.

CS 321 Data Structures

Class Notes

Instructor: Dr. Jyh-haw Yeh
Dept. of Computer Science
Boise State University

Chapter 2: Getting Started

• Algorithms

– An algorithm specifies a sequence of computational steps to solve a well-defined computational problem.

– A problem specifies the desired input/output relationship.

EX: Sorting problem

* Input: a sequence of n numbers $< a_1, a_2, \ldots, a_n >$.
* Output: A permutation (reordering) $< a_{i_1}, a_{i_2}, \ldots, a_{i_n} >$, where $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_n}$ and all $a_{i_j} \in \{a_1, a_2, \ldots, a_n\}$.

– Algorithm can be specified in English or pseudocode.

```
EX: Insertion_Sort(A) // Note that array A's index starts at 1
    1. for j <-- 2 top length[A}
    2.     do key <-- A[j]
    3.        // Insert A[j] into the sorted portion
    4.        i <-- j-1
    5.        while i > 0 and A[i] > key
    6.              do A[i+1] <-- A[i]
    7.                 i <-- i-1
    8.        A[i+1] <-- key
```

Try to use Insertion_Sort to sort $A =< 31, 41, 59, 26, 41, 58 >$.

• Analyzing Algorithms

– Analyzing algorithms is to estimate the running time in terms of the input size n. That is, express the running time as a function of n. Using the Insertion_Sort as an example.

Insertion_Sort(A)	cost	times
1.	c_1	n
2.	c_2	$n-1$
3.	0	
4.	c_4	$n-1$
5.	c_5	$\sum_{j=2}^{n} t_j$
6.	c_6	$\sum_{j=2}^{n}(t_j-1)$
7.	c_7	$\sum_{j=2}^{n}(t_j-1)$
8.	c_8	$n-1$

Let $T(n)$ be the running time of Insertion_Sort(A) with input size n.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

* Best case: If the input array A is a sorted array already, then $t_j = 1$ for all j.

  $$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
  $$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

  It is a **linear** function of n

* Worst case: If the input array A is sorted in a reverse order, then $t_j = j$ for all j.

  $$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\tfrac{n(n+1)}{2} - 1) + c_6(\tfrac{n(n-1)}{2}) + c_7(\tfrac{n(n-1)}{2}) + c_8(n-1)$$
  $$= (\tfrac{c_5+c_6+c_7}{2})n^2 + (c_1 + c_2 + c_4 + \tfrac{c_5-c_6-c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

  It is a **quadratic** function of n.

* In some particular cases, we shall be interested in the average-case or expected running time of an algorithm. (Usually the average-case is as bad as worst-case).

• Other Sorting Algorithms: Selection and Merge

–
```
Merge_Sort(A, p, r)
1. if p < r
2.    then q <-- (p+r)/2  // integer division
3.         Merge_Sort(A, p, q)
```

```
4.          Merge_Sort(A, q+1, r)
5.          Merge(A, p, q, r)

Merge(A, p, q, r)
1. n1 <-- q-p+1
2. n2 <-- r-q
3. create arrays L[1...n1+1] and R[1...n2+1]
4. for i <-- 1 to n1
5.      do L[i] <-- A[p+i-1]
6. for j <-- 1 to n2
7.      do R[j] <-- A[q+j]
8. L[n1+1] <-- infinity
9. R[n2+1] <-- infinity
10. i <-- 1
11. j <-- 1
12. for k <-- p to r
13.      do if L[i] <= R[j]
14.            then A[k] <-- L[i]
15.                 i++
16.            else A[k] <-- R[j]
17.                 j++
```

Try to run the Merge_Sort to an input $A =< 5, 2, 4, 6, 1, 3, 2, 6 >$.

Running time analysis: draw the figure like the one on page 35 (textbook).

* $\lceil \log_2 n \rceil$ levels of merging in the tree
* Each level takes linear (to n) time
* Thus, total running time is $O(n \log n)$ (or a better notation $\Theta(n \log n)$ after we discuss asymptotic notations)

– `Selection_Sort(A)`

```
1. for j = n downto 2
2.     do largest = 1;
3.        for i = 2 to j
4.            do if A[i] >= A[largest]
5.                  then largest = i;
6.        swap A[j] and A[largest]
               // put the largest element to the last position
```

Try to run the Selection_Sort to an input $A =< 5, 2, 4, 6, 1, 3, 2, 6 >$.

Running time analysis:

* nested 'for' loops and each 'for' loop runs linear iterations
* Thus, total running time is $O(n^2)$ (or a better notation $\Theta(n^2)$ after we discuss asymptotic notations)

Chapter 3: Growth of Functions

• Asymptotic Notation

– **Θ-notation: asymptotically tight bound**

∗ *Alternative Definition: for two given functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ $\iff \lim_{n\to\infty} \frac{g(n)}{f(n)} = c$*

∗ Using the definition to show

$f(n) = \frac{1}{2}n^2 - 4n + 100 = \Theta(n^2)$

since

$\lim_{n\to\infty} \frac{n^2}{\frac{1}{2}n^2-4n+100} = 2$

∗ Without using definition, we can determine the Θ-notation (as well as other notations) of a function $f(n)$ by throwing away lower-order terms and ignores the leading coefficient of the highest-order term.

For example,

$f(n) = \frac{1}{2}n^2 - 4n + 100 = \Theta(n^2)$

– ***O*-notation: asymptotically upper bound**

∗ *Alternative Definition: for two given functions $f(n)$ and $g(n)$, $f(n) = O(g(n))$ $\iff \lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ or ∞*

∗ Using the definition to show

$$\begin{aligned} f(n) = \tfrac{1}{2}n^2 - 4n + 100 &= O(n^2) \quad \text{since } \lim_{n\to\infty} \tfrac{n^2}{\frac{1}{2}n^2-4n+100} = 2 \\ &= O(n^3) \quad \text{since } \lim_{n\to\infty} \tfrac{n^3}{\frac{1}{2}n^2-4n+100} = \infty \\ &= O(\infty) \quad \text{since } \lim_{n\to\infty} \tfrac{\infty}{\frac{1}{2}n^2-4n+100} = \infty \end{aligned}$$

∗ Note that if $f(n) = \Theta(g(n))$, then $f(n) = O(g(n))$.

∗ We usually use big-O notation to describe the running time of an algorithm rather than using Θ-notation. However, Θ-notation tells people more exact running time of an algorithm. For example:

Insertion_Sort(A) $= O(n^2)$ for all input n

Insertion_Sort(A) $\neq \Theta(n^2)$

$$\neq \Theta(n)$$

The worst case of Insertion_Sort(A) $= \Theta(n^2)$

- **Ω-notation: asymptotically lower bound**
  - *Alternative Definition: for two given functions $f(n)$ and $g(n)$, $f(n) = \Omega(g(n))$ $\iff \lim_{n\to\infty} \frac{g(n)}{f(n)} = c$ or 0*
  - Using the definition to show

$$\begin{aligned} f(n) = \tfrac{1}{2}n^2 - 4n + 100 &= \Omega(n^2) \quad \text{since } \lim_{n\to\infty} \tfrac{n^2}{\frac{1}{2}n^2-4n+100} = 2 \\ &= \Omega(n) \quad \text{since } \lim_{n\to\infty} \tfrac{n}{\frac{1}{2}n^2-4n+100} = 0 \\ &= \Omega(1) \quad \text{since } \lim_{n\to\infty} \tfrac{1}{\frac{1}{2}n^2-4n+100} = 0 \end{aligned}$$

  - For example:

$$\begin{aligned} \text{Insertion\_Sort(n)} &= \Omega(1) \\ &= \Omega(n) \\ &\neq \Omega(n^2) \end{aligned}$$

  - *Theorem: for any two functions $f(n)$ and $g(n)$,* $f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$
- **Asymptotic notations in equations**
  - $2n^2+3n+1 = 2n^2+\Theta(n) \implies 2n^2+3n+1 = 2n^2+f(n)$, where $f(n) = \Theta(n)$. In this case, $f(n) = 3n + 1$
  - $T(n) = O(n) + \Theta(n) + \Omega(n) \Longrightarrow T(n) = \Omega(n)$

• Categories of functions

growth rate	*slowest*				*fastest*
categories	*constant*	*logarithms*	*polynomials*	*exponentials*	*super exponentials*
examples	5	$\log_2 n$	$n^{0.001} - 10$	$2^{n/2}$	$n!, \quad (\log n)^n$
	1	$\log_{10} n$	$n^{1,000,000}$	3^n	n^n

- For logarithm functions, the base does not matter. $\log_{10} n = \Theta(\log_2 n)$ or $\log_e n = \Theta(\log_2 n)$
- For exponential functions, the base matters. $2^{n/2} = \sqrt{2}^n = O(2^n)$, but $2^{n/2} \neq \Theta(2^n)$

– Comparison between polynomials and exponentials: n^a and b^n

  No matter how big the a is and how small the $b > 1$ is, the exponential b^n will always out-grow the polynomial n^a if n approaches ∞.

  For example, 1.000001^n will out-grow $n^{1,000,000}$ when $n \to \infty$

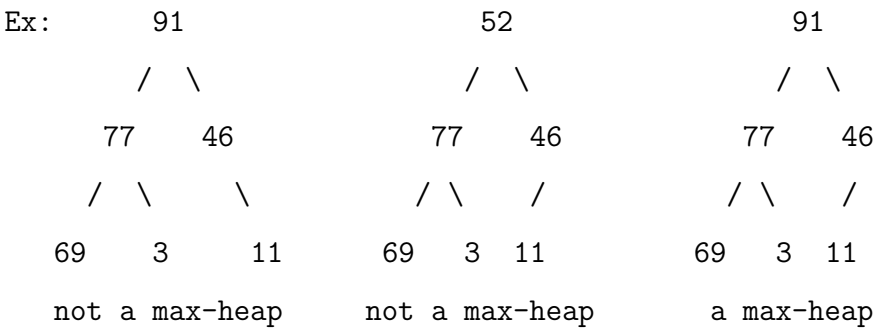
Chapter 6: Heapsort

• Max-Heaps:

– Definition:

To be a binary max-heap, two conditions need to be satisfied.

1. It should be a complete binary tree (all levels, except the last level, must be full and all nodes in the last level need to be as far left as possible).
2. The value of a node should be greater than or equal to its children.



– Array representation for a max-heap:

Assume array index starts at 1. Let `heap-size[A]` stands for the number of elements in the heap stored in the array A.

That is, A[1...heap-size[A]] stores the heap and the root of the heap is stored in A[1].

The `parent-child` relationship between two nodes are represented by the following formulas.

Given a node at array index i, Parent$(i) = \lfloor i/2 \rfloor$

Left$(i) = 2i$

Right$(i) = 2\text{i} + 1$

The example max-heap in this page can be represented in an array as

91	77	46	69	3	11

– The height h of a heap with n nodes: $h = \Theta(\log n)$.

Since a heap with height h will have the minimum and maximum of nodes as follows.

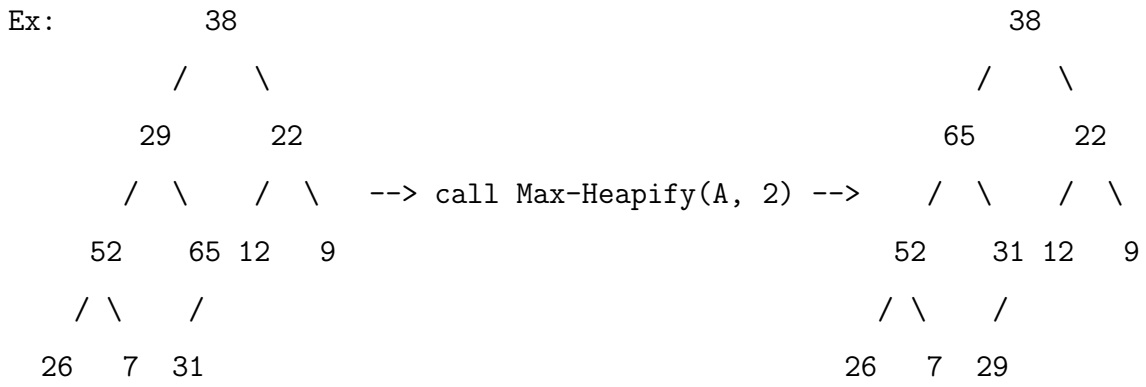
Minimum of $n = 1 + 2 + 2^2 + \ldots + 2^{h-1} + 1 = 2^h$

Maximum of $n = 1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$

From the above two equations, we can derive $h = \Theta(\log n)$.

- **Maintaining the Heap Property:**

```
Max-Heapify(A, i) // heapification downward
   Pre-condition: Both the left and right subtrees of node i are max-heaps
                  and i is less than or equal to heap-size[A]
   Post-condition: The subtree rooted at node i is a max-heap
1. l <-- Left(i)
2. r <-- Right(i)
3. largest <-- i
4. if l <= heap-size[A] and A[l] > A[i]
5.    then largest <-- l
6. if r <= heap-size[A] and A[r] > A[largest]
7.    then largest <-- r
8. if largest != i
9.    then exchange A[i] <--> A[largest]
10.       Max-Heapify(A, largest)
```



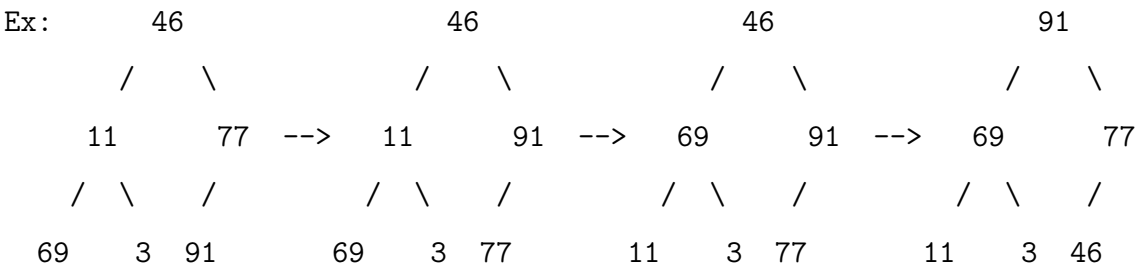
– Running time analysis for Max-Heapify(A, i):

The element A[i] will be swapped down along a tree path. Thus, the running time for the procedure is $O(h)$, where $h = \Theta(\log n)$ is the tree height

• Building a Max-Heap:

Using bottom-up approach to convert an array A[1...n] to a max-heap by calling a sequence of Max-Heapify procedures, starting at the last non-leaf node and ended at the root (backward).

```
Build-Max-Heap(A)
1. heap-size <-- length[A]
2. for i <-- length[A]/2  // integer division
3.     do Max-Heapify(A, i)
```



– Running time analysis for Build-Max-Heap (A):

Call Max-Heapify about $n/2$ times $\implies$ Build-Max-Heap (A) takes $O(n \log n)$. Actually, $O(n \log n)$ is an asymptotically upper bound but not tight. The tight upper bound is $O(n)$.

• The Heapsort Algorithm:

1. Make the input array A to a max-heap by calling Build-Max-Heap(A) procedure. We know A[1] stored the largest element.
2. Exchange A[1] $\leftrightarrow$ A[heap-size[A]] and then decrement heap-size[A] by one.
3. Call Max-Heapify(A, 1) to re-heapify A[1..heap-size[A]].
4. Repeatedly perform Step 2 and Step 3 until heap-size[A] = 1.

```
Heapsort(A)
1. Build-Max-Heap(A)
2. for i <-- length[A] downto 2
3.     do exchange A[1] <--> A[i]
4.        heap-size--
5.        Max-Heapify(A, 1)
```

- Running time analysis of Heapsort(A):
  Heapsort call Build-Max-Heap(A) once and call Max-Heapify(A) $n-1$ times. Thus, the running time is $O(n \log n)$.

- **Priority Queues:**

The root of a max-heap contains the largest value. If we build a priority queue using a max-heap based on the priority values of elements, then the next element to be extracted from the queue is always located at the root.

```
Heap-Extract-Max(A)
1. if heap-size[A] < 1
2.    then error -- heap underflow
3. max <-- A[1]
4. A[1] <-- A[heap-size[A]]          // Give an example.
5. heap-size[A]--                    // This procedure takes O(log n)
6. Max-Heapify(A, 1)
7. return max
```

```
Heap-Increase-Key(A, i, key)
1. if key < A[i]
2.    then error -- new key must be larger than current key
3. A[i] <-- key
4. while i > 1 and A[Parent(i)] < A[i]       // This procedure takes O(log n)
5.       do exchange A[i] <--> A[Parent(i)]
```

```
6.          i <-- Parent(i)
```

```
Max-Heap-Insert(A, key)
1. heap-size[A]++
2. A[heap-size[A]] <-- negative infinity   // This procedure takes O(log n)
3. Heap-Increase-Key(A, heap-size[A], key)
```

- **Comments:**
  - Talk about Exercise 6.5-7.
  - Heapsort uses a special data structure to solve a problem.
  - Heapsort is very similar to selection sort - in each iteration, pick the largest element in the remaining set of elements and put it to the correction position (the "last" position). The difference is that they use different ways to pick the largest element.

Chapter 7: Quicksort

• Description of Quicksort:

It is based on `divide-and-conquer`.

For an array A[p..r],

Divide: Partition the array A[p..r] into two subarrays A[p..q-1] and A[q+1..r] (one of them could be empty) such that

each element of A[p..q-1] $\leq$ A[q] and

each element of A[q+1..r] $\geq$ A[q]

Index q will be computed and returned by this partition procedure

(After partitioning, the element in A[q] is in its correct position)

Conquer: Sort A[p..q-1] and A[q+1..r] recursively.

Combine: No action for combine.

```
Quicksort(A, p, r)
1. if p < r
2.    then q <-- Partition(A, p, r)
3.         Quicksort(A, p, q-1)
4.         Quicksort(A, q+1, r)


Partition(A, p, r)
1. x <-- A[r]
2. i <-- p-1
3. for j <-- p to r-1
4.     do if A[j] <= x                    // work on some examples
5.           then i <-- i+1
6.                exchange A[i] <--> A[j]
7. exchange A[i+1] <--> A[r]
8. return i+1
```

During the execution of the Partition procedure,

1. Elements in the array before index i are less than or equal to the pivot x. That is,
   $A[k] \leq x \quad \text{if } p \leq k \leq i$
2. Elements in the array between the index $i+1$ and $j-1$ are larger than the pivot x. That is,
   $A[k] > x \quad \text{if } i+1 \leq k \leq j-1$
3. Elements in the array after index j are not yet compared.

- **Performance of Quicksort:**

  Depending on whether the partition is "balanced" or not

  If balanced: asymptotically as fast as merge sort: $\Theta(n \log n)$.

  If not balanced: asymptotically as slow as selection sort: $\Theta(n^2)$.

  – Worst case partition:

    This case occurs when partition produces one subarray with $n-1$ elements.

    If this worst case partitioning occurs at each recursive step of the algorithm, then Quicksort takes $T(n) = \Theta(n^2)$ since

    there are n partitions altogether with n, $n-1$, $\ldots, 1$ elements. Thus, the **worst case** running time of Quicksort is $T(n) = n + (n-1) + \ldots + 1 = \Theta(n^2)$.

    For example, if the input array is already sorted in either order.

  – Best case partition:

    This case occurs when partition produces two subarrays with $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$ elements.

    If this best case partitioning occurs at each recursive step of the algorithm, then Quicksort takes $T(n) = \Theta(n \log n)$ since

    there are 1 partition with n elements, 2 partitions with $\frac{n}{2}$ elements, 4 partitions with $\frac{n}{4}$ elements, $\ldots$, n partitions with $\frac{n}{n}$ elements. Thus, the **best case** running time of Quicksort is

    $$\begin{aligned} T(n) &= 1 \cdot \tfrac{n}{1} + 2 \cdot \tfrac{n}{2} + \ldots + n \cdot \tfrac{n}{n} \\ &= 2^0 \cdot \tfrac{n}{2^0} + 2^1 \cdot \tfrac{n}{2^1} + \ldots + 2^{\log n} \cdot \tfrac{n}{2^{\log n}} \\ &= n + n + \ldots + n \quad (n \text{ adds itself } \log n \text{ times}) \end{aligned}$$

$= n \log n$

- Average case running time of Quicksort:

  The **average case** running time of Quicksort is $\Theta(n \log n)$.

  The average case running time analysis of quicksort will be discussed in CS421 with the knowledge of divide-and-conquer.

- **Randomized Versions of Quicksort:**

  Assume all permutations of the input numbers are equally likely. However, this assumption is not realistic because

  - Two special inputs impose the worst-case split: sorted array in either order.
  - These two inputs are very common for lots of applications.

  We need to randomize the input array to reduce the probability of the worst-case.
  Two approaches to randomize the input array.

  1. Before feeding the input to Quicksort algorithm, the input is randomly permuted.
  2. Randomly choose the pivot element at each step in Quicksort.

  The pseudocode for the 2nd approach.

```
Randomized-Partition(A, p, r)
1. i <-- Random(p, r)
2. exchange A[r] <--> A[i]
3. Partition(A, p, r)
```

```
Randomized-Quicksort(A, p, r)
1. if p < r
2.    then q <-- Randomized-Partition(A, p, r)
3.         Randomized-Quicksort(A, p, q-1)
4.         Randomized-Quicksort(A, q+1, r)
```

- Interesting **bolts and nuts problem:**

  There are n pairs of bolts and nuts mixed together. Each bolt has one, and only one, matching nut. Please suggest an efficient algorithm to match all bolts and nuts (A bolt cannot be compared to another bolt. Similarly, a nut cannot be compared to another nut).

Chapter 8: Sorting in Linear Time

• Lower Bound for Sorting:

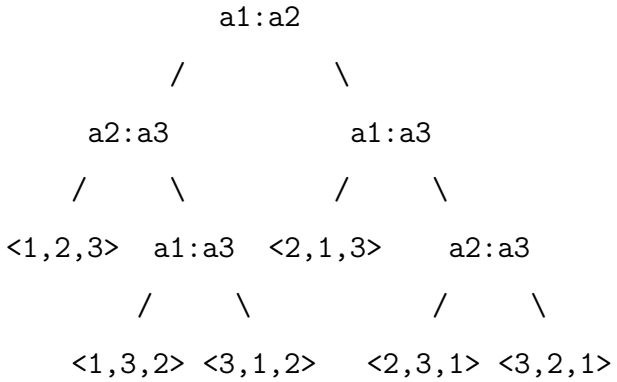
Comparison sort: A sorting algorithm is based only on comparisons between the input elements

Comparison sorts can be viewed abstractly in terms of decision trees.

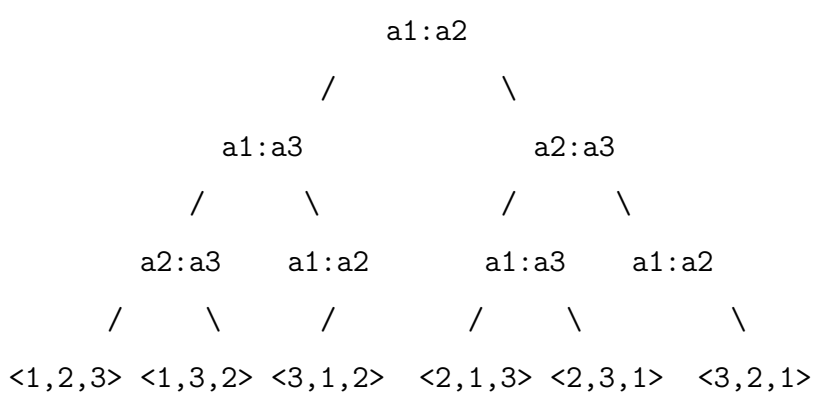
Decision trees: Given an input sequence $< a_1, a_2, \ldots, a_n >$,

- Each internal node is denoted by $a_i : a_j$, for $1 \leq i, j \leq n$.
- Each leaf node is denoted by a permutation $< \pi(1), \pi)2), \ldots, \pi(n) >$.
- each path from the root to a leaf corresponds to an execution of the sorting algorithm for a specific input.
- The left branch of an internal node means $a_i \leq a_j$.
  The right branch for an internal node means $a_i > a_j$.
- There are $n!$ permutations for n elements $\implies$ there are at least $n!$ leaf nodes.

Ex: The decision tree for insertion sort with 3 elements.



Ex: The decision tree for selection sort with 3 elements.



There are $n!$ permutations for n elements $\implies$ the tree has at least $n!$ leave.

Let h be the height of the tree $\implies$ the tree has no more than 2^h leave.

Thus,

$$n! \leq 2^h \implies h \geq \log(n!) \implies h = \Omega(n \log n)$$

The height of a decision tree means the number of comparisons for sorting in the worst-case.

$\implies$ All comparison sorts has a lower bound running time $\Omega(n \log n)$ for the worst-case.

$\implies$ It is impossible to find a new comparison based sorting algorithm that is asymptotically better than merge sort.

However, some non-comparison based sorting algorithms may run in linear time.

- # Counting Sort:

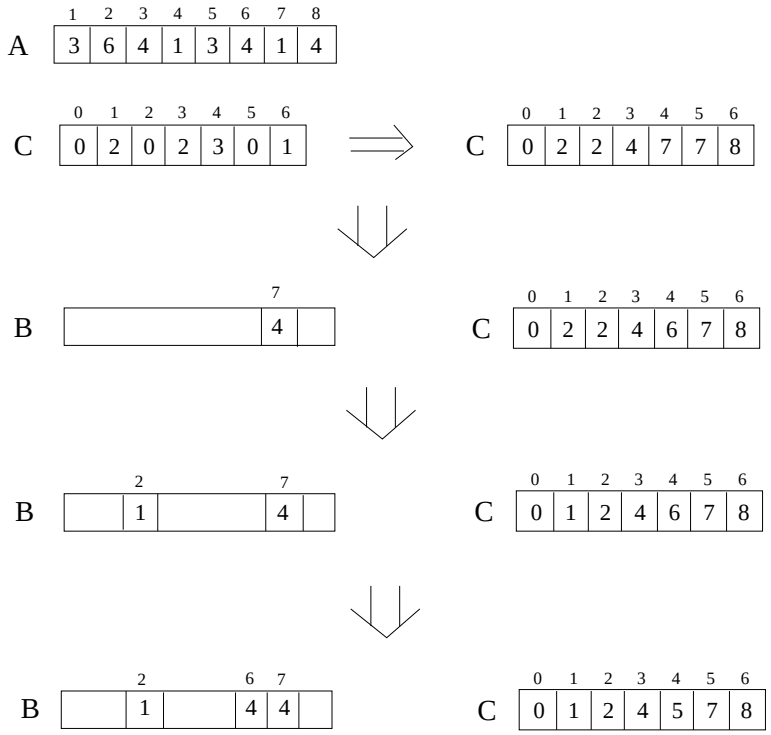
Counting sort assumes that each of the n input elements is an integer within a range $[0..k]$, for some integer k.

An input array $A[1..n]$, an output array $B[1..n]$, and a temporary working storage $C[0..k]$ are necessary for this algorithm. Thus, counting sort does not sort in place.

During the execution of counting sort, $C[i]$ maintains the # of elements less than or equal to i. For each element j in A, put it into B at position $C[j]$.

```
Counting-Sort(A, B, k)
1.  for i <-- 0 to k
2.      do C[i] <-- 0
    // count the # of occurrences of input elements
3.  for j <-- 1 to length[A]
4.      do C[A[j]] <-- C[A[j]] + 1
5.  // C[i] contains the # of elements that is equal to i
6.  for i <-- 1 to k
7.      do C[i] <-- C[i] + C[i-1]
8.  // C[i] now contains the # of elements less than or equal to i
9.  for j <-- length[A] downto 1
10.     do B[C[A[j]]] <-- A[j]
11.        C[A[j]] <-- C[A[j]] - 1
```

Ex:



– Running time analysis:

  Counting-Sort's running time is $\Theta(n+k)$.

  If $k = O(n)$, then $\Theta(n+k) = \Theta(n)$. It's a linear time!

– Counting sort is a **stable** sorting algorithm: elements with the same value in the output array should be in the same order as they do in the input array.

  * Insertion Sort: stable (if no "=" sign in comparison)
  * Selection Sort: stable (if no "=" sign in comparison)
  * Merge Sort: stable (if the "=" sign is in the comparison)
  * Heap Sort: not stable (exchange $A[1] \leftrightarrow A[n]$)
  * Quick Sort: not stable.

    Ex: input: $< 5, 5', 5'', 3, 4 >$ and the output is $< 3, 4, 5'', 5, 5' >$.

• Radix Sort:

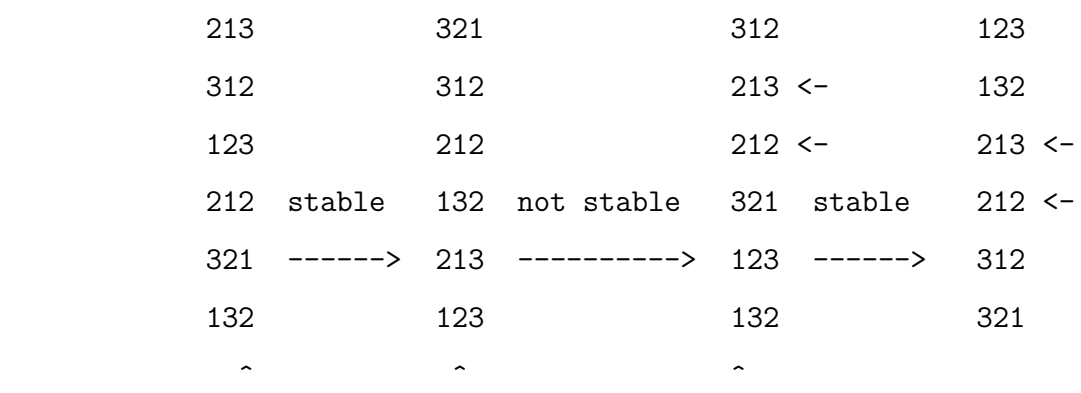
The Radix-Sort sorts the least significant digit first, then the 2nd, ...

The sorting algorithm used to sort each digit should be stable; otherwise Radix-Sort will not work.

Ex:

```
213           321           312           123
312           312           212           132
123           212           213           212
212  stable   132  stable   321  stable   213
321  ------>  213  ------>  123  ------>  312
132           123           132           321
  ^             ^             ^
```

Ex:




```
Radix-Sort(A, d)
1. for i <-- 1 to d
2.     do use a stable sort to sort array A on digit i
```

Two questions:

- Why does the algorithm need to use stable sort to sort each digit?
- Why does the sorting start from sorting the least significant digit first?

Running time analysis:

Suppose all n numbers have d or less digits.

If we use Counting-Sort as the sorting algorithm to sort each digit, then the running time for Radix-Sort is $d \cdot \Theta(n+k) = \Theta(dn+dk)$

If $k = O(n)$ and d is a constant, then the running time becomes $\Theta(n)$.

It's a linear time!

An interesting problem:

Please show how to sort n integers in the range 0 to $n^2 - 1$ in $O(n)$ time.

Chapter 10: Elementary Data Structures

• Stacks and Queues:

– **Stack: last-in, first-out data structure.**

Array representation for a stack:

* An array $S[1..n]$ is allocated to store elements.
* An array attribute top[S] points to an array index in which the most recently inserted element resides. Initially, set top[S] to 0.

```
Stack-Empty(S)
1. if top[S] = 0
2.    then return true                      O(1)
3.    else return false

Push(S, x)       // store at most n elements
1. if (top[S]+1 > length[S])
2.    then error -- stack overflow          O(1)
3.    else top[S] <-- top[S] + 1
4.         S[top[S]] <-- x

Pop(S)
1. if Stack-Empty(S)
2.    then error -- stack underflow
3.    else top[S] <-- top[S] - 1            O(1)
4.         return S[top[S] + 1]
```

Give some examples.

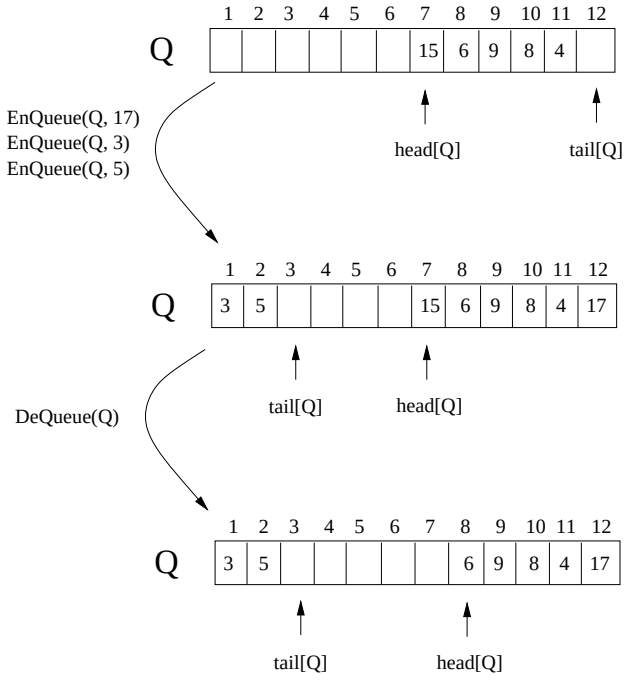
– **Queue: first-in, first-out data structure.**

Array representation for a queue:

* An array $Q[1..n]$ is allocated to store elements. The array will be considered as a circular array.
* An array attribute head$[Q]$ points to an array index in which the earliest inserted element resides. Another array attribute tail$[Q]$ points to an array index in which a new elements should be inserted.
* An initial empty queue: head$[Q] = 1$ and tail$[Q] = 1$.

```
Queue-Empty(Q)
1. if head[Q] = tail[Q]
2.    then return true                        O(1)
3.    else return false


EnQueue(Q, x)      // store at most n-1 elements
1. if (head[Q] = ((tail[Q]+1) mod length[Q]))
2.    then error -- queue overflow
3.    else Q[tail[Q]] <-- x                   O(1)
4.         if tail[Q] = length[Q]
5.            then tail[Q] <-- 1
6.            else tail[Q] <-- tail[Q]+1
                                                    Give some examples
DeQueue(Q)
1. if Queue-Empty(Q)
2.    then error -- queue underflow
3.    else x <-- Q[head[Q]]
4.    if head[Q] = length[Q]                  O(1)
5.       then head[Q] <-- 1
6.       else head[Q] <-- head[Q]+1
7. return x
```

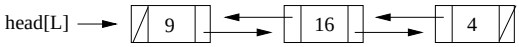
Ex:



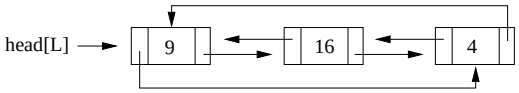
- **Linked Lists:**
  - Linked lists provide the dynamic storage v.s the fixed storage of arrays.
  - In a doubly linked list L, an element (object) x should contain at least 3 fields:
    1. key[x] return the key value of x.
    2. next[x] return the pointer to the next element after x.
    3. prev[x] return the pointer to the previous element before x.
  - For the list L, an attribute head[L] points to the first element in L.
  - In a singly linked list L, an element x has at least two fields: key[x] and next[x].
  - For a linked list, it may be singly or doubly, sorted or not sorted, circular or non-circular.

Assume **doubly, unsorted and non-circular** linked lists are used for the following procedures.

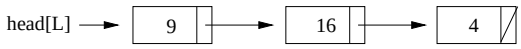
doubly, non−circular:



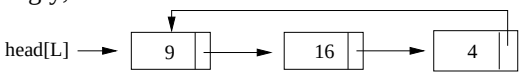
doubly, circular:



singly, non−circular:



singly, circular:




```
// Searching a linked list: go through the list one element at a time.
//                     return a node pointer if found; otherwise return nil.
List-Search(L, k)  // k is a key value
1. x <-- head[L]
2. while x != nil and key[x] != k         // Theta(n) in worst-case
3.       do x <-- next[x]
4. return x

// Insertion: insert a new element x in front of the list
List-Insert-Front(L, x)
1. next[x] <-- head[L]
2. if head[L] != nil
3.    then prev[head[L]] <-- x            // O(1)
4. head[L] <-- x
5. prev[x] <-- nil
```

```
// Deletion: remove an element x from the list.
//           Assume x is indeed in the list.
List-Delete(L, x)
1. if prev[x] != nil
2.    then next[prev[x]] <-- next[x]
3.    else head[L] <-- next[x]            // O(1)
4. if next[x] != nil
5.    then prev[next[x]] <-- prev[x]
```

To delete a given key k rather than a given node from a list, it requires to call List-search(L, k) to find the node pointer x, and then call List-Delete(L, x).
Totally, it takes $\Theta(n)$ time in the worst-case.

- **Representing Rooted Trees:**
  - Trees are composed by tree nodes.
  - Each tree node has a key field and some other pointer fields pointing to other nodes. Number of pointer fields in a tree node may be different for different types of trees.
  - A tree T has an attribute root$[T]$: a pointer to the root of the tree.

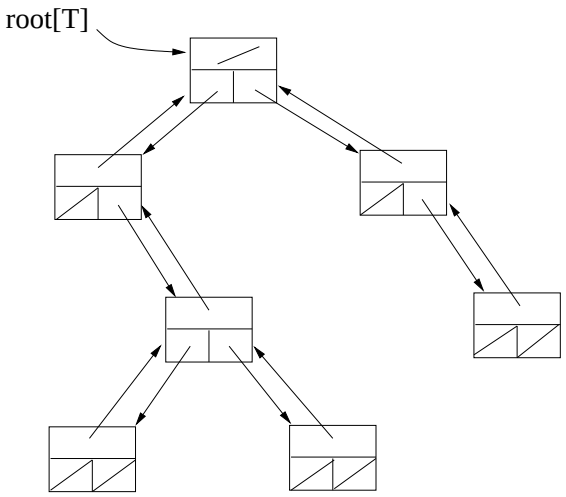
Binary Trees: For each node x, there are 3 pointer fields

- p$[x]$ is a pointer to x's parent.
- left$[x]$ is a pointer to x's left child.
- right$[x]$ is a pointer to x's right child.

See next page for example.

Rooted Trees with Unbounded branches:

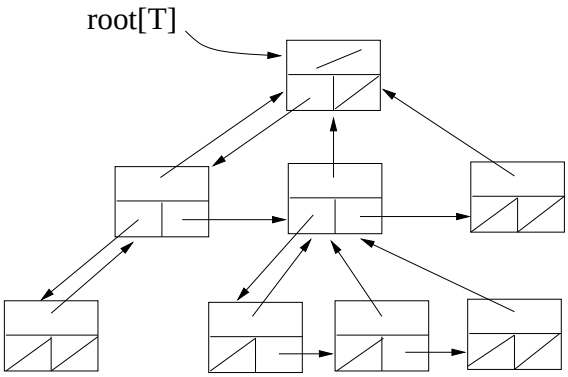
- each node can have any number of children.
- Using left-child, right-sibling representation.



The structure of a binary tree

– 3 pointer fields for each node x.

  * p[x] is a pointer to x's parent.
  * left[x] is a pointer to x's left-most child.
  * right[x] is a pointer to the sibling of x immediately to the right.



The structure of a rooted tree with unbounded branches

Chapter 11: hash Tables

To search an element in a hash table:

Worst-case: $\Theta(n)$ (no better than a linked list).

Average-case: $O(1)$.

• Direct-address Tables:

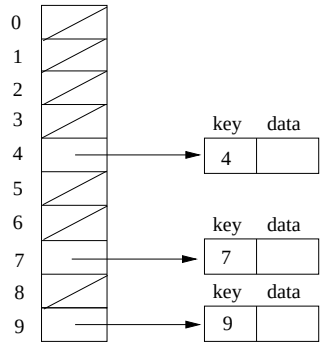
– Each element to be stored has a unique key.

– Suitable for applications with small set U (the set of all possible keys).

– Suppose $U = \{0, 1, \ldots, m\}$. To store a dynamic set of elements, a direct-address table $T[0...m-1]$ is allocated, where

$$T[i] = \begin{cases} x & \text{if the element with key } i \text{ is stored, where } x \text{ points to the element} \\ nil & \text{otherwise} \end{cases}$$

Ex: $U = \{0, 1, \ldots, 9\}$ and the set of elements (keys) stored $= \{4, 7, 9\}$.




```
Direct-Address-Search(T, k)
1. return T(k)                  // O(1)

Direct-Address-Insert(T, x)
1. T[key[x]] <-- x              // O(1)

Direct-Address-Delete(T, x)
1. T[key[x]] <-- nil            // O(1)
```

• Hash Tables:

– It's impractical for direct addressing if U is large.

Ex: If U is the set of all possible SSN, then T requires 10^9 entries.

– We can reduce the memory requirement to $\Theta(n)$ and still have $O(1)$ average searching time by hashing technique, where n is the number of elements stored.

```
Direct-addressing: an element with key k is stored in entry k.
Hashing          : an element with key k is stored in entry h(k),
                   where h: hash function and h(k): hash value.
```

For a hash table $T[0..m-1]$, the hash function $h : U \rightarrow \{0, 1, \ldots, m-1\}$.

– The hashing technique is for applications that $|U| >> m$ (the size of table T).
$\Longrightarrow$ collisions may occur (two keys k_1, k_2 that $h(k_1) = h(k_2)$).

Solution for collision:

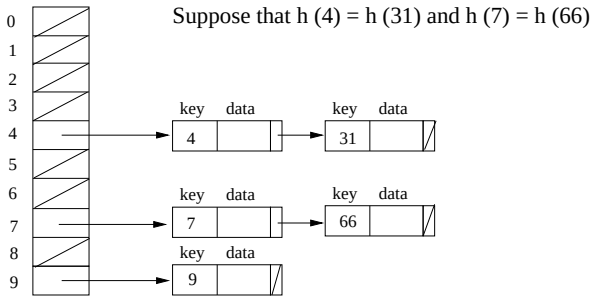
* avoid the collisions altogether: impossible.
* choose h to be "random" to minimize the # of collisions.

Two topics to research on:

1. What is a good choice of h?
2. How to resolve the collisions?

– Collision resolution by chaining:

We put all elements colliding on one entry into a linked list. For example:



```
Chained-Hash-Insert(T, x)
1. insert x at the front of list T[h(key[x])]        // O(1)
```

```
Chained-Hash-Search(T, k)
1. search for an element with key k in list T[h(k)]
```

// Average time: $\Theta(1+\alpha)$, where $\alpha = n/m$ is the load factor of the table T.

```
Chained-Hash-Delete(T, x)                // O(1): if doubly linked list.
1. delete x from the list T[h(key[x])]   // Same running time as search
                                         //      if singly linked list.
```

- **Hash Functions:**
  - Good hash function: **simple uniform hashing**.
    Each key is equally likely to hash to any of the m entries.
  - Assumption: Let the domain of keys is the set of natural numbers.
    If the keys are not numbers (e.g., strings), they should be mapped to numbers before hashing.
  - **The division method:** $h(k) = k \bmod m$.
    - the division method is almost simple uniform.
    - m should not be a power of 2 (if $m = 2^p$, then $h(k)$ is the p low-order bits of k. It is better to make hash function depends on all bits of the key).
    - Good choice for m: primes that are not too close to exact powers of 2.
      Ex: To hold 2000 keys and 3 elements examined in average for an unsuccessful search. The collision is resolved by chaining. What the table size m should be?
      Sol: $m = 701$, since 701 is a prime and $701 \simeq 2000/\alpha$, and also 701 is not close to any power of 2.

- **Open Addressing:**
  - Another collision resolution technique.
  - Each entry in the table T can store at most one element, rather than a linked list of elements in chaining. Thus, $\alpha \leq 1$.
  - To perform insertion, we successively examine, or probe, the hash table until we find an empty entry.
  - The probe sequence (the sequence of entries examined) depends on the probing techniques used. There are 3 popular probing techniques.

Linear Probing: The probe sequence is

$$h(k,i) = (h'(k) + i) \bmod m, \ for\ i = 0, 1, \ldots, m-1$$

That is, $T[h'(k)],\ T[h'(k)+1], \ldots, T[m-1],\ T[0],\ T[1], \ldots, T[h'(k)-1]$.

This technique suffers on the **primary clustering** problem: long runs of occupied entries.

- Why linear probing usually results in primary clustering?
- What's wrong with the primary clustering?

  The following example shows that clustering will result in longer search time.

  Let $n = m/2$, i.e., $\alpha = 1/2$.

  case 1: If every odd entry is filled and every even entry is empty,
  then an unsuccessful search will take 1.5 probes in average since
  based on the general formula of expected value
  $E(v) = \sum_{\forall events\ i} P(i) \cdot V(i)$, we have
  expected number of probes $= \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 = 1.5$

  case 2: If first n entries are filled (the second n entries are empty),
  then an unsuccessful search will have an expected number of probes
  $= \frac{1}{m} \sum_{i=1}^{n} (i+1) + \frac{n}{m} = \frac{n+5}{4}$ probes

  $\frac{n+5}{4} \gg 1.5$. Thus, clustering increases the searching time.

Quadratic Probing: The probe sequence is

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \quad where\ c_1, c_2 \neq 0, i = 0, 1, \ldots, m-1$$

c_1, c_2 and m need to be chosen carefully to **fully utilize** the table T (see problem 11.3).

- Fully utilize the table T: Given any key k, the probe sequence of k can check the entire table.

If two keys k_1, k_2 have the same initial probe position, then they will have the same probe sequence. That is,
$h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i), \forall\ i$.
This phenomenon leads to a milder form of clustering: **secondary clustering**.

Double Hashing: The probe sequence is

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m, \ for\ i = 0, 1, \ldots, m-1$$

- The probe sequence depends in two ways upon the key k.
  In this case, if two keys have the same initial probe position, they may not have the same probe sequence.
  Thus, double hashing do not suffer on the secondary clustering.
- In order to fully utilize the table, given any k, $h_2(k)$ **must be always relatively prime to** m.
  If $d = \gcd\{h_2(k), m\}$, then only $\frac{1}{d}$ th of table will be searched.
  Two different approaches to make $h_2(k)$ always relatively prime to m.
  1. Let $m = 2^p$, where p is some positive integers.
     Design h_2 so that it always produces an odd number.
  2. Let m be a prime number.
     Design h_2 so that it always produces a positive integer less than m.
     Ex: Let m be a prime and let

$$\begin{cases} h_1(k) = k \bmod m \\ h_2(k) = 1 + (k \bmod m'), \text{where } m' = m-1 \text{ or } m-2 \end{cases}$$

Pseudocode: If an entry is occupied by an element but the element has been deleted later, we need to set a flag "deleted" to this entry.

```
Hash-Search(T, k)
1. i <-- 0
2. repeat j <-- h(k, i)
3.        if T[j] = k
4.           then return j
5.           else i <-- i+1
6. until T[j] = nil or i = m
7. return nil

Hash-Insert(T, k)
1. i <-- 0
2. repeat j <-- h(k, i)
3.        if T[j] = nil or deleted
4.           then T[j] = k
5.                return j
6.           else i <-- i+1
7. until i = m
8. error - hash table overflow

Hash-Delete(T, k)
1. i <-- 0
2. repeat j <-- h(k, i)
3.        if T[j] = k
4.           then T[j] <-- deleted
5.                return j
6.           else i <-- i+1
7. until T[j] = nil or i = mi = m
8. error - k is not in the table
```

Analysis of hashing by chaining and open address hashing:

Theorem 11.1 *For hashing by chaining, an unsuccessful search takes time* $\Theta(1+\alpha)$ *in average, under the simple uniform hashing assumption.*

Theorem 11.2 *For hashing by chaining, a successful search takes time* $\Theta(1+\alpha)$ *in average, under the simple uniform hashing assumption.*

Theorem 11.6 *For open-address hashing, the expected # of probes in an unsuccessful search is at most* $1/(1-\alpha)$*, assuming uniform hashing.*

Theorem 11.8 *For open-address hashing, the expected # of probes in a successful search is at most* $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$*, assuming uniform hashing.*

Without giving a specific table state, the above theorems can be applied to find the average search time. However, if a specific table state is given, the way to find the average search time is different.

Ex:

Suppose a table T with 10 entries stores 6 elements in $T[0]$, $T[1]$, $T[4]$, $T[5]$, $T[7]$, $T[9]$.

What is the expected # of probes for an unsuccessful search?

Assuming linear probing is used and no "deleted" flag exists.

(Hint: using the general formula for computing the expected value)

Tree Structure and Applications

- **Recursive structure:**
  - A tree consists of a set of tree nodes.
  - These tree nodes are connected by two types of pointers: parent pointers and child pointers.
  - Recursive structure: each tree node may contain pointers pointing to other tree nodes.
  - Binary trees: two child pointers - leftchild and rightchild.
  - Because of the recursive structure, some tree properties can be computed by recursive methods such as tree height, number of nodes, sum of the tree, etc.
- **Tree Terminology:**
  - Binary trees: two child pointers
  - Ternary trees: three child pointers
  - General trees: each node can have any number of child pointers (i.e., $0, 1, 2, \ldots,$ or ∞ children)
  - Binary tree traversals: pre-order, in-order, and post-order traversals
    - Pre-order traversal: for each node, (1) print the node first; (2) print the left subtree; and then (3) print right subtree.
    - In-order traversal: for each node, (1) print the left subtree first; (2) print the node; and then (3) print right subtree.
    - Post-order traversal: for each node, (1) print the left subtree first; (2) print the right subtree; and then (3) print the node.
    - Give some examples in class.
  - Tree height h: the length of the longest path from the root to leaf nodes.
    - An empty tree has height -1
    - A single node tree has height 0

* Thus, the pseudocode for finding the tree height is

```
Tree-height(node r)
1. if (r == null)
2.    then return -1;
3. leftHeight = Tree-height(r.leftchild());
4. rightHeight = Tree-height(r.rightchild());
5. if leftHeight >= rightHeight
6.    then return 1 + leftHeight;
7.    else return 1 + rightHeight;
```

– Each node in a tree has a depth: the number of links from the root to the node.

* The depth of the root is 0
* The depth of a deepest leaf node is the height of the tree
* Given a node n in a tree with the root r, the depth of the node n in the tree is

```
Depth(node r, node n)
1. if (r == null) or (n == null)
2.    then return error;
3. d = 0;
4. while (n != r)
5.       do n = n.parent(n);
6.          d++;
7. return d;
```

– Leftmost node: starting from the root, always go to left child until reaching a node without a left child.

– Rightmost node: starting from the root, always go to right child until reaching a node without a right child

– Number of nodes in a tree, rooted at a node r

```
Tree-size(node r)
```

```
1. if (r == null)
2.    then return 0;
3. return 1 + Tree-size(r.leftchild()) + Tree-size(r.rightchild());
```

- Sum of a tree, rooted at a node r: the pseudocode is similar to the Tree-size.

• Application: Expression Trees:

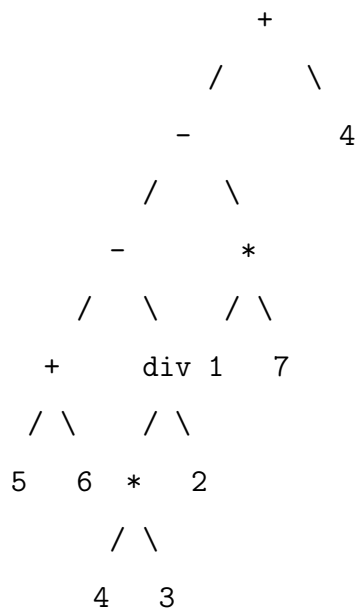
- Arithmetic expressions:
  - Arithmetic infix-expression: binary operator appears between it's two operands
    For example, $5 + 6 - 4 * 3 / 2 - 1 * 7 + 4$
  - Arithmetic prefix-expression: binary operator appears right before it's two operands
    For example, $+ - - + 5 \ 6 / * 4 \ 3 \ 2 * 1 \ 7 \ 4$
  - Arithmetic postfix-expression: binary operator appears right after it's two operands
    For example, $5 \ 6 + 4 \ 3 * 2 / - 1 \ 7 * - 4 +$
- Why prefix and postfix expressions?
  Because arithmetic postfix expressions can be easily evaluated by a computer algorithm using a stack. Given a postfix expression, scan through the expression token by token
  - If next token read is an operand, push it to the stack
  - If next token is an operator, pop two operands from the stack and apply the operator to these two operands, and then push back the result to the stack.
  - At the end, if the stack contains a single value, it means the expression is a valid expression and the value is the evaluation result of the expression. Otherwise, the expression is invalid.

  Give examples in class.
- Converting from infix to prefix and postfix expressions: human algorithm
  Given an infix expression such as $5 + 6 - 4 * 3 / 2 - 1 * 7 + 4$, we can

* fully parenthesize the expression to $((((5+6)-((4*3)/2))-(1*7))+4)$
* Converting to the prefix expression: (1) For each operator, replace the corresponding opening parenthesis by the operator; (2) Remove all the closing parentheses.
* Converting to the postfix expression: (1) For each operator, replace the corresponding closing parenthesis by the operator; (2) Remove all the opening parentheses.

– Converting from infix to prefix and postfix expressions: computer algorithm
Given an infix expression such as 5 + 6 − 4 ∗ 3 / 2 − 1 ∗ 7 + 4, we can

* convert the arithmetic expression to an expression tree, where each internal node represents an operator with its left subtree representing the left operand and its right subtree representing the right operand.
For example, the expression tree for the given expression is



* The arithmetic prefix expression can be formed by performing a pre-order traversal on the expression tree.
* Similarly, the arithmetic postfix expression can be formed by performing a post-order traversal on the expression tree.

- **Application: Huffman Trees**
  - Purpose: data compression/encoding

    In a ASCII text file, each char requires a byte (8 bits).

    To reduce the size of the file, we can encode each char by Huffman codes.
  - Huffman codes: Depending on the frequencies of all characters appearing in a document, we can use less then 8-bit to represent the chars with higher frequencies and use more than 8-bit for those less-frequent chars.

    As a result, the overall size of the encoded file will be smaller.
  - Huffman encoding process:
    - Huffman tree construction:
      1. All the characters are in leaf nodes. For each internal node, the left branch means 0 and the right branch means 1. The Huffman code for each character is the 0-1 bit-string represented by the path from the root to the corresponding leaf node.
      2. The location of each character depends on the expected frequency of that character appearing in a document.
      3. At beginning, we can construct a Huffman tree from a given text file by counting the occurrences of all characters in the file. Let's call it a Huffman Tree Construction File (HTCF).
      4. HTCF needs to be shared by two communicating parties so that they can construct the same Huffman Tree for successful encoding and decoding.

      Tree Construction algorithm:

      1. Count the frequencies of all characters in a given file (HTCF).
      2. Construct a priority queue containing a set of Huffman trees, where smaller weight of the tree (sum of frequency counts of all chars in the tree) has a higher priority
      3. At beginning, the priority queue contains all single-node Huffman trees, where each Huffman tree contains only one character.

4. Extract two Minimum Huffman Trees from the priority queue
5. Combine these two Huffman trees into one tree by creating a new root and these two Huffman trees will be left and right subtrees of the new root
6. Insert the combined Huffman tree back to the priority queue
7. Repeat steps 4, 5 and 6 until only one Huffman tree left in the priority queue

* File encoding:

  Based on the constructed Huffman tree, we can create a mapping table of characters and their bit-strings.

  The mapping table then can be used to encode other files.

* File decoding:

  Upon receiving an encoded file, the receiving end can based on the same mapping table to decode the file.

Given Huffman tree examples in class.

Chapter 12: Binary Search Trees

• What is a Binary Search Trees:

A binary search tree is a binary tree with the following two properties:

1. If a node y is in the left subtree of a node x, then $key[y] \leq key[x]$.
2. If a node y is in the right subtree of a node x, then $key[y] > key[x]$.

Inorder tree walk can print out all the keys in a binary search tree in a sorted order.

```
Inorder-Tree-Walk(x)
1. if x != nil
2.    then Inorder-Tree-Walk(left[x])     // Theta(n)
3.         print x
4.         Inorder-Tree-Walk(right[x])
```

• Querying a Binary Search Tree:

Querying a Binary Search Tree: retrieve information from the tree without modifying the tree.

Query operations of a binary search tree include Search, Minimum, Maximum, Successor, Predecessor, ...

All of the above operations take $O(h)$, where h is the height of the tree.

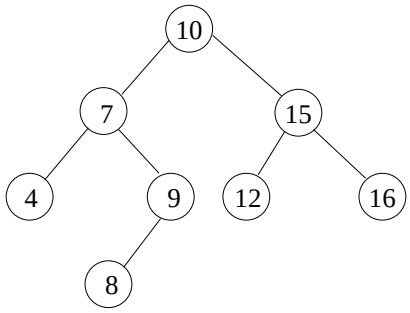
```
BST-Search(x, k) // x points to the root
1. if x = null or k = key[x]
2.    then return x
3. if k < key[x]
4.    then return BST-Search(left[x], k)
5.    else return BST-Search(right[x], k)
// The nodes searched during the recursion form a path from
// the root downward. Thus, running time is O(h)

Iterative-BST-Search(x, k)
```

```
1. while x != nil and k != key[x]
2.       do if k < key[x]
3.             then x <-- left[x]
4.             else x <-- right[x]
5. return x

BST-Minimum(x)
1. while x != nil and left[x] != nil
2.       do x <-- left[x]                 // O(h)
3. return x

BST-Maximum(x)
1. while x != nil and right[x] != nil
2.       do x <-- right[x]                // O(h)
3. return x
```

The successor of a node x is the node y, where

$$y = \begin{cases} \text{Minimum}(\text{right}[x]) & \text{if right}[x] \neq \text{nil} \\ \text{The lowest ancestor of } x \text{ whose left child is also an ancestor of } x & \text{if right}[x] = \text{nil} \end{cases}$$

Ex:



case 1: the successor of 10
= Minimum (right[10])
= 12

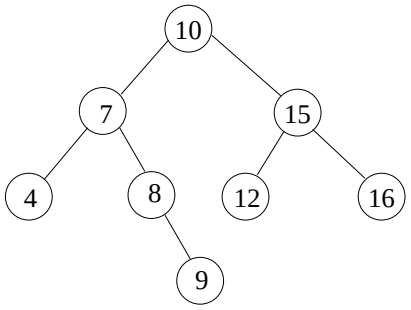
case 2: the successor of 9
= 10

```
BST-Successor(x)
1. if right[x] != nil
2.    then return BST-Minimum(right[x])
3. y <-- p[x]
4. while y != nil and x = right[y]
5.       do x <-- y
6.          y <-- p[y]
7. return y

// O(h): since we either follow a path downward (1st case) or
//       a path upward (2nd case)
```

The predecessor of a node x is the node y, where

$$y = \begin{cases} \text{Maximum}(\text{left}[x]) & \text{if left}[x] \neq \text{nil} \\ \text{The lowest ancestor of } x \text{ whose right child is also an ancestor of } x & \text{if left}[x] = \text{nil} \end{cases}$$

Ex:



case 1: the predecessor of 10
= Maximum (left[10])
= 9

case 2: the predecessor of 8
= 7
the predecessor of 12
= 10

```
BST-Predecessor(x)
1. if left[x] != nil
2.    the return BST-Maximum(left[x])
3. y <-- p[x]
4. while y != nil and x = left[y]
5.       do x <-- y
6.          y <-- p[y]
7. return y

// O(h): same argument as BST-Successor(x)
```

- **Insertion and Deletion:**

  BST-Insert(T, z) is to insert a node z to a binary search tree T, where key$[z] = v$, left$[z]$ = right$[z]$ = p$[z]$ = nil initially.

  BST-Insert always inserts a new node z as a leaf node.

```
1.  y <-- nil                      // y is the parent of x
2.  x <-- root(T)                  // x keep track of a path
3.  while x != nil
4.        do y <-- x
5.           if key[z] <= key[x]
6.              then x <-- left[x]
7.              else x <-- right[x]
8.  p[z] <-- y
9.  if y = nil
10.    then root[T] <-- z
11.    else if key[z] <= key[y]
12.            then left[y] <-- z
13.            else right[y] <-- z
```

  Steps 3 - 7: find the position to insert the new node.

  Steps 8 - 13: set the pointers to insert the new node.

  Takes $O(h)$ time: trace downward from the root to a leaf to find the position to insert.

  Give an example.

To delete a node z from a binary search tree, there are 3 cases to consider.

1. If z does not have any children, then

   re-set $p[z]$: replace z with nil as $p[z]$'s child.

2. If z has only one child, then

   takes out z by making a new link between its child and its parent.

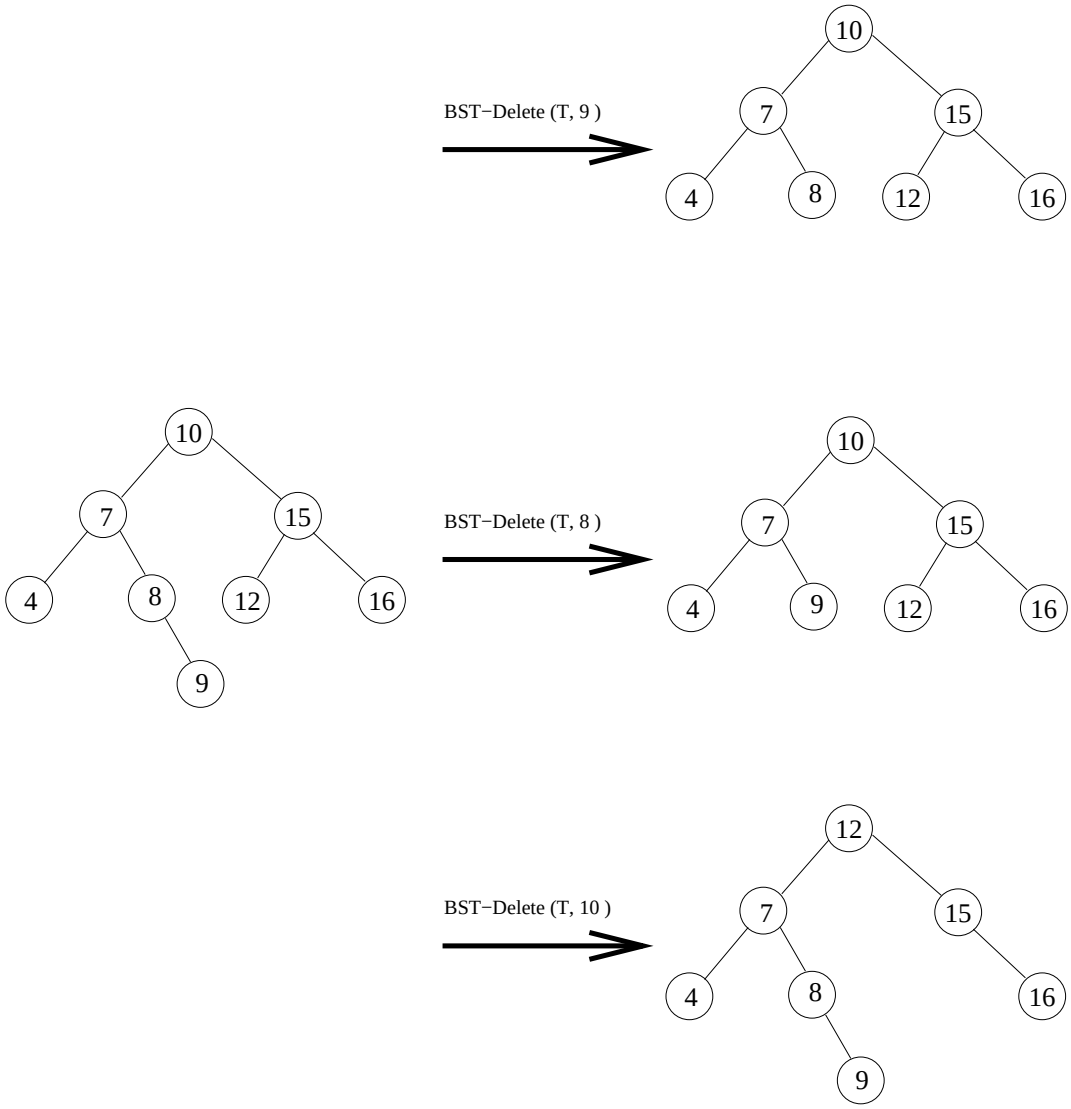
3. If z has two children, then

   takes out z's successor y (y has no left child) and copy the contents of y to z.

```
BST-Delete(T, z)
1.  if left[z] =  nil or right[z] = nil   // y is the node to be removed
2.     then y <-- x                       // where y has at most one child
3.     else y <-- BST-Successor(z)
4.  if left[y] != nil
5.     then x <-- left[y]              // x is the non-nil child of y, or
6.     else x <-- right[y]             // x = nil if y has no children
7.  if x != nil
8.     then p[x] <-- p[y]
9.  if p[y] = nil
10.    then root[T] <-- x                // Steps 7 - 13: remove y
11.    else if y =left[p[y]]
12.            then left[p[y]] <-- x
13.            else right[p[y]] <-- x
14. if y != z
15.    then key[z] <-- key[y]
16.         copy other fields from node y to node z
17. return y
```

Take $O(h)$ time: case 1 or 2 take $\Theta(1)$, but case 3 takes $O(h)$.

Examples are on next page.


BST−Delete (T, 9)
10
7
15
4
8
12
16
10
7
15
4
8
12
16
9
BST−Delete (T, 8)
10
7
15
4
9
12
16
BST−Delete (T, 10)
12
7
15
4
8
16
9

Balanced Binary Search Trees

- **What is a Balanced Binary Search Trees:**

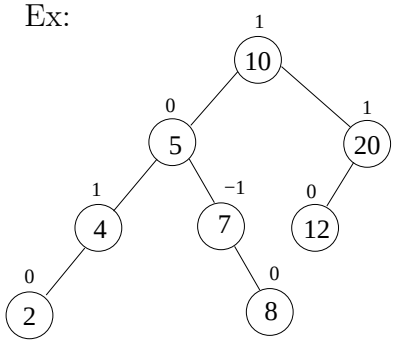
  It is a binary search tree with height $\Theta(\log n)$, where n is the # of nodes in the tree.

- **AVL Trees:**

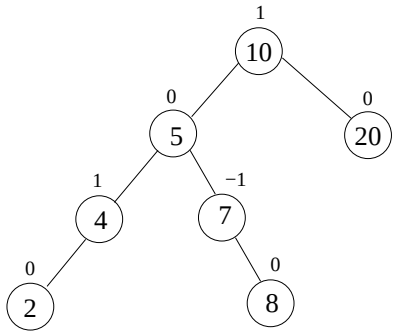
  – An AVL tree is a BST.

  – For each node, the difference between the height of its left and right subtrees is either $+1, 0$, or -1.

  Ex:



An AVL tree

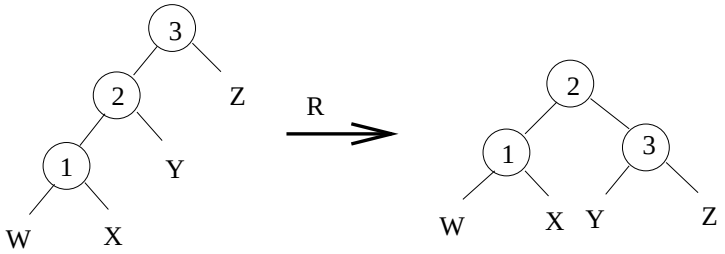


not an AVL tree

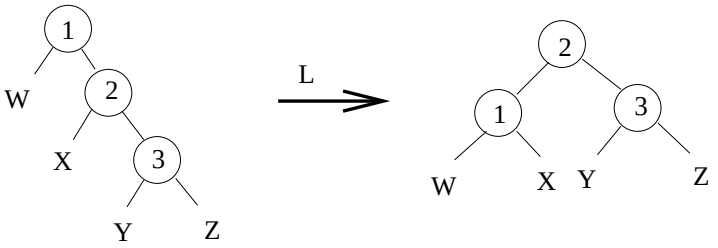
  – Technique to maintain AVL tree: 4 different rotations.

  See next page.

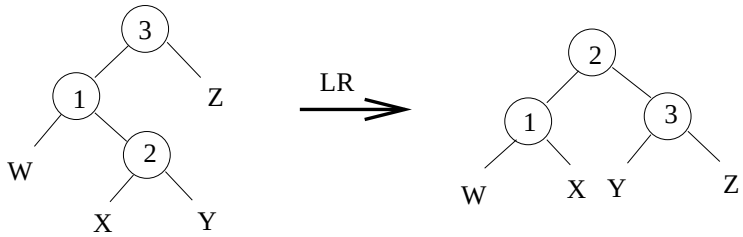
R Rotation:



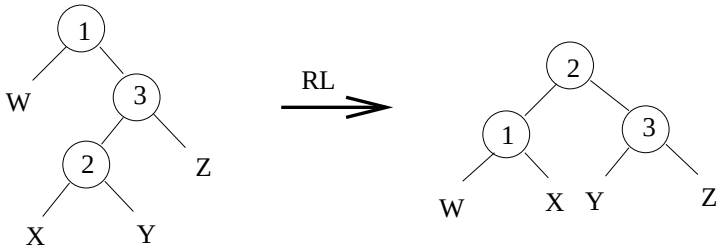
L Rotations:



LR Rotation:



RL Rotation:



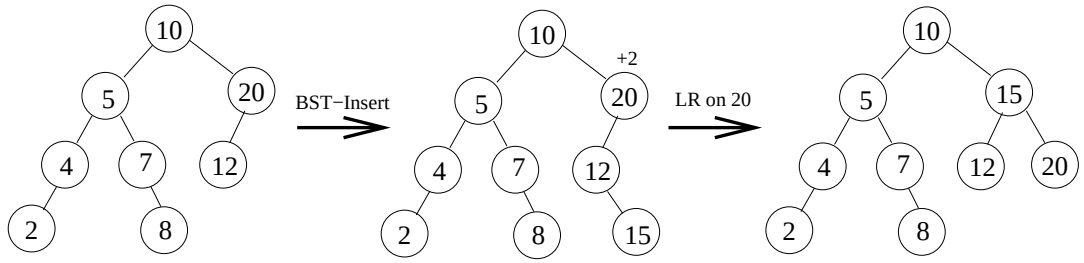
– **AVL Insertion:** Two steps.

1. Perform the BST-Insert(T, z).
2. Perform re-structuring through rotations if necessary.
   * A rotation is performed when a subtree rooted at a node whose **balanced factor** becomes $+2$ or -2 after BST-Insert. If there are several such unbalanced nodes, we rotate at a node A that is closest to the newly inserted leaf.
   * To decide which rotation to use, we start from the node A and test the balanced factor in the following way.

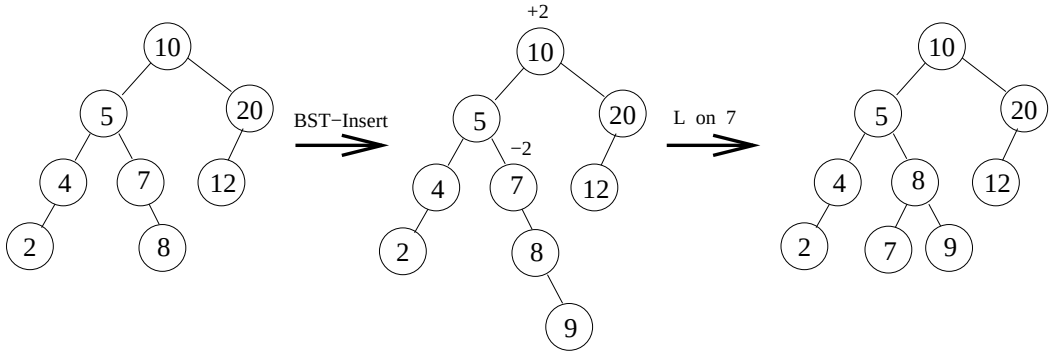
$$+2 \xrightarrow{L} +1/0 \xrightarrow{L} : \text{R Rotation.}$$
$$+2 \xrightarrow{L} -1 \xrightarrow{R} : \text{LR Rotation.}$$
$$-2 \xrightarrow{R} +1 \xrightarrow{L} : \text{RL Rotation.}$$
$$-2 \xrightarrow{R} -1/0 \xrightarrow{R} : \text{L Rotation.}$$

Ex:

Insert 15:



Insert 9:



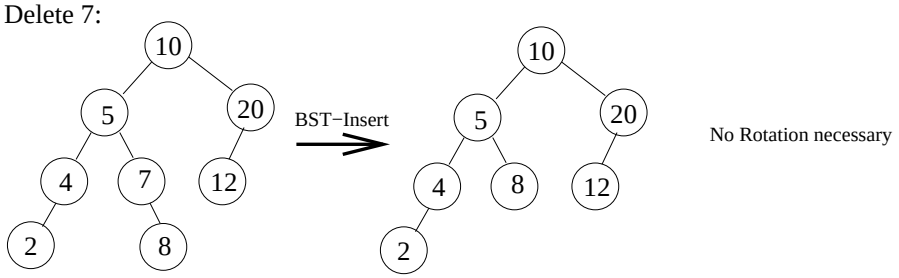
– **AVL Deletion:** Two steps.

1. Perform the BST-Delete(T, z).
2. Perform re-structuring through rotations if necessary.
   * A rotation is performed when a subtree rooted at a node whose **balanced factor** becomes $+2$ or -2 after BST-Insert. If there are several such unbalanced nodes, we rotate at a node A that is closest to the newly deleted node (i.e., the node has been physically removed).
   * To decide which rotation to use, we start from the node A and test the balanced factor in the following way.

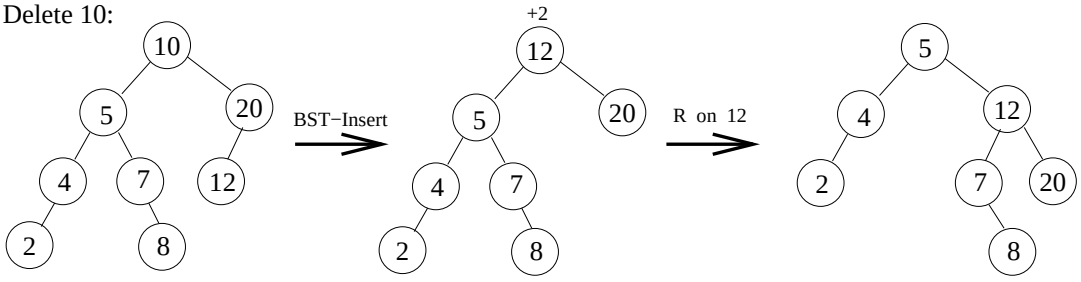
$$+2 \xrightarrow{L} +1/0 \xrightarrow{L} : \text{R Rotation.}$$
$$+2 \xrightarrow{L} -1 \xrightarrow{R} : \text{LR Rotation.}$$
$$-2 \xrightarrow{R} +1 \xrightarrow{L} : \text{RL Rotation.}$$
$$-2 \xrightarrow{R} -1/0 \xrightarrow{R} : \text{L Rotation.}$$

Ex:

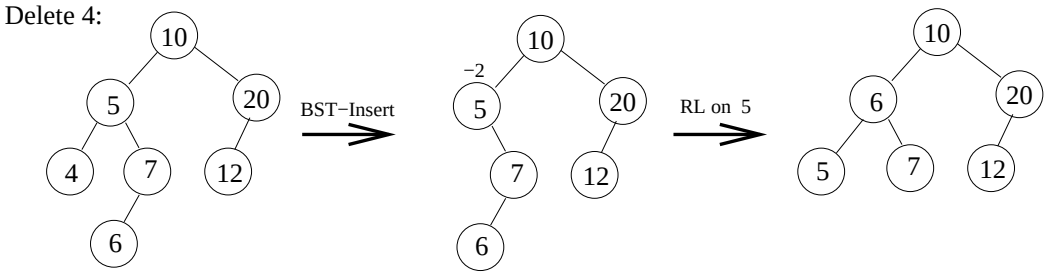
Delete 7:



Delete 10:



Delete 4:



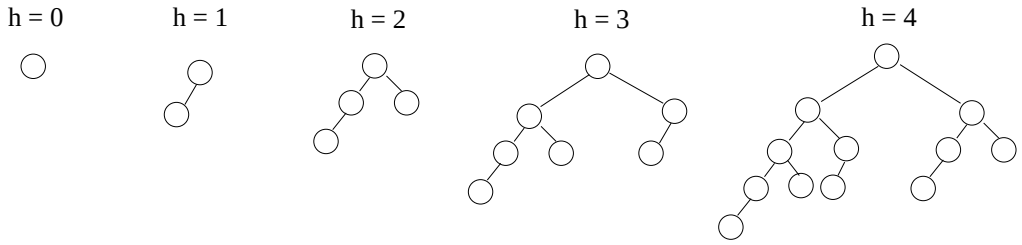
– **Height of AVL Trees:** $\Theta(\log n)$

Given an AVL tree with height h.

The maximum # of nodes: the tree is full.

$$\begin{aligned} & n \leq 2^0 + 261 + \ldots + 2^h = \textstyle\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \\ \Longrightarrow\ & n \leq 2^{h+1} - 1 \\ \Longrightarrow\ & h \geq \log(n+1) - 1 \\ & = \Omega(\log n) \end{aligned}$$

The minimum # of nodes:



Let n_h be the minimum # of nodes of an AVL tree with height h

$$n_h = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ n_{h-1} + n_{h-2} + 1 & \text{if } h \geq 2 \end{cases}$$

Recall the Fibonacci numbers as follows.

$$F_h = \begin{cases} 0 & \text{if } h = 0 \\ 1 & \text{if } h = 1 \\ F_{h-1} + F_{h-2} & \text{if } h \geq 2 \end{cases}$$

Thus, we have $n_h > F_h, \forall\ h \geq 0$.

Since $F_h = \frac{\phi^h - \overline{\phi}^h}{\sqrt{5}}$, where $\phi = 1.61803$ and $\overline{\phi} = -0.61803$.

$$\begin{aligned} & n_h > F_h = \tfrac{\phi^h - \overline{\phi}^h}{\sqrt{5}} \simeq \tfrac{\phi^h}{\sqrt{5}} \quad \text{if } h \text{ is large} \\ \Longrightarrow\ & n \geq n_h > \tfrac{\phi^h}{\sqrt{5}} \\ \Longrightarrow\ & \log_\phi(\sqrt{5} \cdot n) > h \\ \Longrightarrow\ & h < \log_\phi \sqrt{5} + \log_\phi n \\ & = O(\log n) \end{aligned}$$

Therefore, the height of any AVL tree is $\Theta(\log n)$.

Chapter 22: Elementary Graph Algorithms

• Graph Representations:

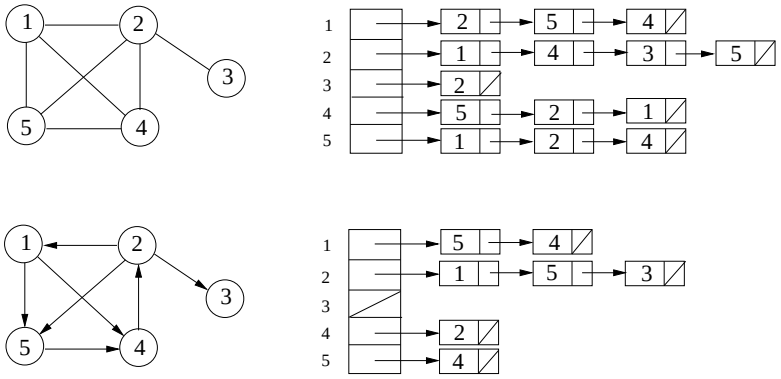
`Adjacency-list` representation for sparse graphs: $|E| \ll |V|^2$.

`Adjacency-matrix` representation for dense graphs: $|E|$ is close to $|V|^2$.

– **Adjacency-list:**

A graph $G =< V, E >$ can be represented by an array Adj of $|V|$ lists, one for each vertex in V, where V and E are the sets of vertices and edges, respectively. For each vertex $u \in V$, $Adj[u]$ is a list containing all the vertices v that $(u, v) \in E$.

Ex:



If G is an undirected graph, the sum of the lengths of all lists is equal to $2|E|$.

If G is a directed graph, the sum of the lengths of all lists is equal to $|E|$.

The amount of memory required for adjacency-list representation is $O(V + E)$ (both directed and undirected).

A weighted graph has a weight function $w : E \rightarrow Real$. Each $w(u, v)$ of the edge $(u, v) \in E$ is stored with vertex v in u's list.

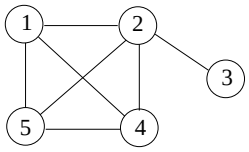
To determine if an edge (u, v) is in a given graph: $O(|E|)$. It's not efficient.

– **Adjacency-matrix:**

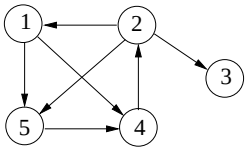
A graph $G =< V, E >$ can be represented by a $|V| \times |V|$ matrix $A = (a_{ij})$, where

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Ex:



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



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

If G is a directed graph, the number of 1's in the matrix is equal to $|E|$.

If G is an undirected graph, the number of 1's in the matrix is equal to $2|E|$.

Memory requirement is $|V|^2$.

The weighted graph in an adjacency-matrix representation: Store the weight $w(u,v)$ of the edge $(u,v) \in E$ in the entry a_{uv}. If $(u,v) \notin E$, then

$$a_{uv} = \begin{cases} \text{nil} \\ 0 \quad \text{depends on the applications} \\ \infty \end{cases}$$

To determine whether an edge (u,v) is in a given graph takes only constant time.

• Breadth-First search:

Given a graph $G =< V, E >$ and a source vertex s, breadth-first search discover every vertex that is reachable from s through the edges in E.

It can compute the shortest path (in terms of minimal # of edges) from s to all reachable vertices and produce a "breadth-first tree" with root s that contains all reachable vertices.

It works for both directed and undirected graphs.

Search strategy: The algorithm discovers all vertices at distance k from s before discovering any vertex at distance $k+1$.

```
BFS(G, s)
1.  for each vertex u in G(V)-s
2.      do color[u] <-- white
3.         p[u] <-- nil          // p[u]: parent of u
4.         d[u] <-- infinity     // d[u]: distance from s to u
5.  color[s] <-- gray
6.  p[s] <-- nil
7.  d[s] <-- 0
8.  Q <-- empty                  // Q is a queue. It will contain
                                 // all gray vertices and nothing else
9.  EnQueue(Q, s)
10. while Q is not empty
11.       do u <-- DeQueue(Q)
12.          for each v in Adj[u]
13.              do if color[v] = white
14.                    then color[v] = gray
15.                         p[v] <-- u
16.                         d[v] <-- d[u]+1
17.                         EnQueue(Q, v)
18.          color[u] <-- black
```

Three colors are used in the algorithm to help keeping track of the status of each vertex.

- **white:** not yet discovered.
- **gray:** discovered but not yet finished.
- **black:** finished.

A vertex is black (finished) if all its neighbors are discovered.

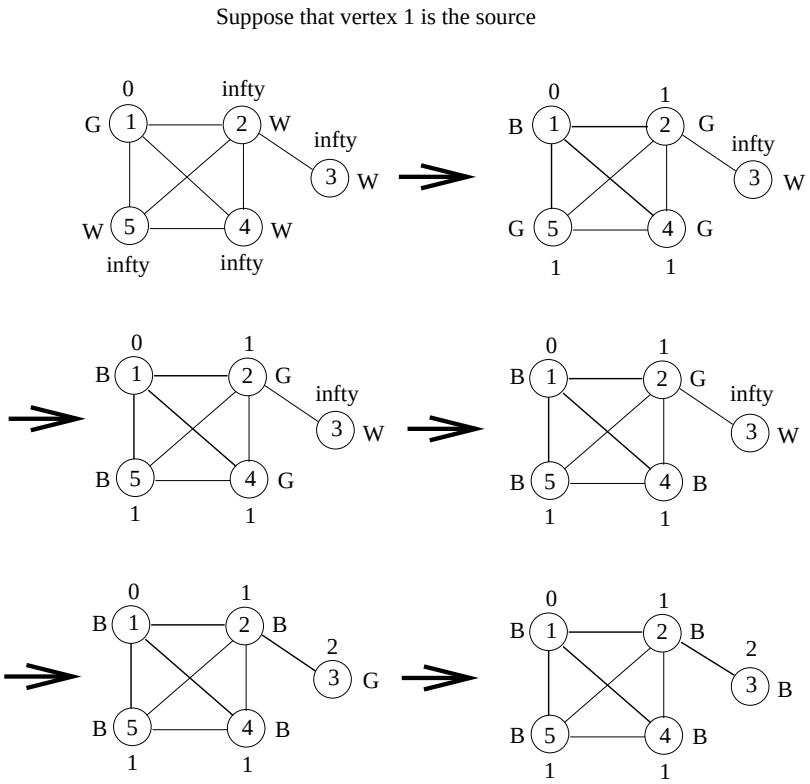
To generate a breadth-first tree, the edges used to discover white vertices will be added to the initial empty tree. That is, a white vertex v is discovered from a gray vertex u through an edge $(u, v) \in E$, we add v and (u, v) to the tree.

A vertex is discovered at most once (from white to gray), it has at most one parent.

Running time analysis:

- Initialization (line 1 to line 4): $O(V)$
- The adjacency-list of each vertex is examined at most once: $O(E)$
- Each vertex is EnQueued and DeQueued at most once: $O(V)$
- Totally, the running time is $O(V + E)$

Ex:



Question: Why the breadth-first tree is a shortest tree (in terms of # of edges) ?

Answer: because the breadth-first search discovers all vertices at the distance k from s before discovering any vertex at distance $k+1$ from s.

$\Longrightarrow$ Any vertex will be discovered as early as possible. That is, if there are multiple paths from s to a vertex u, u will be discovered through the shortest one.

- # Depth-First Search:

Given a graph $G =< V, E >$, depth-first search discovers all vertices in G to form a depth-first forest (a set of depth-first trees) because the search may be repeated from multiple sources.

Search strategy: The algorithm discovers vertices as deep as possible, if there is no deeper vertices to be discovered, then the search "backtrack" to the predecessor from the current vertex.

Three colors again to indicate the status of each vertex during search: white (not yet discovered), gray (discovered but not yet finished), and black (finished).

Two **timestamps** may be generated for each vertex during search to keep track of the relative ordering of events occurred to each vertex. Two events for each vertex: discover and finish.

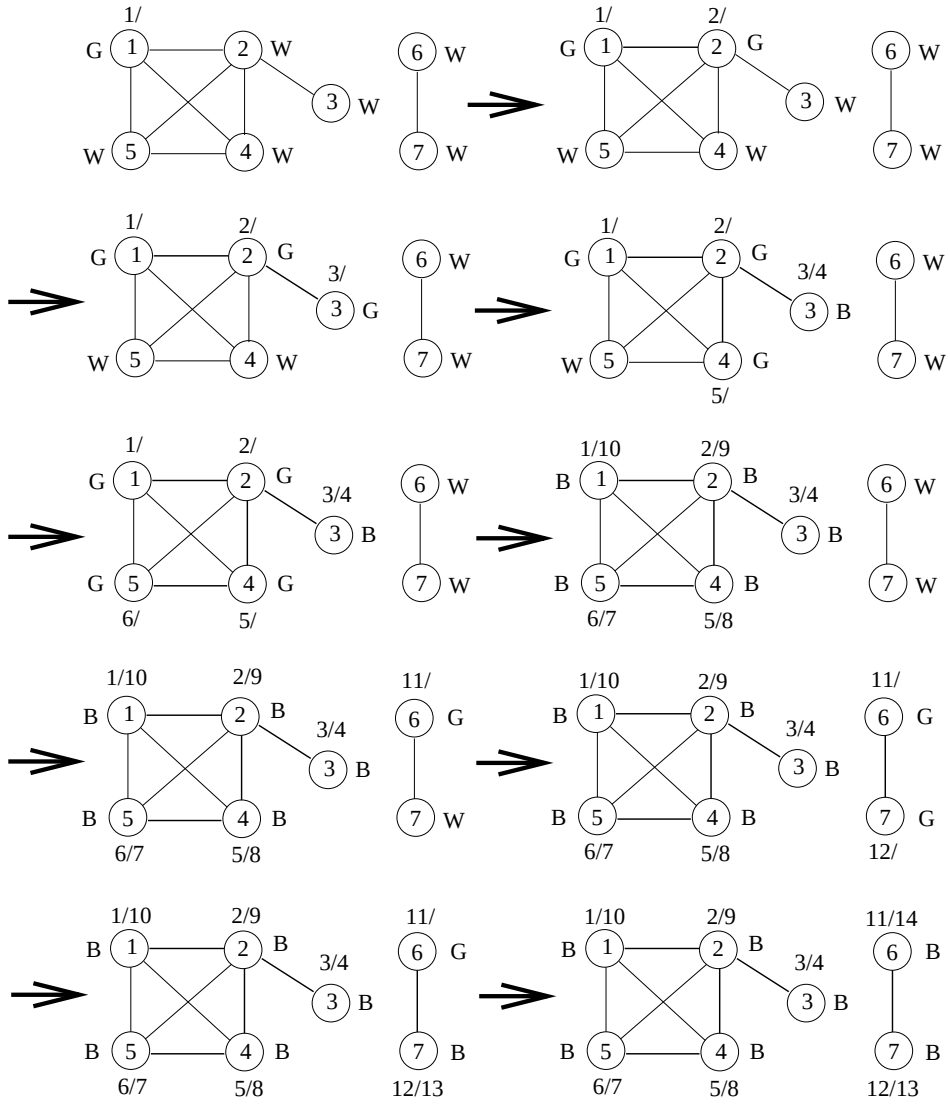
  - 1^{th} timestamp $d[u]$ for each vertex u records the time when u is discovered (from white to gray).
  - 2^{nd} timestamp $f[u]$ for each vertex u records the time when u is finished (from gray to black).
  - The time recorded is not a real time. A timer, with initial value 0, is incremented by 1 when an event occurred. The timestamps store the value of the timer when events occurred.

```
DFS(G)
1.  for each vertex u in V[G]
2.      do color[u] <-- white
3.         p[u] <-- nil          // p[u]: parent of u
4.  time <-- 0
5.  for each vertex u in V[G]
6.      do if color[u] = white
7.            then DFS-Visit(u)

DFS-Visit(u)
1.  color[u] <-- gray
2.  d[u] <-- ++time
3.  for each v in Adj[u]
4.      do if color[v] = white
5.            then p[v] <-- u
6.                 DFS-Visit(v)
7.  color[u] <-- black
8.  f[u] <-- ++time
```

Running time analysis:

- There are two `for` loops in DFS, each takes $\Theta(|V|)$ iterations.
- DFS-Visit is called exactly once for each vertex $v \in V$. Each DFS-Visit(v) examines all vertices in the list $Adj[v]$. Thus, total # of vertices examined is $\sum_{v \in G} Adj[v] = \Theta(|E|)$.
- The running time is $\Theta(|V| + |E|)$.

Ex:



The timestamps have the parenthesis structure:

- For each $d[u]$, we write down a left parenthesis and then u, i.e., "$(u$".
- For each $f[u]$, we write down u and then a right parenthesis, i.e., "$u)$".
- Doing the above mapping, in the order of timestamp values.

  For the above example, the structure is

$$(1\ (2\ (3\ 3)\ (4\ (5\ 5)\ 4)\ 2)\ 1)\ (6\ (7\ 7)\ 6)$$

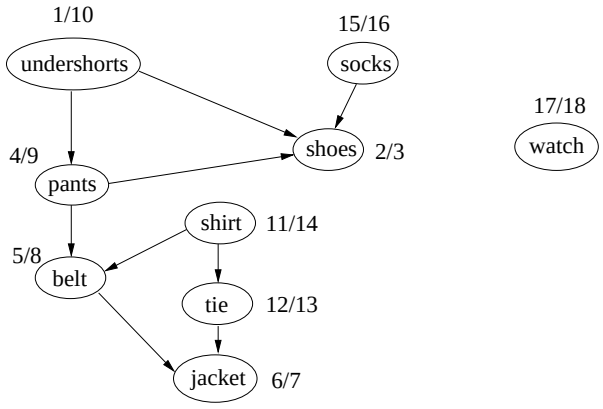
Vertex v is a proper descendant of vertex u in the depth-first forest if, and only if, $d[u] < d[v] < f[v] < f[u]$

- **Topological Sort:**

  An application for DFS search on a directed acyclic graph or "DAG".

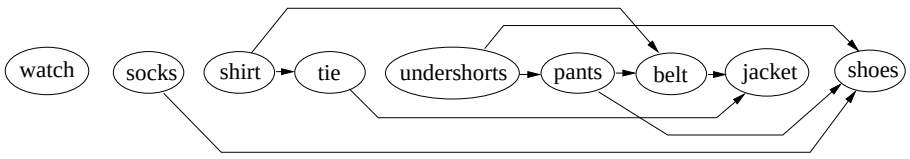
  A topological sort of a DAG $G =< V, E >$ is a linear ordering of all vertices in V, where, for all edges $(u, v) \in E$, u appears before v in the ordering.

  Ex: A DAG shows how a person can get dressed. The topological order suggests a way to get dressed.




```
Topological-Sort(G)
1. call DFS(G) to compute f[v] for each vertex v.
2. insert a node at the front of a linked list when the node is finished.
3. return the linked list
```

  For the above example, the topological sequence is



  For a given DAG, it may have many topological sequences. Each sequence corresponds to a different DFS search.

Chapter 23: Minimum Spanning Trees

Given a connected, undirected graph $G =< V, E >$, each edge $(u, v) \in E$ has a weight $w(u, v)$. The minimum spanning tree T is an acyclic subset of E that connects all vertices of V and whose weight $w(T) = \sum_{(u,v)\in V} w(u, v)$ is minimized.

Since T is acyclic and connects all vertices, it is a tree. We call it minimum spanning tree.

• Growing a Minimum Spanning Tree:

– The generic algorithm:

Manage a set A of edges that is always a subset of some minimum spanning tree. Initially $A =$. At each step, wee add an edge $(u, v) \in E$ into A, where (u, v) is **safe** to A.

An edge (u, v) is safe to A if $A \cup \{(u, v)\}$ is still a subset of some minimum spanning tree.

```
Generic-MST(G, w)
1.  A <-- {}
2.  while A does not form a minimum spanning tree
3.        do find an edge (u,v) that is safe to A
5.           A <-- A U {(u,v)}    // U: union
6.  return A
```

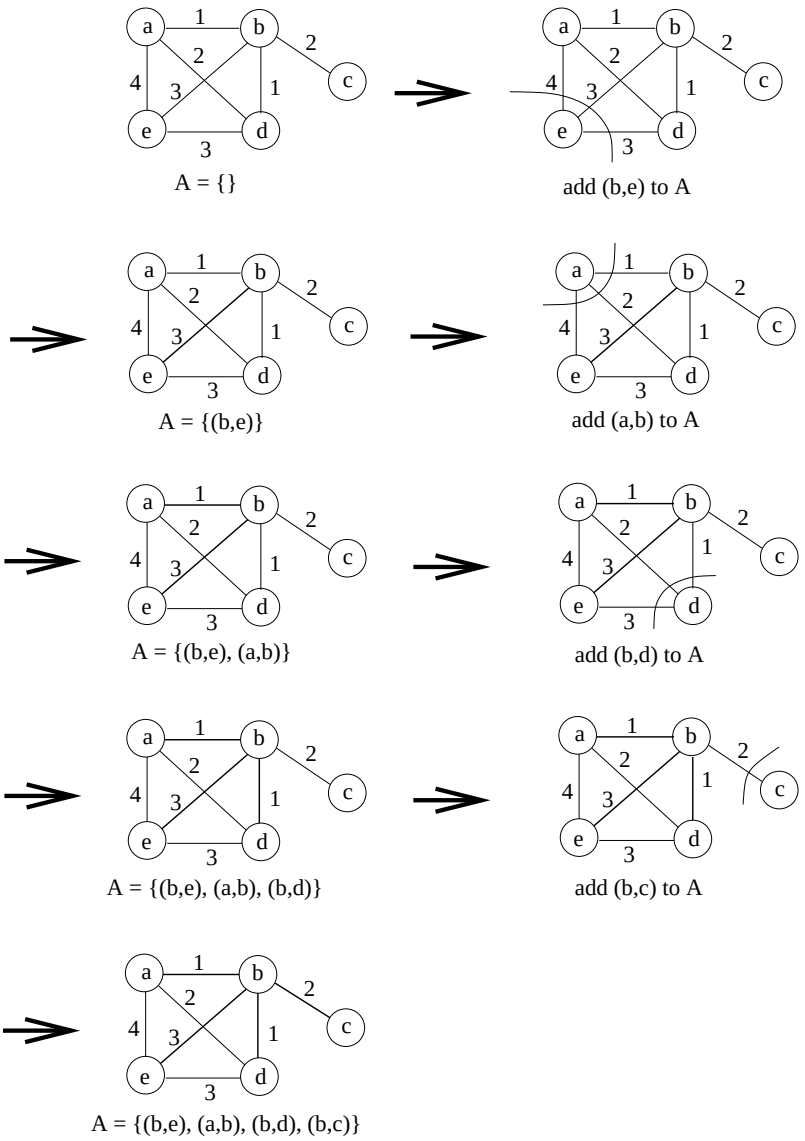
– What kind of edges are safe to A?

* Let a **cut** $(S, V - S)$ of an undirected graph $G =< V, E >$ is a partition of V.
* **An edge** $(u, v) \in E$ **crossing the cut** $(S, V - S)$ if one of its end points is in S and the other one is in $V - S$.
* **A cut respects a set** A **of edges** if no edge in A crosses the cut.
* An edge is a **light edge crossing a cut** if its weight is the minimum among all edges crossing the cut.

Theorem: *If A is a subset of E that is included in some minimum spanning tree for G. Let $(S, V-S)$ be any cut of G that respects A, and let (u,v) be a light edge crossing $(S, V-S)$. Then, the edge (u,v) is safe to A.*

We ignore the proof of the theorem. The above theorem suggests us a way to find a minimum spanning tree.

Ex:



It is easy for human with eye vision to find a sequence of cuts that respects A. But for a computer algorithm, it is not easy. Any minimum spanning tree algorithm tries to suggest a sequence of cuts respects the growing A.

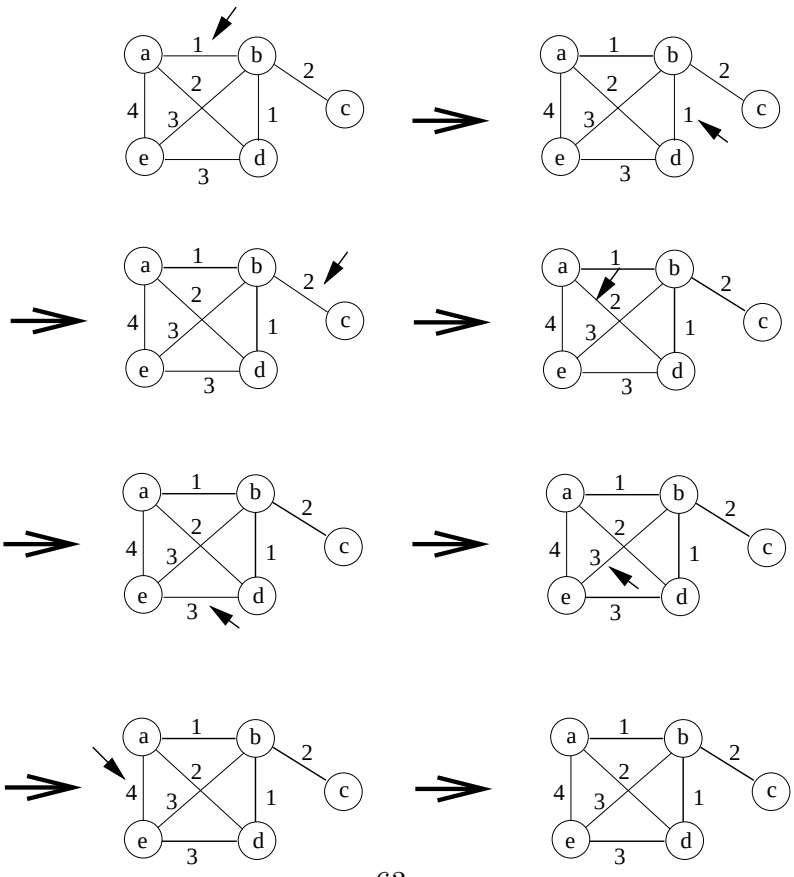
- **Kruskal's Algorithm:**

  The safe edge added to A at each step is always the least-weight edge in the graph that connects two distinct components.

  The algorithm is greedy because at each step it adds to A an edge with the least possible weight.

```
MST-Kruskal(G, w)
1.  A <-- {}
2.  for each vertex v in V[G]
3.      do Make-Set(v)
4.  sort the edges of E by non-decreasing weight w
5.  for each edge (u,v) in E, in order by non-decreasing weight
6.      do if Find-Set(u) != Find-Set(v)
7.             then A <-- A U {(u,v)}
8.                  Union(u,v)
9.  return A
```

Ex:



- **Prim's Algorithm:**

  The set A maintained is a growing tree.

  The safe edge added to A at each step is always the least-weight edge crossing the cut $(B, V - B)$, where B is the set of vertices connected by edges in A.

  The algorithm is greedy because the tree is augmented at each step with an edge that contributes the minimal amount of possible weight.

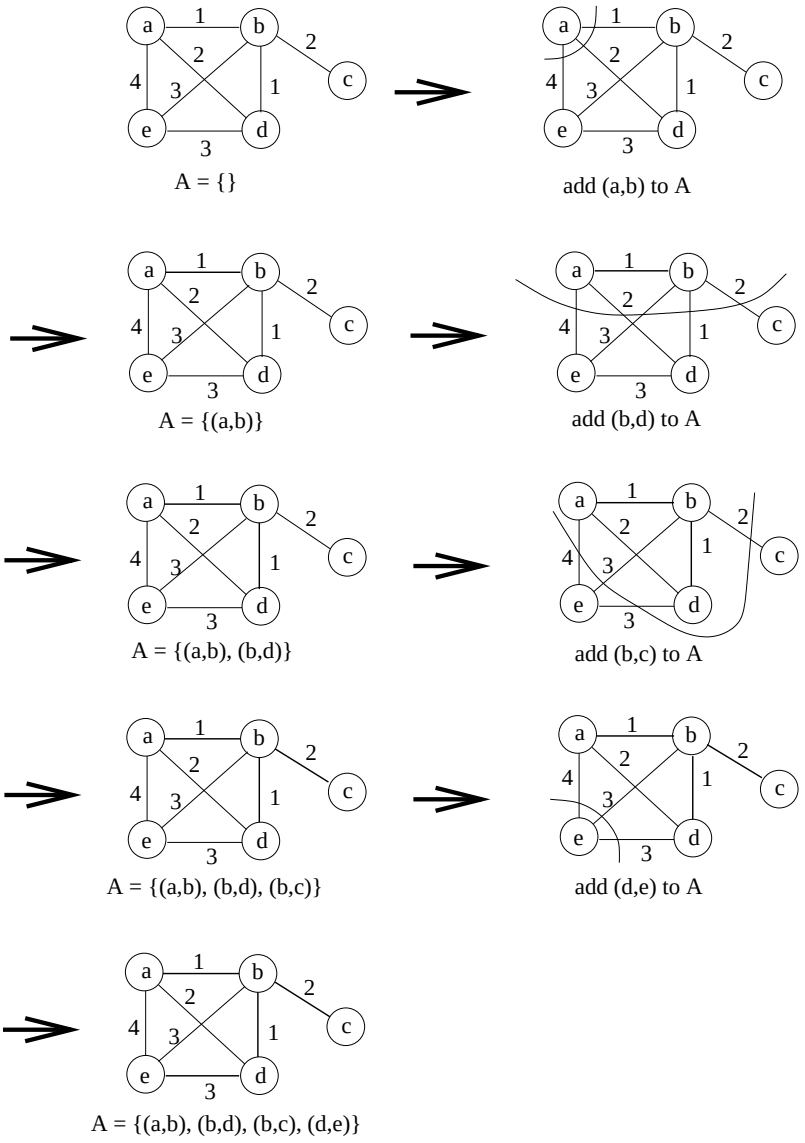
```
MST-Prim(G, w, r)  // r: source vertex
1.  Q <-- V[G]     // Q: priority queue
2.  for each u in Q
3.      do key[u] <-- infinity
4.  key[r] <-- 0
5.  p[r] <-- nil
6.  while Q != {}
7.        do u <-- Extract-Min(Q)
8.           for each v in Adj[u]
9.               do if v in Q and w(u,v) < key[v]
10.                    then p[v] <-- u
11.                         key[v] <-- w(u,v)
```

  Prim's algorithm uses a priority queue Q based on a key field.

  The queue Q maintains all vertices that are still not in the growing tree.

  The key field for each vertex v in Q, key$[v]$, is the weight of the light edge connecting v to the tree.

Ex:



Actually, the sequence of cuts suggested by Prim is the sequence of cuts that separate the growing tree from the rest of vertices at each step.