* is *z*'s child with the greater height
  - *x* is *y*'s child with the greater height



AVL Trees

- Perform double rotation to rebalanced the tree



Sorting
Merge-Sort

- A divide-and-conquer algorithm

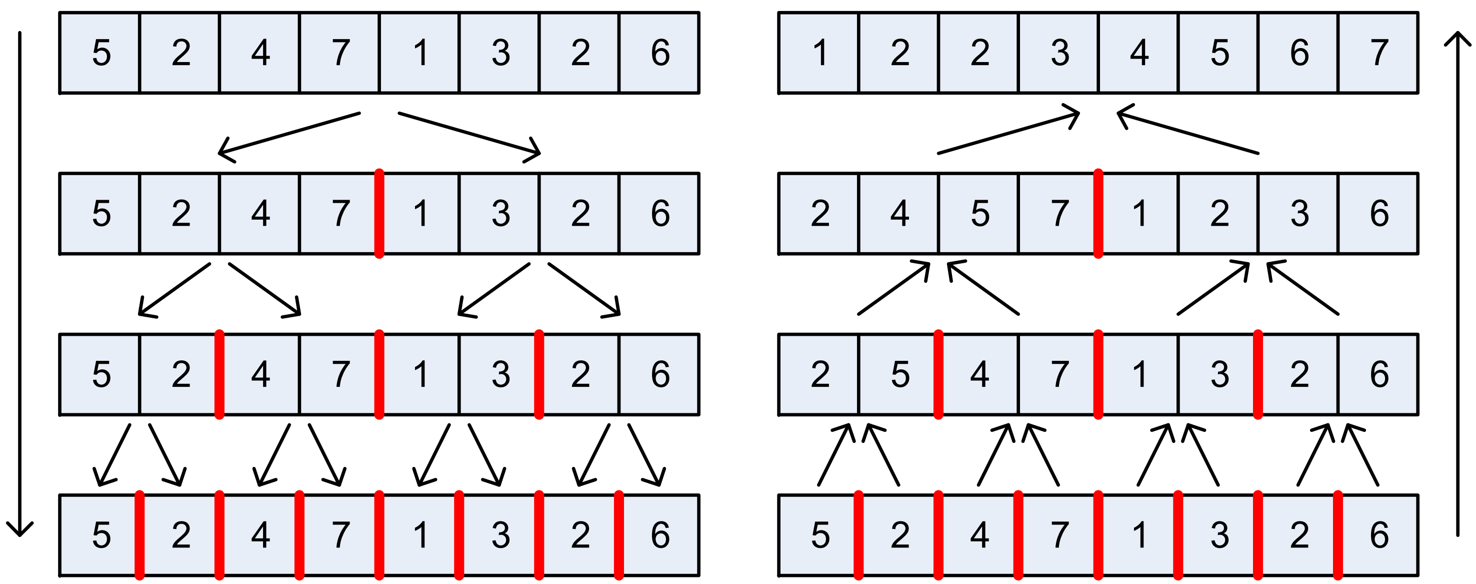
- Divide:
  - If input size is smaller than a certain threshold, solve it using a straightforward method.
  - Otherwise, divide the input into two or more subproblems.
- Conquer: Solve the subproblems recursively.
- Combine: Merge solutions to subproblems to generate a solution to the original problem.

Sorting
Merge-Sort

- Outline of the algorithm:
  1. Divide: If *S* has zero or one element, return *S* (because it is already sorted). Otherwise, divide *S* into two separate arrays, S_1 and S_2, of approximately equal size. S_1 contains the first $\lfloor n/2 \rfloor$ elements of *S* and S_2 contains the remaining $\lceil n/2 \rceil$ elements.
  2. Conquer: Sort S_1 and S_2 recursively.
  3. Combine: Put the elements back to *S* by merging the sorted sequences S_1 and S_2 into a sorted sequence.

Sorting
Merge-Sort

- Illustration



Sorting
Merge-Sort

- Array-based implementation

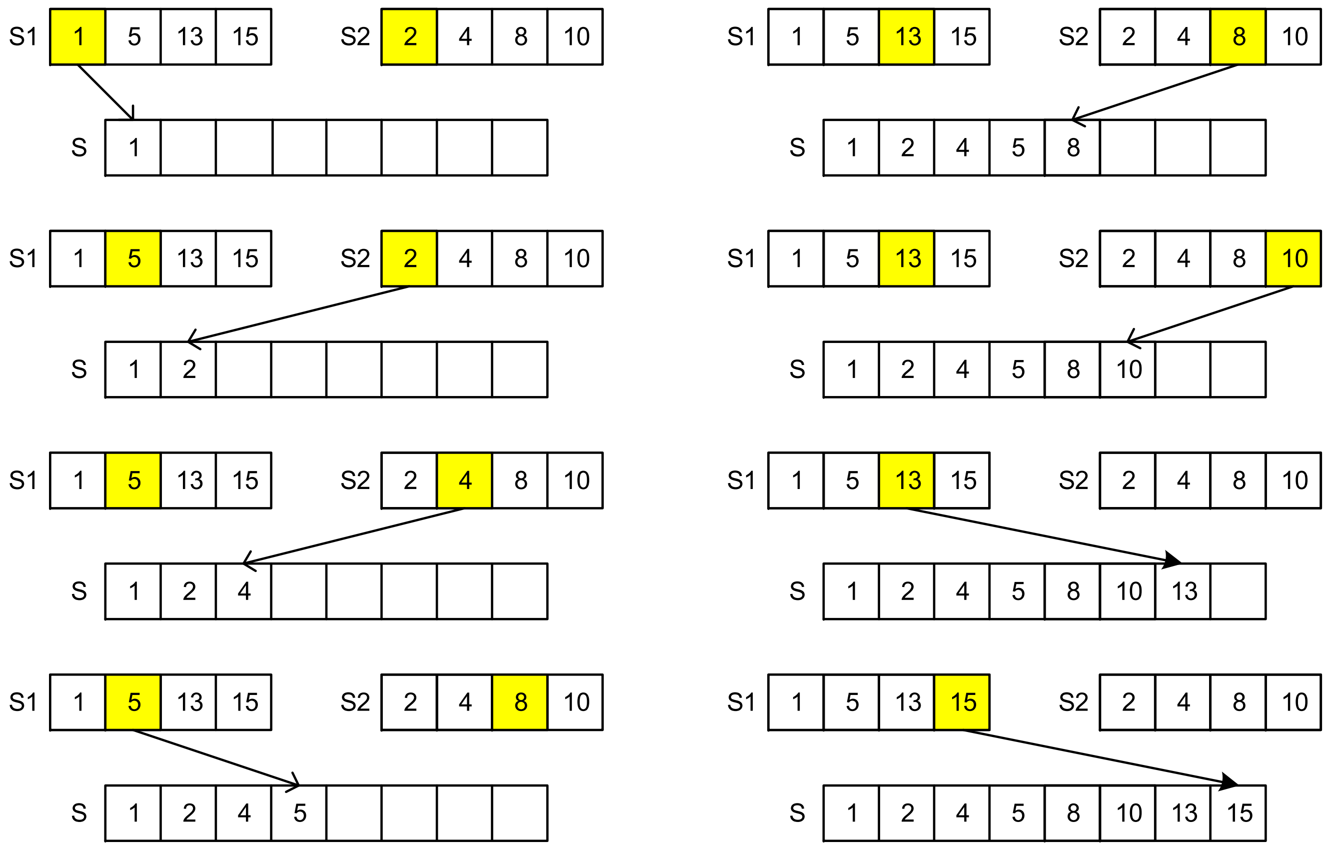
```
public static <K> void merge(K[] S1, K[] S2, K[] S, Comparator<K> comp) {
  int i = 0, j = 0;
  while (i + j < S.length) {
    if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
      S[i+j] = S1[i++];        // copy ith element of S1 and increment i
    else
      S[i+j] = S2[j++];        // copy jth element of S2 and increment j
  }
}
```

- Running time: *O*(n)

Sorting

Merge-Sort

- Merge



Sorting
Merge-Sort

- Java implementation

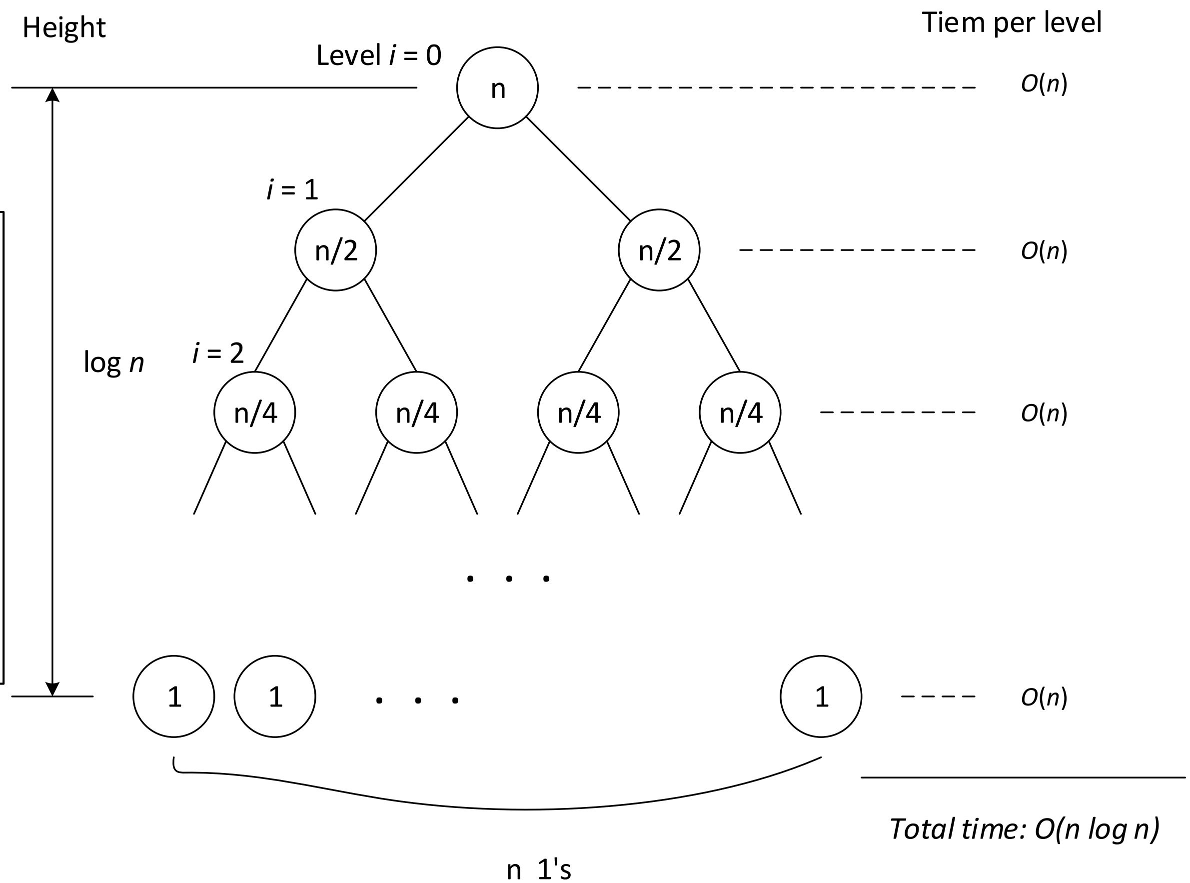
```
public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
   int n = S.length;
   if (n < 2) return;    // array is trivially sorted
   int mid = n/2;
   K[ ] S1 = Arrays.copyOfRange(S, 0, mid);  // copy of first half
   K[ ] S2 = Arrays.copyOfRange(S, mid, n);  // copy of second half
   mergeSort(S1, comp);                      // sort copy of first half
   mergeSort(S2, comp);                      // sort copy of second half
   merge(S1, S2, S, comp);  // merge sorted halves back into original
}
```

Sorting
Merge-Sort

- Running time analysis
  - Recursive calls are made in lines 7 and 8.
  - Excluding the recursive calls, the program takes O(n).
  - Each recursive call is made on a subarray with $n/2$ elements.
  - The running time of the *mergeSort* on an subarray with n/2 elements is $O(n/2)$.
  - As the successive recursive calls are made, the size of subarray becomes $n/2$, $n/4$, $n/8$, …, and so on, and eventually it becomes 1.
  - This can be represented as a recursion tree.

Sorting
Merge-Sort

- Running time analysis



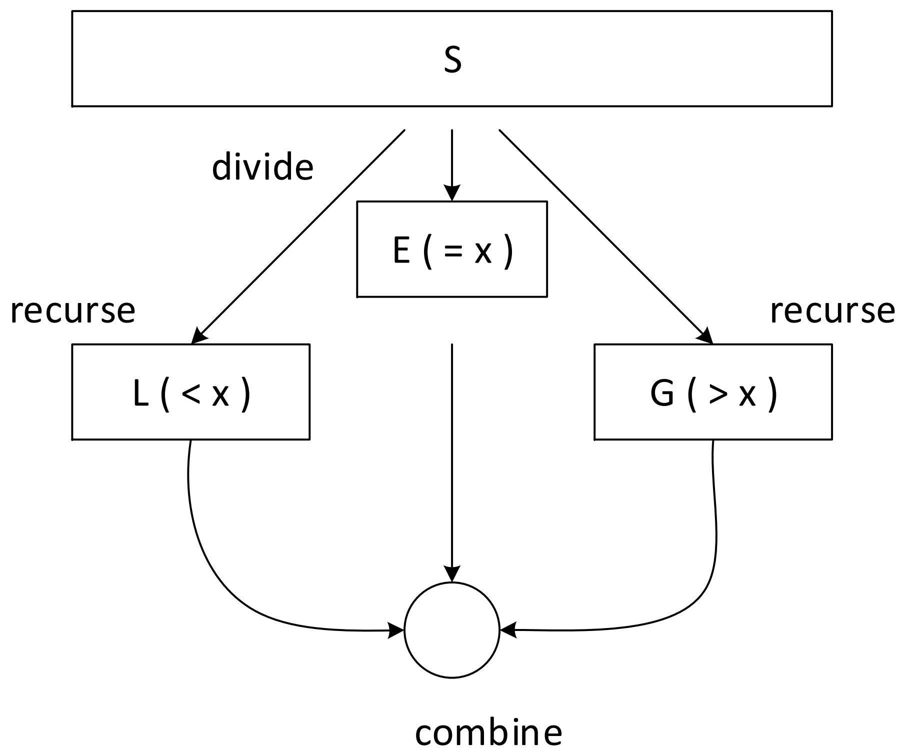
- Each level takes O(n)
- There are (log n + 1) levels
- Total running time = $O(n)(\log n + 1) = O(n)(\log n) + O(n) = O(n \log n)$

Sorting
Quick-Sort

- Outline
  - Divide: If *S* has only one element, return. Otherwise, remove all elements from *S* and put them into three sequences:
    - *L*: This sequence contains the elements that are less than *x*.
    - *E*: This sequence contains the elements that are equal to *x*.
    - *G*: This sequence contains the elements that are greater than *x*.
  - If the elements in *S* are distinct, then *E* has only one element, which is *x*.
  - Conquer: Recursively sort *L* and *G*.
  - Combine: Put back the elements from the three parts into *S* in order.
- The element *x* is called *pivot*.

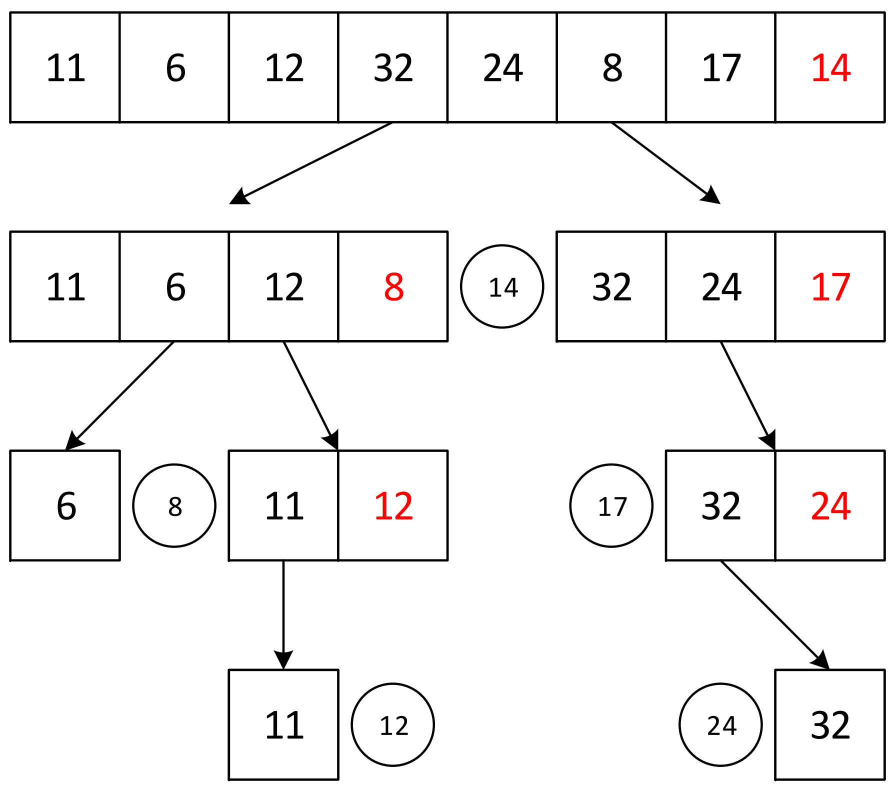
Sorting
Quick-Sort

- Outline



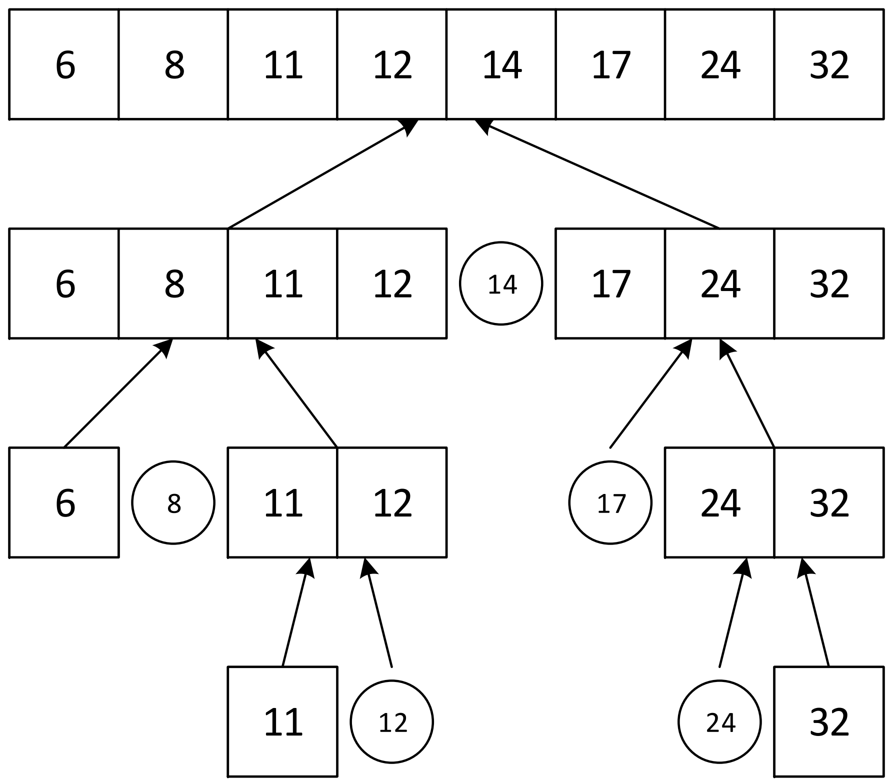
Sorting
Quick-Sort

- Illustration



Sorting
Quick-Sort

- Illustration (continued)



Sorting
Quick-Sort

- The “divide” step is usually called *partition*.
- Partitioning array *S* with *n* elements.
  - S[n – 1] is used as the pivot
  - Keeps two pointers, *left* and *right*
  - *Left* begins at S[0] and moves right until it meets the first element that is equal to or larger than the pivot, *right marker*.
  - *Right* begins at S[n – 2] and moves right until it meets the first element that is equal to or smaller than the pivot, *left marker*.
  - Left marker and right marker are swapped.
  - Repeat this until left and right cross each other
  - Left marker is swapped with pivot.

Sorting
Quick-Sort

- Partitioning illustration

85	24	63	45	17	31	96	50
l						*r*	

85	24	63	45	17	31	96	50
l					*r*		

swap

31	24	63	45	17	85	96	50
		l		*r*			

swap

31	24	17	45	63	85	96	50
			r <	*l*			

l and r crossed*; stop; swap S*[*l*] with pivot

31	24	17	45	50	85	96	63

Sorting
Quick-Sort

- Running time analysis
  - Can use the same method we used for merge-sort (i.e., use a recursion tree).
  - In merge-sort, we always have a balanced divide.
  - In quick-sort, depending on the pivot value, there may be a very unbalanced partitioning
  - In the best case:
    - Always balanced partitioning is created.
    - Running time is $O(n \log n)$
    - Even when partitions are not completely balanced (for example 1 : 9), the running time is still $O(n \log n)$

Sorting
Quick-Sort

- Running time analysis (continued)
  - In the worst case:
    - We always have an extremely unbalanced partitioning, i.e., no element on one side and $n - 1$ elements on the other side.
    - This occurs if an array is already sorted and the last element is chosen as a pivot.
    - Running time is $O(n^2)$.

Sorting
Quick-Sort

- Improvement
  - Randomized quick-sort: pivot is chosen randomly
  - *median-of-three* method: the median of the first element, the middle element, and the last element is used as a pivot.
  - When the input size becomes smaller than a certain threshold, we stop the recursion and sort that subarray using insertion-sort. There is no known one threshold value that is considered best. Our textbook suggests 50 and some experiments showed that a value around 15 is a reasonably good choice.

Sorting

Lower Bound for sorting

- The running time of any comparison-based sorting algorithm is $\Omega(n \lg n)$ in the worst case.

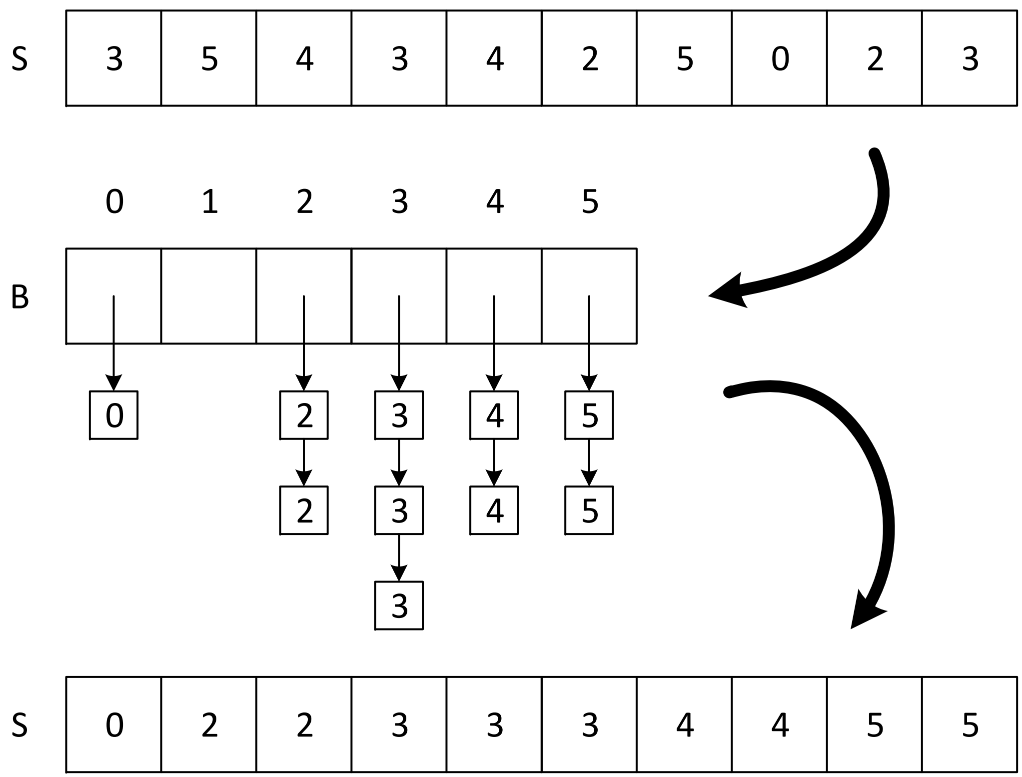
- Linear-time sorting: counting-sort, bucket-sort, radix-sort.
- Will discuss bucket-sort and radix-sort.

Sorting
Bucket-Sort

- Sorts a sequence of elements in a linear time with a constraint.
- Constraint:
  - The elements are integers in the range [0, N – 1], for some integer $N \geq 2$.
  - If the elements to be sorted are objects, then the objects must have integer keys with total ordering.

Sorting
Bucket-Sort

- Illustration (N = 6)



Sorting
Bucket-Sort

- Pseudocode

Algoritm bucketSort(*S*)
Input: Sequence *S* of entries with integer keys in range [0, *N* – 1]
Output: Sequence *S* sorted in nondecreasing order of keys

create an empty array *B* of size *N*
for each entry *e* in *S* do
  let *k* be the key of *e*
  remove *e* from *S* and add it to the end of bucket *B*[*k*], which is a
    sequence
for *i* = 0 to *N – 1* do
  for each entry in sequence *B*[*i*] do
    remove *e* from *B*[*i*] and insert it at the end of *S*

Sorting

Stable Sorting

- Let S = $((k_0, v_0), (k_1, v_1), \ldots, (k_{n-1}, v_{n-1}))$.
- Assume there are two entries (k_i, v_i) and (k_j, v_j) with an identical key, i.e, $k_i = k_j$, $i \neq j$
- We say a sorting algorithm is *stable* if (k_i, v_i) precedes (k_j, v_j) in S before sorting, then (k_i, v_i) also precedes (k_j, v_j) in S after sorting.

- Example:
  - S = ((9, W), (4, F), (7, H), (4, A), (2, P)) before sorting
  - S = ((2, P), (4, F), (4, A), (7, H), (9, W)) after sorting

- The bucket-sort described earlier is stable if S and B behave as queues.

Sorting
Radix-Sort

- Illustration:
  - Sorting three digit numbers
  - Each column is sorted using a stable sorting algorithm

456	932	912	148
723	912	723	239
148	723	932	456
239	745	239	648
932	456	745	723
912	148	148	745
648	648	648	912
745	239	456	932

Sorting
Comparison

- Running times

Running Time (average)	Sorting Algorithms
O(n)	bucket-sort, radix-sort
O(n log n)	heap-sort, quick-sort, merge-sort
O(n^2)	insertion-sort

Sorting
Comparison

- Insertion-Sort
  - When the number of elements is small (typically less than 50), insertion-sort is very efficient.
  - Insertion-sort is very efficient for an “almost” sorted sequence.
  - In general, due to its quadratic running time, insertion-sort is not a good choice except for the situations listed above.

Sorting
Comparison

- Heap-Sort
  - Heap-sort runs in $O(n \log n)$ in the worst case.
  - It works well on small- and medium-sized sequences.
  - It can be made an in-place sorting algorithm.
  - Its performance is poorer than that of quick-sort and merge-sort on large sequences.
  - Heap-sort is not a stable sorting algorithm.

Sorting
Comparison

- Quick-Sort
  - Worst-case running time is $O(n^2)$.
  - Experimental studies showed quick-sort outperformed heap-sort and merge-sort.
  - Quick-sort has been a default algorithm as a general-purpose, in-memory sorting algorithm.
  - It was used in *C* libraries.
  - Java uses it as the standard sorting algorithm for sorting arrays of primitive types.

Sorting
Comparison

- Merge-Sort
  - Worst-case running time is $O(n \log n)$.
  - It is difficult to make merge-sort an in-place sorting algorithm. So, it is less attractive than heap-sort or quick-sort.
  - Merge-sort is an excellent algorithm for sorting data that resides on the disk (or storage outside the main memory).

Sorting
Comparison

- Tim-Sort
  - Tim-sort is a hybrid algorithm which uses a bottom-up merge-sort and insertion-sort.
  - Tim-sort has been the standard sorting algorithm in Python since 2003.
  - Java uses Tim-sort for sorting arrays of objects.

Sorting
Comparison

- Bucket-Sort and Radix-Sort
  - Excellent for sorting entries with small integer keys, character strings, or *d*-tuple keys from a small range.

Selection Problem

- Selection problem: Given a set *S* of *n* comparable elements and an integer *k*, $1 \le k \le n$, find the element $e \in S$ that is larger than exactly $k - 1$ elements of *S*.
- The k^{th} smallest element is also referred to as the k^{th} *order statistic*.
- We assume *S* is a sequence.

- Will discuss *randomized quick-select*, which runs in $O(n)$ expected time.
- Similar to the randomized quick-sort algorithm.

Selection Problem

- Pseudocode

Algorithm quickSelect (S, k) // find the k^{th} order statistic

if n == 1 // n is the size of S

  return the (first) element

pick a random pivot element x of S and divide S into three subsequences:

L, storing the elements in S less than x

E, storing the elements in S equal to x

G, storing the elements in S greater than x

if $k \le |L|$ then // case 1

  return quickSelect(L, k)

else if $k \le |L| + |E|$ // case 2

  return x

else // case 3

  return quickSelect(G, $k - |L| - |E|$)

Selection Problem

- Illustration (Case 1: if $k \le |L|$)

  Find 5th order statistic.

  pivot = 9

- After partition:

$k = 5 \le |L|$, recurse on L with $k = 5$

7	3	5	1	6	2	9	9	13	15	17	10

L

$|L| = 6$

E

$|E| = 2$

G

$|G| = 4$

Selection Problem

- Illustration (Case 2: else if $k \le |L| + |E|$)

  Find 7th order statistic.

  pivot = 9

- After partition:

k = 7 ≤ |L| + |E|, return 9

7	3	5	1	6	2	9	9	13	15	17	10

L

|L| = 6

E

|E| = 2

G

|G| = 4

Selection Problem

- Illustration (Case 3: else if $k > |L| + |E|$)

  Find 10th order statistic.

  pivot = 9

- After partition:

k = 10 > |L| + |E|, recurse on G with k = 2

7	3	5	1	6	2	9	9	13	15	17	10

L

|L| = 6

E

|E| = 2

G

|G| = 4

Greedy Algorithms

- Consider the algorithms of the class project.
- The problem is: Given a start node *S*, find the a shortest path from *S* to a destination node *D*.
- We can solve this problem by
  - Find all possible paths from *S* to *D*.
  - Select a path with the shortest length.
- This approach guarantees that we find a solution, but it could be expensive.
- A greedy approach: Beginning at *S*, select the next node which is best at that moment, such as based on *direct distances*.
- Another simple example: *coin changing* problem

Greedy Algorithms

- When we solve an optimization problem, we need to make a series of choices.
- When making a choice, the greedy method considers all options that are “available at that moment” and chooses the best option among them.
- In other words, it chooses a “locally optimal” option.
- The greedy method does not always lead to a global optimal solution.
- However, for many practical problems, the greedy method gives us a global optimal solution.

- Will describe the *Huffman code* algorithm, which is a greedy algorithm.

Huffman Code - Introduction

- A data is considered as a sequence of characters.
- Each character is encoded to a unique binary string, called a *codeword*.
- Example:
  - ‘A’ is encoded to a codeword 0000
  - ‘B’ is encoded to a codeword 0001
  - and so on
- Decoding: Converting a codeword to the initial character.

Huffman Code - Introduction

- There are different ways of encoding characters to binary strings.
- A fixed-length code uses the same number of bits for different characters.
- Example of a fixed-length code: ASCII code.
- A variable-length code uses different number of bits for different characters.

Huffman Code - Introduction

- Fixed-length code vs. variable-length code
  - Fixed-length code: Uses the same number of bits for all characters.
  - Variable-length code: Uses different number of bits for different characters.
- Prefix code: No codeword is a prefix of some other codeword.
- For example, if the codeword for ‘X’ is 10100 and the codeword for ‘Y’ is ‘101”, then this code is NOT a prefix code (because 101 is a prefix of 10100).
- Prefix codes simplify the decoding process.

Huffman Code - Introduction

- A goal of data compression: Minimize the size of the compressed data (where each character is represented by a codeword).
- The Huffman code is a *variable-length*, *prefix* code used for data compression.
- It uses a smaller number of bits for a character that appears in the document with a high frequency and uses a larger number of bits for a character that appears rarely.

Huffman Code - Introduction

- The following table shows the frequency of occurrences of each character in a given data and two coding schemes.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Huffman Code - Introduction

- The fixed-length code requires 300,000 bits (3 bits X 100,000 characters).

- The variable-length code requires less number of bits:

  $45000 \cdot 1 + 13000 \cdot 3 + 12000 \cdot 3 + 16000 \cdot 3 + 9000 \cdot 4 + 5000 \cdot 4 = 224{,}000$ bits

  This code happens to be an optimal code for the given data.

Huffman Code - Introduction

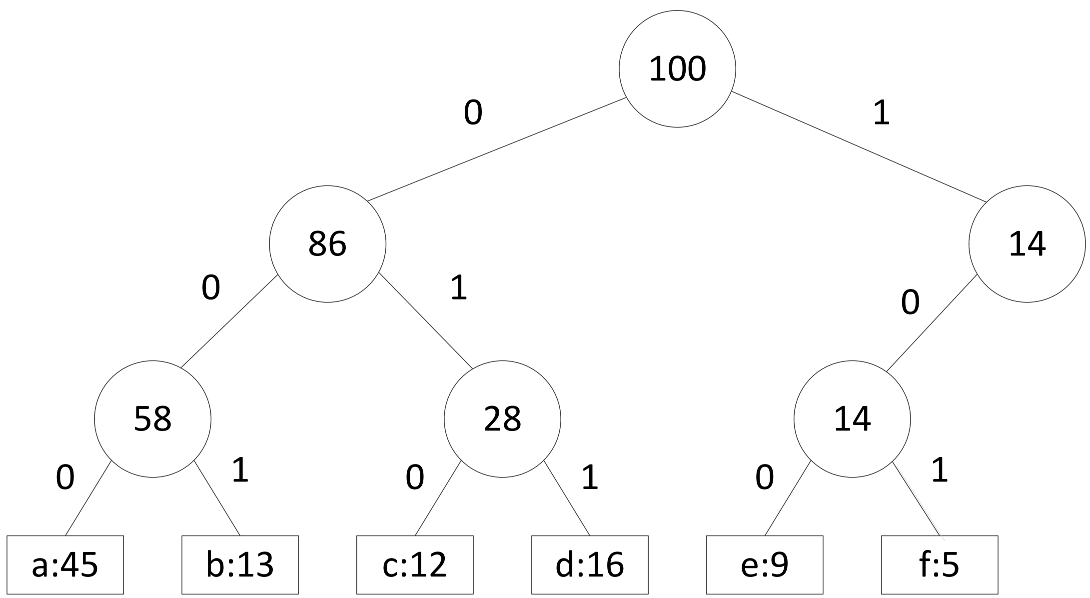
- Huffman code algorithm is a greedy algorithm that constructs an *optimal prefix code* called *Huffman code*.

- Encoding: Represent each character in the data with the corresponding codeword.

- Decoding: Convert an encoded data to the original data. This can be done efficiently using a binary tree.

Huffman Code - Decoding

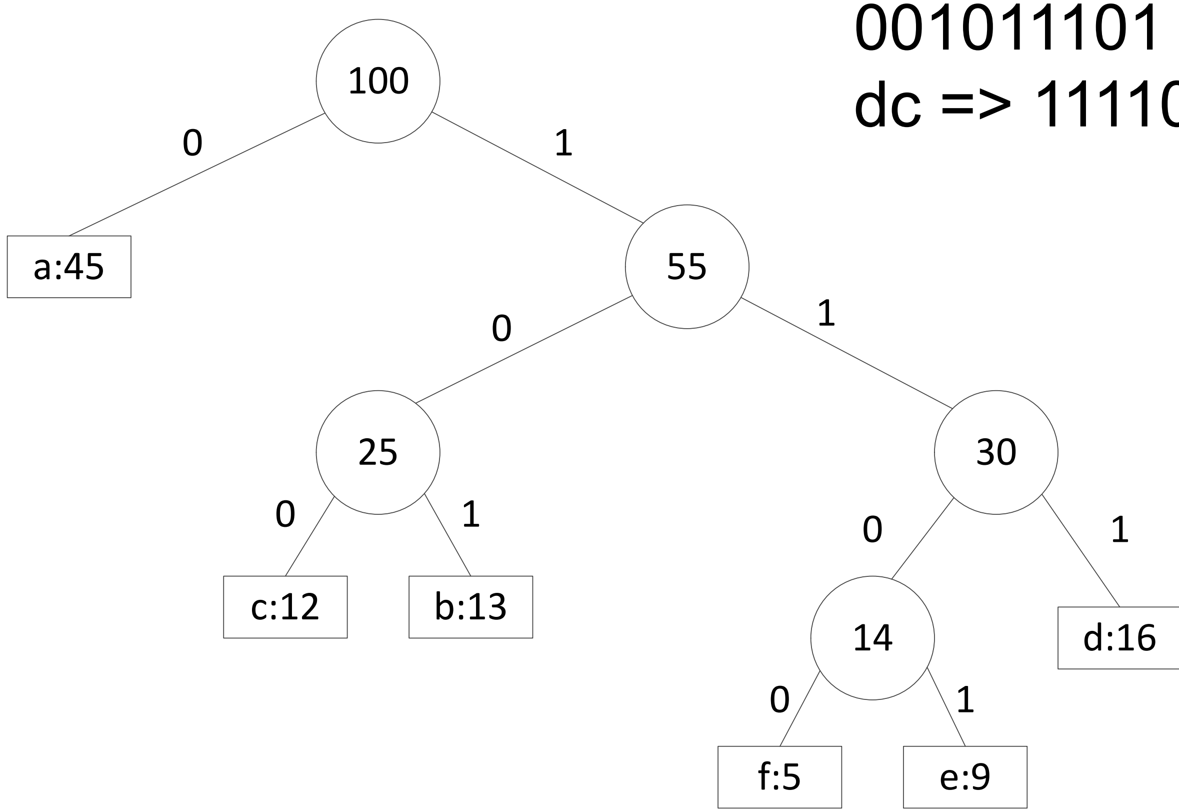
- Coding tree for the fixed-length code (of the above example)



Huffman Code - Decoding

- Coding tree for the variable-length code, Huffman code (of the above example)

001011101 => aabe
dc => 111100



Huffman Code - Decoding

- In a binary tree for an optimal code, each node has exactly two children.
- Decoding:
  - Begin at the root and scan the binary code.
  - If a bit is 0, go down to the left. If a bit is 1, go down to the right.
  - When you are at a leaf node, the decoding of one character is done and the character is shown in the leaf node.
  - Go back to the root and repeat the same with the remaining bit string.

Huffman Code - Decoding

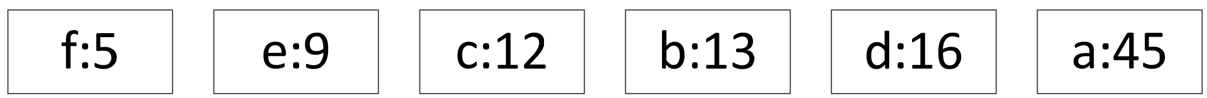
- Decoding of 001011101 (Huffman code):
  - Scanning the first bit, 0, takes you to a leaf node with the character *a*. So, it is decoded as *a*.
  - Next 0 is also decoded as *a*.
  - The next three bits 101 leads to *b*.
  - The next four bits 1101 decodes to *e*.
  - So, the decoded string is *aabe*.

Huffman Code - Encoding

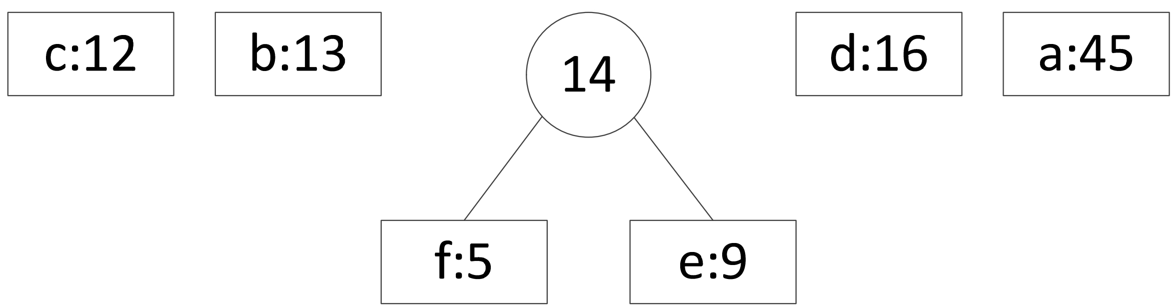
- To encode a character, follow the path from the root to the leaf corresponding to the character, and concatenate the bits along the path.
- Example: encoding *dc*
  - The path from the root to the leaf with *d*: 111
  - The path from the root to the leaf with *c*: 100
  - So, the *dc* is encoded to 111100

Constructing a Huffman Code

- Illustration
  - (a) Initial Q (which is a priority queue)

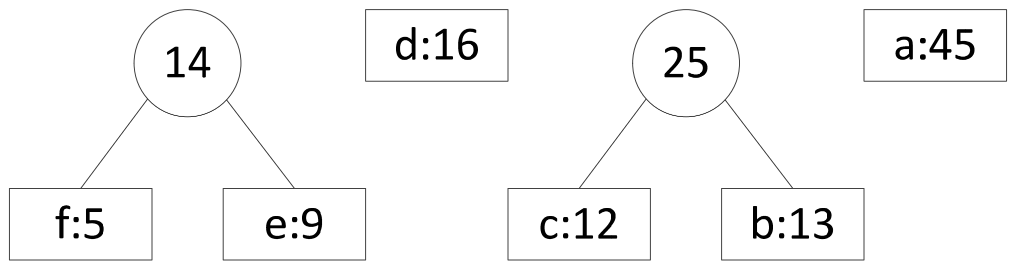


  - (b) (f:5) and (e:9) are extracted, merged, and inserted into Q.

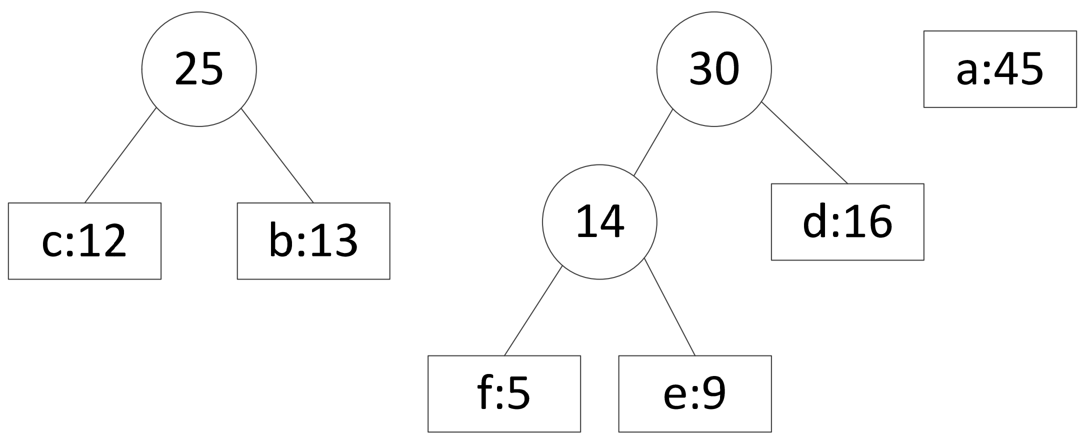


Constructing a Huffman Code

(c) (c:12) and (b:13) are extracted, merged, and inserted into Q.

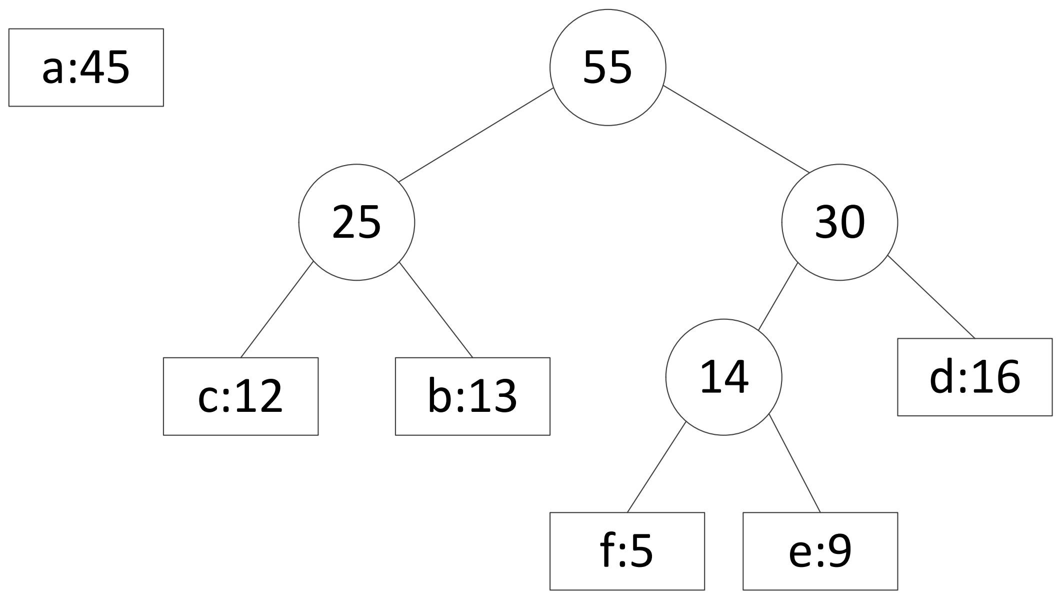


(d) ((f:15, e:9):14) and (d:16) are extracted, merged, and inserted into Q.



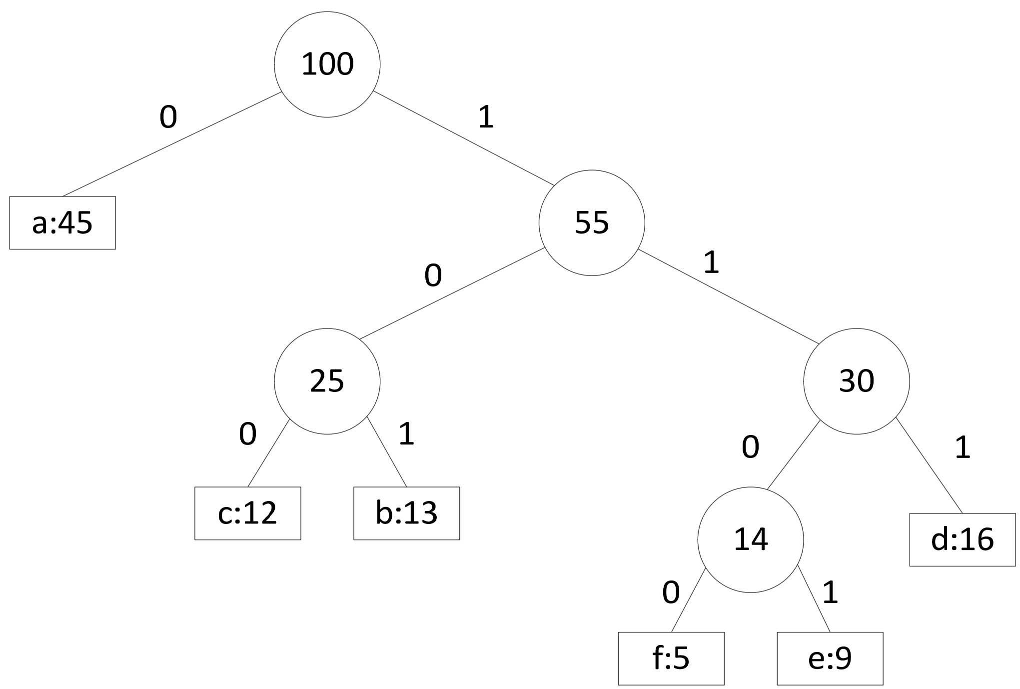
Constructing a Huffman Code

(e) ((c:12, b:13):25) and (((f;15, e:9):14, d:16):30) are extracted, merged, and inserted into Q.



Constructing a Huffman Code

(f) (a:45) and ((c:12, b:13):25, (((f:5, e:9):14, d:16):30):55) are extracted, merged, and inserted into Q.



Dynamic Programming

- Refers to a technique or an approach, not an algorithm.
- Solves problems by combining solutions to subproblems (like divide-and-conquer).
- If subproblems are not independent, some subproblems are solved multiple times.
- Dynamic programming approach:
  - Bottom-up approach: Problems are solved in the increasing order of size (i.e., smallest problem first, followed by the next smallest problem, and so on).
  - Each subproblem is solved once and the solution is stored in a table.
- Typically used for optimization problems.

Dynamic Programming

- Consider the following problem (from Aho, Hopcroft, and Ullman):
  - Two baseball teams X and Y are competing for the World Series championship.
  - A team wins the championship title if it wins four out of seven games.

  - $P(i, j)$ is defined as: the probability that one of the teams, say X, will eventually win the championship title, given that X still needs to win i more games to win the title and Y still needs to win j more games to win the title.

Dynamic Programming

- Consider the following problem (continued):
  - Example: X won 1 game and Y won 2 games. Then, X needs 3 more games and Y needs 2 more games, and the probability that X will win the championship title is denoted $P(3, 2)$.
  - We assume that two teams are equally likely to win any particular game.
  - Two extreme cases

    $P(0, j) = 1$ for any $j > 0$ // X won the championship

    $P(i, 0) = 0$ for any $i > 0$ // Y won the championship

Dynamic Programming

- Consider the following problem (continued):
  - In general, we can calculate $P(i, j)$ recursively as follows:

$$
\begin{aligned}
P(i, j) &= 1, \text{ if } i = 0 \text{ and } j > 0 \\
&= 0, \text{ if } i > 0 \text{ and } j = 0 \\
&= (P(i-1, j) + P(i, j-1)) / 2, \text{ if } i > 0 \text{ and } j > 0
\end{aligned}
$$

  - This is a divide-and-conquer approach.
  - But, some subproblems are solved multiple times.

Dynamic Programming

- Consider the following problem (continued):
  - For example,

    $P(7, 7) = (P(6, 7) + P(7, 6)) / 2$

    $P(6, 7) = (P(5, 7) + P(6, 6)) / 2$

    $P(7, 6) = (P(6, 6) + P(7, 5)) / 2$

  - In this example, $P(6, 6)$ is calculated more than once.

Dynamic Programming

- Dynamic programming approach:
  - We solve smaller problems first (smaller problems refer to $P(i, j)$ with small i and j).
  - Store the results in a table.
  - When we solve a larger problem, we use the solutions to smaller problems, which are stored in the table.

Dynamic Programming

- Illustration
  - First, we solve $P(0, j)$ for all j (i.e., j = 1, 2, 3, 4, 5, 6) and solve $P(i, 0)$ for all i (i.e., i = 1, 2, 3, 4, 5, 6) and store them in a table:

$P(i, j)$

						1	6
						1	5
						1	4
						1	3
						1	2
						1	1
0	0	0	0	0	0		0
6	5	4	3	2	1	0	

j

i

Dynamic Programming

- Illustration (continued)
  - Next,
    - $P(1, 1) = (P(0, 1) + P(1, 0)) / 2 = (1 + 0) / 2 = 1/2$;
    - $P(1, 2) = (P(0, 2) + P(1, 1)) / 2 = (1 + 1/2) / 2 = 3/4$;
    - $P(2, 1) = (P(1, 1) + P(2, 0)) / 2 = (1/2 + 0) / 2 = 1/4$;

$P(i, j)$

						1	6
						1	5
						1	4
						1	3
					3/4	1	2
				1/4	1/2	1	1
0	0	0	0	0	0		0
6	5	4	3	2	1	0	

j

i

Dynamic Programming

- Illustration (continued)
  - Next,
    - $P(1, 3) = (P(0, 3) + P(1, 2)) / 2 = (1 + 3/4) / 2 = 7/8;$
    - $P(2, 2) = (P(1, 2) + P(2, 1)) / 2 = (3/4 + 1/4) / 2 = 1/2;$
    - $P(3, 1) = (P(2, 1) + P(3, 0)) / 2 = (1/4 + 0) / 2 = 1/8;$

$P(i, j)$

						1	6
						1	5
						1	4
					7/8	1	3
				1/2	3/4	1	2
			1/8	1/4	1/2	1	1
0	0	0	0	0	0		0
6	5	4	3	2	1	0	

j

i

Dynamic Programming - LCS

- A subsequence of a given sequence is the given sequence with zero or more elements left out.
- The following are subsequences of

  S = <GGATAATTGAGA>

  - s1 = <GGTGA>
  - s2 = <GATAGA>
  - s3 = <GGATGAGA>
  - s4 = <TAATGA>

  . . .

Dynamic Programming - LCS

- Given two sequences

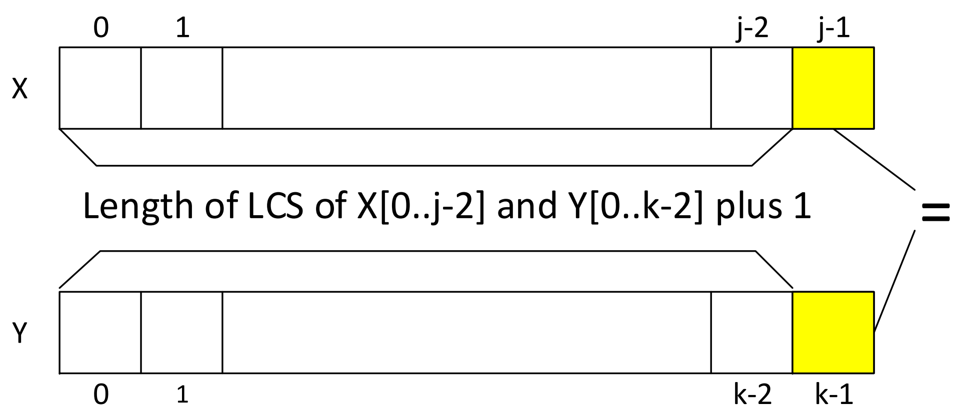
  $X = x_0x_1 \ldots x_{n\text{-}1}$, $Y = y_0y_1 \ldots y_{m\text{-}1}$

  The longest common subsequence (LCS) of X and Y is a longest sequence that is a subsequence of both X and Y.

- Brute-force method:
  - Among 2^n subsequences of X, identify those that are also subsequences of Y. And, select a longest subsequence.
  - This takes $O(2^n m)$

Dynamic Programming - LCS

- For simplicity, we will use an array notation to represent a sequence and its elements.
- Given two sequences X[0..j-1] and Y[0..k-1], $L_{j,k}$ denotes the length of the longest common subsequence of X and Y.
- When j = 0 or k = 0, $L_{j,k}$ = 0.
- When $j \geq 1$ and $k \geq 1$, there are two cases
  - Case 1. $X[j-1] = Y[k-1]$

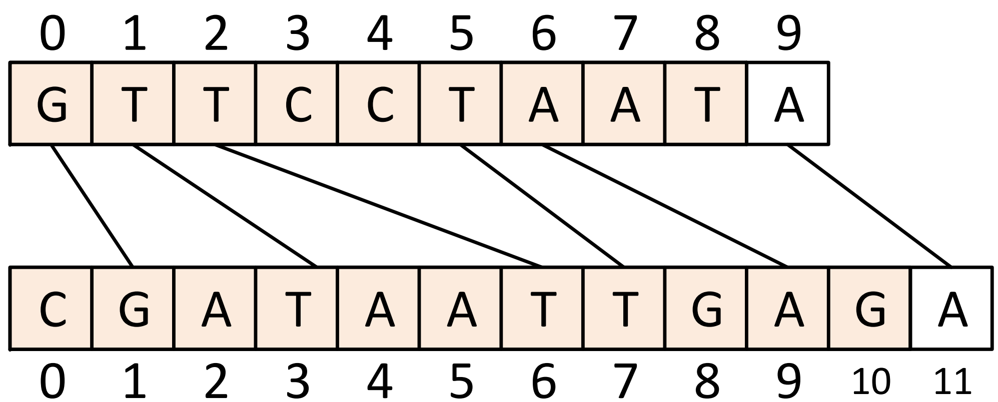


Dynamic Programming - LCS

– Case 1 (continued)

In this case, $L_{j,k}$ is one more than $L_{j-1,k-1}$, that is, the length of a longest common subsequence of X[0..j-2] and Y[0..k-2]:

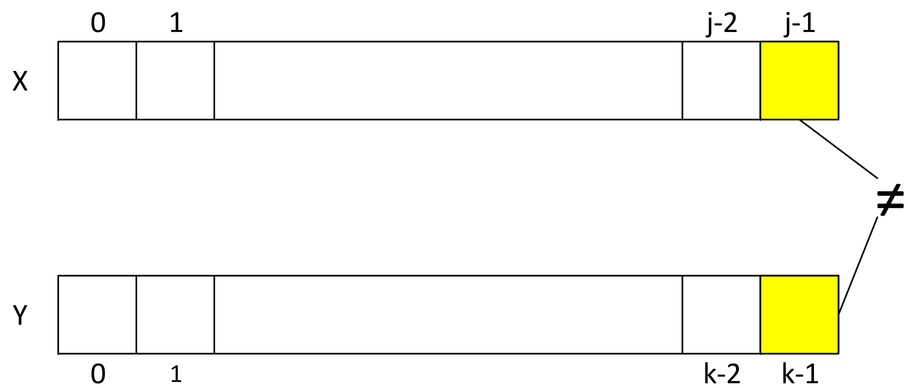
$$L_{j,k} = 1 + L_{j-1,k-1}$$



$$L_{10,12} = 1 + L_{9,11}$$

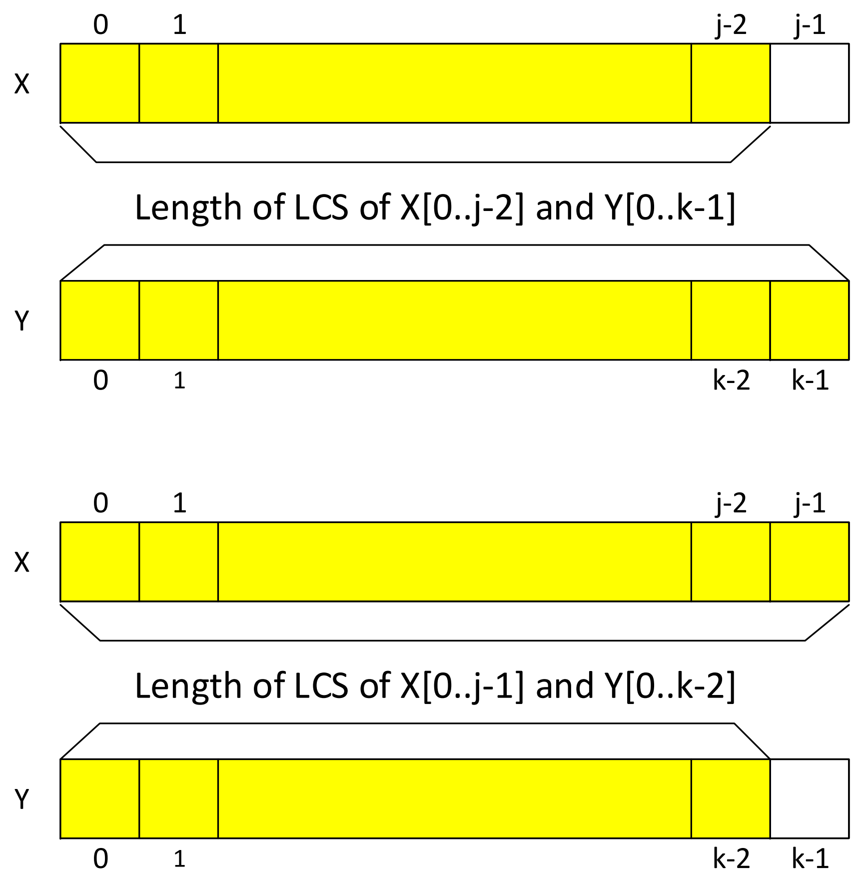
Dynamic Programming - LCS

– Case 2. $X[j\text{-}1] \neq Y[k\text{-}1]$



Dynamic Programming - LCS

– Case 2 (continued): The larger of the following two.



Dynamic Programming - LCS

– Case 2 (continued)

	0	1	2	3	4	5	6	7	8	9
X	G	***C***	***A***	***G***	***T***	***T***	A	***G***	T	A

LCS = CAGTTG, $L_{9,12} = 6$

	0	1	2	3	4	5	6	7	8	9	10	11
Y	***C***	***A***	C	T	T	***G***	***T***	A	C	***T***	***G***	C

	0	1	2	3	4	5	6	7	8	9
X	G	***C***	***A***	G	***T***	***T***	A	***G***	***T***	***A***

LCS = CATTGTA, $L_{10,11} = 7$

	0	1	2	3	4	5	6	7	8	9	10	11
Y	***C***	***A***	C	***T***	***T***	***G***	***T***	***A***	C	T	G	C

$L_{10,12} = \max\{L_{9,12}, L_{10,11}\} = 7$

Dynamic Programming - LCS

– Java code (computes *L*[*j*][*k*])

```
public static int [ ][ ] LCS(char[ ] X, char[ ] Y) {
  int n = X.length;
  int m = Y.length;
  int[ ][ ] L = new int[n+1][m+1];
  for (int j=1; j < n+1; j++)
    for (int k=1; k < m+1; k++)
      if (X[j-1] == Y[k-1])            // align this match
        L[j][k] = L[j-1][k-1] + 1;
      else                             // choose to ignore one character
        L[j][k] = Math.max(L[j-1][k], L[j][k-1]);
  return L;
}
```

Dynamic Programming - LCS

– Java code (reconstructs LCS)

```
public static char[ ] reconstructLCS(char[ ] X, char[ ] Y, int[ ][ ] L) {
   StringBuilder solution = new StringBuilder();
   int j = X.length;
   int k = Y.length;
   while (L[j][k] > 0)            // common characters remain
     if (X[j-1] == Y[k-1]) {
       solution.append(X[j-1]);
       j--;   k--;
     } else if (L[j-1][k] >= L[j][k-1])  j--;
     else  k--;
   return solution.reverse().toString().toCharArray();
 }
```

Dynamic Programming - LCS

- L matrix for X = GCAGTTAGTA, Y = CACTTGTACTGC

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	2	2	2	2
0	1	2	2	2	2	2	2	2	2	2	2	2
0	1	2	2	2	2	3	3	3	3	3	3	3
0	1	2	2	3	3	3	4	4	4	4	4	4
0	1	2	2	3	4	4	4	4	4	5	5	5
0	1	2	2	3	4	4	4	5	5	5	5	5
0	1	2	2	3	4	5	5	5	5	5	6	6
0	1	2	2	3	4	5	6	6	6	6	6	6
0	1	2	2	3	4	5	6	7	7	7	7	7

References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.
- A.V. Aho, J.E. Hopcroft, and J.D. Ullman, “Data Structures and Algorithms,” Addison-Wesley, 1983, pp. 312 – 314.