CPSC 221
Basic Algorithms and Data Structures

June 15, 2015

Administrative stuff

Schedule change:

Steve Wolfman here on Wednesday to discuss proof of program correctness using induction.

I probably won't be here on Thursday either.

Administrative stuff

Programming Assignment 2 is posted and is now due no later than 11:59 pm on Thursday, June 18. Please note the 24-hour extension.

Lab this week is just time with TAs to work on your last assignment, or to ask questions as you prepare for your final exam.

Administrative stuff

Midterm exams will be returned tomorrow and Wednesday in lab.

If you think that you've written a correct answer but you didn't get full marks for it, write a statement explaining why you think your answer is completely and totally correct and give it to the lab TAs, along with your exam. Do not leave the lab with your exam if you want an answer to be re-evaluated. Once you leave the lab with your exam, we will not re-evaluate any answers.

Administrative stuff

DO NOT return your exam for re-evaluation just because you think you deserve more partial credit for your incorrect answer. You will most likely lose more marks because your answer is already incorrect and on re-evaluation we'll probably decide that we were too generous with partial credit in the first place.

Administrative stuff

Your final exam is next week:
Wednesday, June 24 at 8:30am in ESB 1013.

You may bring 6 pages of notes, written on both sides of 8.5 x 11 inch paper. Do not bring 12 pages of notes, even if written only on one side.

Do not bring a calculator. If you have a calculator on your desk while writing the exam, we will refer you to the Faculty of Science for academic misconduct.

Counting

In CS, we often encounter situations where we want to count the number of possible outcomes of an event. For example, we may wish to determine: the number of possible paths to follow in a directed network (graph), the number of possible 5-8 character passwords, etc.

Suppose that the customers of a bank are asked to use 3-digit PINs to protect their accounts when using the bank's ATM machines. If the other constraints are: no 2 digits can be the same, and the only allowable digits are 1, 2, 3, & 4, how many PINs are possible?

(a) Describe the problem in mathematical notation.

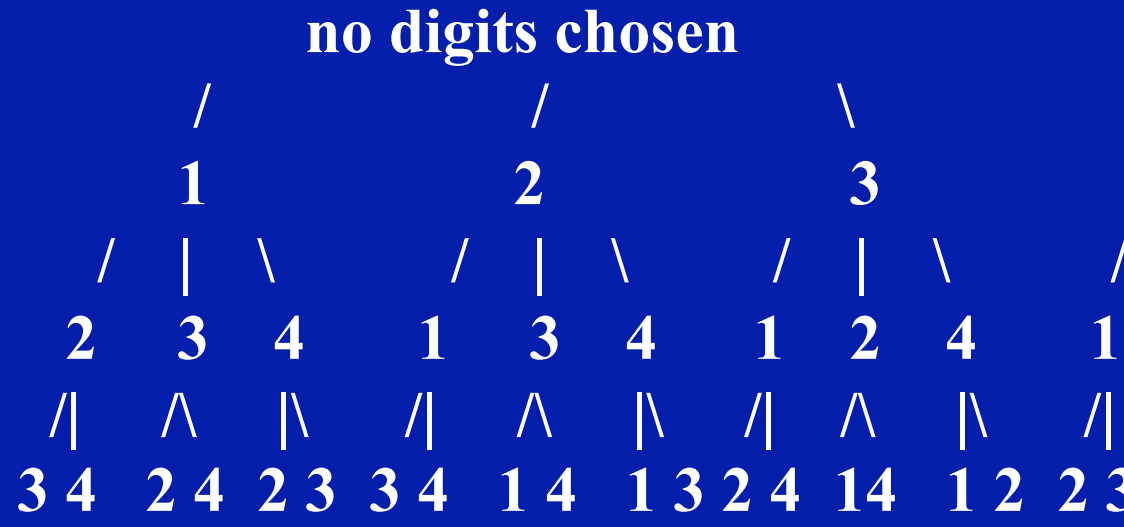
Suppose that the customers of a bank are asked to use 3-digit PINs to protect their accounts when using the bank's ATM machines. If the other constraints are: no 2 digits can be the same, and the only allowable digits are 1, 2, 3, & 4, how many PINs are possible?

(a) Describe the problem in mathematical notation.

want x_1, x_2, x_3 where $x_1, x_2, x_3 \in \{1,2,3,4\}$ and $\forall i \; \forall j$ if $i \neq j$ then $x_i \neq x_j$, $i,j \in \{1,2,3\}$

(b) Model the problem using a tree diagram.

(b) Model the problem using a tree diagram.



Nodes in the lowest level = (2+2+2) * 4 = 24
Or 4 * 3 * 2 = 24

Multiplication Principle (Product Rule)

If an operation consists of t steps and:

Step 1 can be performed in n_1 ways,

Step 2 can be performed in n_2 ways,

…

Step t can be performed in n_t ways, then the entire operation can be performed in $n_1 n_2 \ldots n_t$ different ways.

Example: How many postal codes begin with the letter V and end with the digit 4?

— — — — — —

Multiplication Principle (Product Rule)

If an operation consists of t steps and:

Step 1 can be performed in n_1 ways,

Step 2 can be performed in n_2 ways,

…

Step t can be performed in n_t ways, then the entire operation can be performed in $n_1 n_2 \ldots n_t$ different ways.

Example: How many postal codes begin with the letter V and end with the digit 4?

V					4	
1	10	26	10	26	1	$10^2(26^2)$

Multiplication Principle (Product Rule)

A variation on the multiplication rule:

The size of a union of m disjoint (i.e., no elements in common) sets, each of size n, is m x n.

Example:

Use the multiplication principle to prove that the number of subsets of a set containing n elements is 2^n.

Example:

Use the multiplication principle to prove that the number of subsets of a set containing n elements is 2^n.

Given each element in the set, for any given subset there are two choices for that element-- in the subset or out. If we consider all possible elements, that is 2 choices for every element, or 2 * 2 * 2...

Example: Suppose a programming language requires you to define variable names (identifiers) using exactly 3 ***different*** upper case characters. How many different identifiers contain either an A or a B, but not both?

Example: Suppose a programming language requires you to define variable names (identifiers) using exactly 3 ***different*** upper case characters. How many different identifiers contain either an A or a B, but not both?

We have three characters. One of them must be A or B. Which element is an A or B has 3 choices. Then we must decide which it is, A or B (2 choices). Finally for the remaining two characters we have 24 choices for the first (since it cannot be an A or B now), and 23 for the second, since it cannot be an A or a B and cannot be the same as the other element we just chose.

So, 2 * 3 * 24 * 23

Addition Principle (Sum Rule)

If $X_1, X_2, \ldots, X_t$ are pairwise disjoint sets (i.e., $X_i \cap X_j = \varnothing$ $\forall i \; \forall j$ s.t. $i, j \in \{1, 2, \ldots, t\}$, $i \neq j$), then the number of ways to select an element from any of X_1 or X_2 or … or X_t is:

$$| X_1 | + | X_2 | + \ldots + | X_t |$$

disjoint means they have no elements in common

So in English, this means that the size of a union of a family of mutually disjoint sets is the sum of the sizes of the sets

We're also using the sum rule when we're counting executions in code:

```
for i = 1 to n – 1
   for j = i + 1 to n
      if (A[i] > A[j])
         swap (A[i],A[j])
```

How many time is the comparison A[i] > a[j] made?

(n-1) + (n-2) + ... + 1

That's the addition principle in action

When we can write set S as a union of disjoint sets S_1, S_2, ..., S_k, we say that we have partitioned S into blocks S_1, S_2, ..., S_k, and that those blocks form a partition of S.

So we can partition {1,2,3,4,5} into {1},{3,5},{2,4}, for example.

More addition principle:

If a finite set S has been partitioned into blocks, then the size of S is the sum of the sizes of the blocks.

Example: Suppose a user is asked to create an **alphanumeric** password consisting of at least 6 characters, but no more than 8 characters. How many different passwords are possible?

Example: Suppose a user is asked to create an **alphanumeric** password consisting of at least 6 characters, but no more than 8 characters. How many different passwords are possible?

We can partition the set of possible choices into three groups that are disjoint from one another: 6-char passwords, 7-char passwords and 8-char passwords (assume upper- and lower-case letters and 10 digits)

Total choices for passwords is then $62^6 + 62^7 + 62^8$

Back-of-the-envelope calculation. For the previous example, roughly how long would it take for a program to perform a **brute-force attack** to discover a user's password, if we assume (guess?) that a million passwords can be tested each second?

Best case?

Average case?

Worst case?

Side thought: common user password habits make a shambles of security.

Back-of-the-envelope calculation. For the previous example, roughly how long would it take for a program to perform a **brute-force attack** to discover a user's password, if we assume (guess?) that a million passwords can be tested each second?

2.2×10^{14} passwords/(10^6 passwords/sec)

2.2×10^8 seconds

~30 million seconds per year

approximately 7 years

Best case? 1

Average case? $0.5 \times 2.2 \times 10^{14}$

Worst case? 2.2×10^{14}

Side thought: common user password habits make a shambles of security.

Example: Suppose a user is asked to create an alphanumeric password consisting of at least 6 characters, but no more than 8 characters, but this time, *at least* 1 of the chars. has to be a number.

Does this feel more secure?

Example: Suppose a user is asked to create an alphanumeric password consisting of at least 6 characters, but no more than 8 characters, but this time, *at least* 1 of the chars. has to be a number.

How many possible combinations are there this time?

Example: Suppose a user is asked to create an alphanumeric password consisting of at least 6 characters, but no more than 8 characters, but this time, *at least* 1 of the chars. has to be a number.

How many possible combinations are there this time?

 # 6-char passwords: $62^6 - 52^6$ (# of invalid)
 # 7-char passwords: $62^7 - 52^7$ (# of invalid)
 # 8-char passwords: $62^8 - 52^8$ (# of invalid)

Example: Suppose a user is asked to create an alphanumeric password consisting of at least 6 characters, but no more than 8 characters, but this time, *at least* 1 of the chars. has to be a number.

How many possible combinations are there if we *know* users are “minimally complex” in their password choices?

Example: Suppose a user is asked to create an alphanumeric password consisting of at least 6 characters, but no more than 8 characters, but this time, *at least* 1 of the chars. has to be a number.

How many possible combinations are there if we *know* users are "minimally complex" in their password choices?

if the number is the last character...
 # 6-char passwords: $52^5 \times 10$
 # 7-char passwords: $52^6 \times 10$
 # 8-char passwords: $52^7 \times 10$

Example: Suppose a user is asked to create an alphanumeric password consisting of at least 6 characters, but no more than 8 characters, but this time, *at least* 1 of the chars. has to be a number.

Now, will the passwords be *more* secure or *less* secure?

Inclusion-Exclusion Principle

If an object can be found in 2 or more sets at the same time, then we cannot use the addition principle. Why not?

If A and B are sets, then the total number of elements in *either* A or B is given by:

$$|A \cup B| = |A| + |B| - |A \cap B|$$

Example using a Venn Diagram:

Inclusion-Exclusion Principle

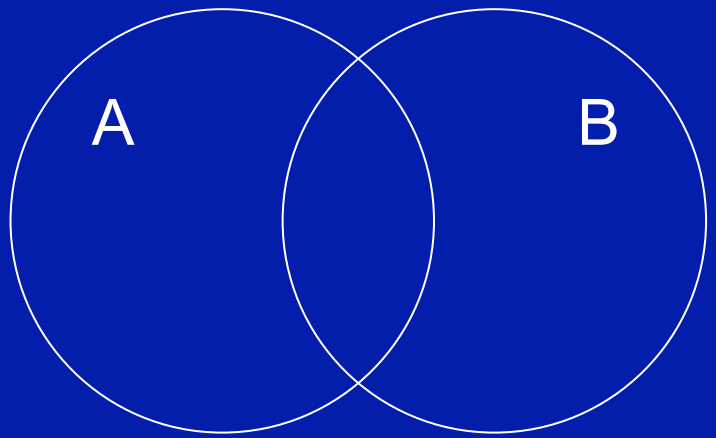
If an object can be found in 2 or more sets at the same time, then we cannot use the addition principle. Why not?

We could over count

If A and B are sets, then the total number of elements in *either* A or B is given by:

$$|A \cup B| = |A| + |B| - |A \cap B|$$

Example using a Venn Diagram:



Example: How many 8-bit strings either start with “1” or end with “00”? (assume one or the other but not both)

Example: How many 8-bit strings either start with "1" or end with "00"? (assume one or the other but not both)

There are 2^7 choices for strings that start with a 1.

There are 2^6 choices for strings that end with a 00.

...but these groups both include strings that start with a 1 AND end with a 00.

Since there are 2^5 choices for strings that start with a 1 and end with a 00...

...we remove the extra counted strings:

$2^7 + 2^6 - 2^5 - 2^5$ Why subtract 2^5 twice?

Permutations & Combinations

How many different outcomes are there if you choose *r* balls from an urn containing *n* different balls?

Example: $r = 3$ Urn = {**A,B,C,D**}	Order matters **ABC ≠ CAB**	Order doesn't **ABC = CAB**
Repetition *not* OK **e.g., AAB** is *not* counted	**ABC, ABD, ACB, ACD, ADB, ADC, BAC, BAD, BCA, BCD, BDA, BDC, CAB, CAD, CBA, CBD, CDA, CDB, DAB, DAC, DBA, DBC, DCA, DCB** *r*-permutations	**ABC, ABD, ACD, BCD** *r*-combinations
Repetition OK (ball is thrown back in urn after being picked) e.g., **AAB** is counted	**AAA, AAB, AAC, AAD, ABA, ABB, ABC, ABD, ...** *r*-permutations w/ repetitions	**AAA, AAB, AAC, AAD, ABB, ABC, ABD, ACC, ACD, ADD, BBB, BBC, BBD, BCC, BCD, BDD, CCC, CCD, CDD, DDD** *r*-combinations w/ repetitions

(1) *r*-Permutations
(Order matters, and repetition is not allowed.)

An *r*-permutation of *n* distinct elements $x_1, x_2, \ldots, x_n$ is an ordering of an *r*-element subset of $\{x_1, x_2, \ldots, x_n\}$. The number of *r*-permutations is:

$$P(n, r) = n^{(r)} = n! / (n - r)!$$

Derivation:

(1) *r*-Permutations
(Order matters, and repetition is not allowed.)

An *r*-permutation of *n* distinct elements $x_1, x_2, \ldots, x_n$ is an ordering of an *r*-element subset of $\{x_1, x_2, \ldots, x_n\}$. The number of *r*-permutations is:

$$P(n, r) = n^{(r)} = n! / (n - r)!$$

Derivation:

We need to divide out the effect of the permutations of the elements that are *not* in our chosen set.

n(n-1)(n-2)...(n-r+1)(n-r)(n-r-1)...(1) / (n-r)(n-r-1)...(1)

which simplifies (divide out common elements):

n(n-1)(n-2)...(n-r+1)

Example 1: In how many ways can 5 electoral candidates finish in an election, assuming no ties?

Example 2: In how many ways can 7 girls and 3 boys line up, if the boys must stand next to each other?

Example 1: In how many ways can 5 electoral candidates finish in an election, assuming no ties?

$$5 * 4 * 3 * 2 * 1$$

Example 2: In how many ways can 7 girls and 3 boys line up, if the boys must stand next to each other?

Example 1: In how many ways can 5 electoral candidates finish in an election, assuming no ties?

5 * 4 * 3 * 2 * 1

Example 2: In how many ways can 7 girls and 3 boys line up, if the boys must stand next to each other?

Choose the boys to be one grouping. There are 3! choices for how the boys may stand. Then there are 8! choices for how the individual girls plus the grouping of boys may be placed. So, 3!8!

E.g. G BBB G G G G G G

Example 3: Suppose an operating system has a queue of 3 low priority and 5 high priority processes ready to run. In how many ways could these processes be ordered for execution if 2 low priority processes are not allowed to be executed back to back?

Example 3: Suppose an operating system has a queue of 3 low priority and 5 high priority processes ready to run. In how many ways could these processes be ordered for execution if 2 low priority processes are not allowed to be executed back to back?

Since there are 5 high priority jobs, there are 5! choices.

We have to place the low priority processes with at least one HP between them:

_ H1 _ H2 _ H3 _ H4 _ H5 _ (_ indicates eligible spot for one LP process)

So we have 6 possible locations and have to choose 3 of them, which is 6 * 5 * 4

Answer: 5! * 6 * 5 * 4

(2) *r*-Combinations (Order doesn't matter, and repetition is *not* allowed.)

An *r*-combination of *n* distinct elements $x_1, x_2, \ldots, x_n$ is an *r*-element subset of $\{x_1, x_2, \ldots, x_n\}$. The number of *r*-combinations is:

$$C(n, r) = \binom{n}{r} = n! / (n - r)!\, r!$$

Example 4: A donut shop has 10 kinds of donuts. In how many ways can 6 distinct kinds of donuts be selected?

Example 5: Show how to derive a relationship between *r*-permutations and *r*-combinations.

(2) *r*-Combinations (Order doesn't matter, and repetition is *not* allowed.)

An *r*-combination of *n* distinct elements $x_1, x_2, \ldots, x_n$ is an *r*-element subset of $\{x_1, x_2, \ldots, x_n\}$. The number of *r*-combinations is:

$$C(n, r) = \binom{n}{r} = n! / (n - r)!\, r!$$

Example 4: A donut shop has 10 kinds of donuts. In how many ways can 6 distinct kinds of donuts be selected?

10 choose 6

Example 5: Show how to derive a relationship between *r*-permutations and *r*-combinations.

(2) *r*-Combinations (Order doesn't matter, and repetition is *not* allowed.)

An *r*-combination of *n* distinct elements $x_1, x_2, \ldots, x_n$ is an *r*-element subset of $\{x_1, x_2, \ldots, x_n\}$. The number of *r*-combinations is:

$$C(n, r) = \binom{n}{r} = n! / (n - r)!\, r!$$

Example 4: A donut shop has 10 kinds of donuts. In how many ways can 6 distinct kinds of donuts be selected?

10 choose 6

Example 5: Show how to derive a relationship between *r*-permutations and *r*-combinations.

there are r! ways to reorder an r-combination. So, nCr is nPr/r!, which is correct!

(3) *r*-Permutations with Repetition
(Generalized *r*-Permutations)

Here, *order matters*, and *repetition is allowed*.

Suppose we have a set of n distinct elements $x_1, x_2, \ldots, x_n$ and we select a sequence of r elements, allowing repeats. How many different sequences are possible?

(3) *r*-Permutations with Repetition
(Generalized *r*-Permutations)

Here, *order matters*, and *repetition is allowed*.

Suppose we have a set of n distinct elements $x_1, x_2, \ldots, x_n$ and we select a sequence of r elements, allowing repeats. How many different sequences are possible?

$$\mathbf{n^r}$$

(4) *r*-Combinations with Repetition (Generalized r-Combinations)

Here, order *doesn't* matter, and *repetition is allowed*.

Suppose we have a set of n distinct elements $x_1, x_2, \ldots, x_n$. The number of unordered r-element selections from this set, with repetition allowed, is:

(4) *r*-Combinations with Repetition (Generalized r-Combinations)

Here, order *doesn't* matter, and *repetition is allowed*.

Suppose we have a set of *n* distinct elements $x_1, x_2, \ldots, x_n$. The number of unordered *r*-element selections from this set, with repetition allowed, is:

(r + n - 1) choose (n-1)

or (equivalently)

(r + n - 1) choose r

Example 8: If a donut shop sells 3 kinds of donuts: plain, glazed, and jelly, then how many possible selections of 7 donuts are there?

Example 8: If a donut shop sells 3 kinds of donuts: plain, glazed, and jelly, then how many possible selections of 7 donuts are there?

Can think of this as a partitioning problem:

O O O | O O| O O <--- two partitions

..each group represents a kind of donut

2 dividers and 9 positions (possible) for the dividers

Summary of Formulas

Order Matters & Repetition is *not* Allowed	Order does *not* Matter & Repetition is *not* Allowed
$n^{(r)}$	$\begin{bmatrix} n \\ r \end{bmatrix}$
Order Matters & Repetition is Allowed	Order does *not* Matter & Repetition is Allowed
n^r	$\begin{bmatrix} r + n - 1 \\ n - 1 \end{bmatrix}$

Note: the square brackets are really supposed to be parentheses. PowerPoint can be frustrating at times.

Permutations of Indistinguishable Objects

Multinomial Theorem: The number of different permutations of n objects where there are:

n_1 indistinguishable type 1 objects

n_2 indistinguishable type 2 objects

…

n_k indistinguishable type k objects

and $n_1 + n_2 + \ldots + n_k = n$ is:

Proof:

Permutations of Indistinguishable Objects

Multinomial Theorem: The number of different permutations of n objects where there are:

n_1 indistinguishable type 1 objects

n_2 indistinguishable type 2 objects

…

n_k indistinguishable type k objects

and $n_1 + n_2 + \ldots + n_k = n$ is:

Proof:

by mult. principle:

$$(n \text{ choose } n_1)(n - n_1 \text{ choose } n_2)\ldots (n-n_1-n_2-\ldots-n_{k-1}) \text{ choose } n_k$$

$$n! / n!(n-n_1)! * (n-n_1)! / n!(n-n_1-n_2)! * (n-n_1-n_2)! / n!(n-n_2)! * \ldots * (n-n_1-n_2 - \ldots - n_{k-1})! / n_k!\, 0!$$

Example 9: How many strings can be made by reordering the letters PEPPER?

Example 9: How many strings can be made by reordering the letters PEPPER?

Since the duplicate letters are indistinguishable:

We have 6 choose 3 choices for P, 3 choose 2 choices E, 1 choose 1 choices R, so:

(6 choose 3)(3 choose 2)

Theorem: The number of ways to place n distinguishable objects into r distinguishable boxes so that n_i objects are placed into box i (where $i = 1, 2, \ldots, r$ and $n_1 + n_2 + \ldots + n_r = n$) is:

Example 10: How many ways are there to distribute hands of 5 cards to each of 4 players from a deck of 52 cards?

Theorem: The number of ways to place n distinguishable objects into r distinguishable boxes so that n_i objects are placed into box i (where $i = 1, 2, \ldots, r$ and $n_1 + n_2 + \ldots + n_r = n$) is:

$$\mathbf{n! / n_1! \, n_2! \, n_3! \ldots n_r!}$$

Example 10: How many ways are there to distribute hands of 5 cards to each of 4 players from a deck of 52 cards?

Theorem: The number of ways to place n distinguishable objects into r distinguishable boxes so that n_i objects are placed into box i (where $i = 1, 2, \ldots, r$ and $n_1 + n_2 + \ldots + n_r = n$) is:

$$\mathbf{n! / n_1! \, n_2! \, n_3! \ldots n_r!}$$

Example 10: How many ways are there to distribute hands of 5 cards to each of 4 players from a deck of 52 cards?

n = 52
r = 5

52! / (5! 5! 5! 5! (52-20)!)

We don't care about distinguishing between the cards once they're in a hand (just like the duplicate letters in "PEPPER")

Home Exercise #1: How many non-negative integer solutions are there to the equation: $x_1 + x_2 + x_3 + x_4 = 10$?

Home Exercise #2: How would your answer to Home Exercise #1 change if we had these constraints on the number of x_i's:

$x_1 \geq 1$, $x_2 \geq 2$, $x_3 \geq 0$, and $x_4 \geq 0$?

(Hint: what if $y_1 = x_1 - 1$? What constraint is there on y_1?)

Home Exercise #3: How many times is the following print statement executed, if $n = 4$?

```
for i1 = 1 to n
    for i2 = 1 to i1
        for i3 = 1 to i2
            print  i1, i2, i3
```

Home Exercise #1: How many non-negative integer solutions are there to the equation: $x_1 + x_2 + x_3 + x_4 = 10$?

Think of this as 10 "things" to distribute: | | | | | | | | | |

We can then distribute those things over x_1, x_2, x_3, x_4

r = 10
n = 4

E.g. pulling x_1, x_2, x_1, x_2, x_1, x_2, x_3, x_1, x_3, x_4, out of the barrel would give us 4 + 3 + 2 + 1

(r + n - 1) choose r (order no, rep OK!)

(10 + 4 - 1) choose 10

Home Exercise #2: How would your answer to Home Exercise #1 change if we had these constraints on the number of x_i's:

$x_1 \geq 1$, $x_2 \geq 2$, $x_3 \geq 0$, and $x_4 \geq 0$?

(Hint: what if $y_1 = x_1 - 1$? What constraint is there on y_1?)

Now we have to assign at least 1 thing to x_1, 2 things x_1 to 2, so let's do that first. That leaves us only 7 things to place:

r = 7
n = 4

E.g. pulling x_1, x_2, x_2, x_3, x_1, x_3, x_4, out of the barrel would give us 3 (the two x1s we drew, plus the one we put in at the beginning) + 4 (the two x2s we drew, plus the two things we put in at the beginning + 2 + 1

(r + n - 1) choose r (order no, rep OK!)

(7+ 4 - 1) choose 7

Home Exercise #3: How many times is the following print statement executed, if $n = 4$?

```
for i1 = 1 to n
    for i2 = 1 to i1
        for i3 = 1 to i2
            print  i1, i2, i3
```

Looking for a triple that satisfies 1 <= i3 <= i2 <= i1 <= n

r = 3 (choose three possible values)
n = 4 (from four possibilities)

(r + n - 1) choose r (order no, rep OK!)

(3 + 4 - 1) choose 3

Questions?

Time for practice?

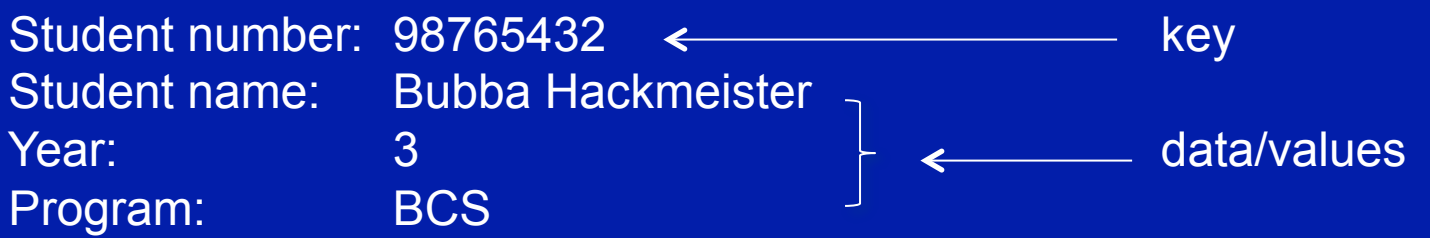
No, time for quick review. I'll post the in-class worksheet for this lecture, and answers, on Connect.

Tree terminology

To keep things simple, we have been talking about binary trees as if a node contains only a value and two pointers to its subtrees.

In reality, a node will have a key (usually unique) to identify the associated value(s) or data contained in the node (and, of course, the pointers to the subtrees). It is the key that is used in searching. This may come up in future examples.

For example:



Tree terminology

The path from node N_1 to node N_k is a sequence of nodes N_1, N_2, ..., N_k where N_i is the parent of N_{i+1}.

The length of the path is the number of edges in the path.

The depth or level of a node N is the length of the path from the root to N. The depth of the root is 0.

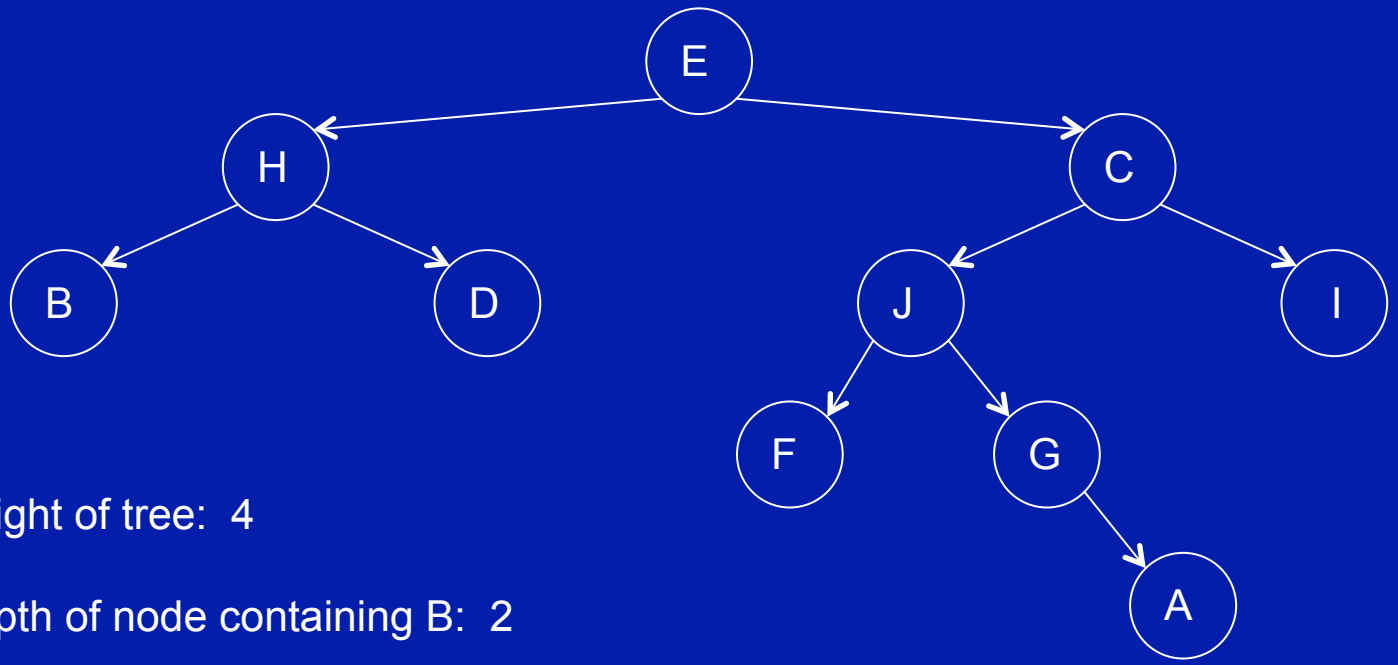
The height of a node N is the length of the longest path from N to a leaf node (a node with no child). The height of a leaf node is 0.

Tree terminology

The height of a tree is the height of its root node. The height of the empty tree is -1. The height of a tree with only a single node is 0.

The number of nodes in a binary tree of height h is at least h + 1 and no more than $2^{h+1} - 1$.

Tree terminology



Height of tree: 4

Depth of node containing B: 2

Height of node containing B: 0

# of nodes in this tree: 10

# of leaves (i.e., terminal or external nodes): 5

# of non-leaf (i.e., internal) nodes: 5

Tree terminology

A full binary tree is a binary tree in which each node has exactly 0 or 2 children. Each internal node has exactly 2 children, and each leaf has 0 children.

A complete binary tree is a binary tree of height h in which leaves appear only at two adjacent levels, depth h – 1 and depth h, and the leaves at the very bottom (depth h) are in the leftmost positions.

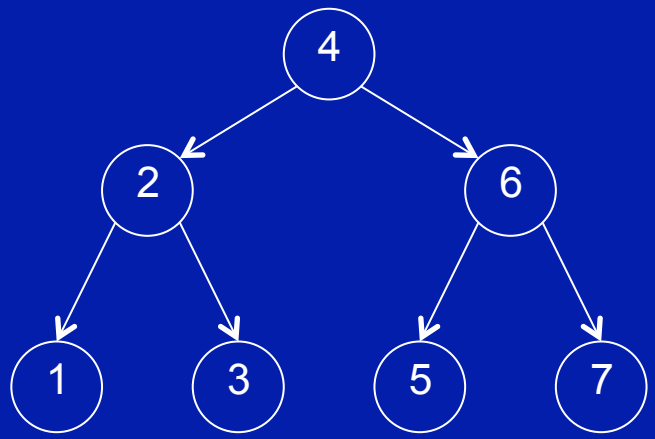
Attributes of a “good” BST

Search in a binary search tree is fast (O(lg n)) if the tree is shallow. (e.g., complete binary search tree, or one which isn’t necessarily complete but the “fringe” is only at the bottom level)

But search slows down when one branch is much longer than the other.

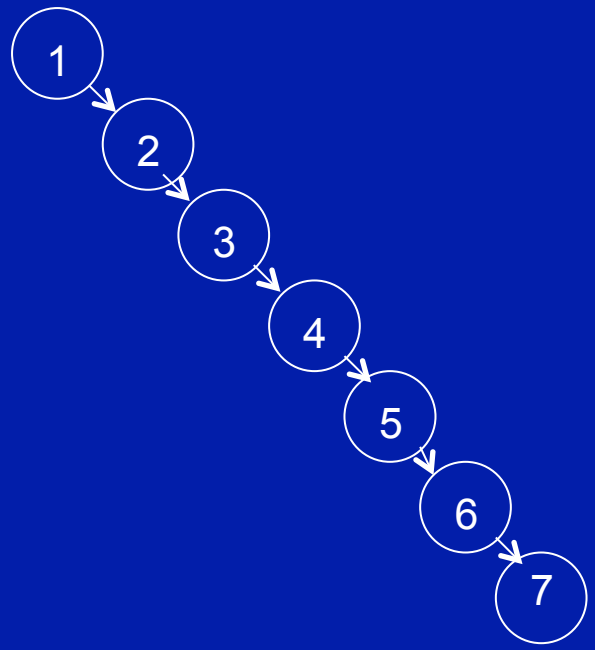
Attributes of a “good” BST

This is good:



Attributes of a “good” BST

This? Not so good:



Balance

Balance = height(right subtree) – height(left subtree)*

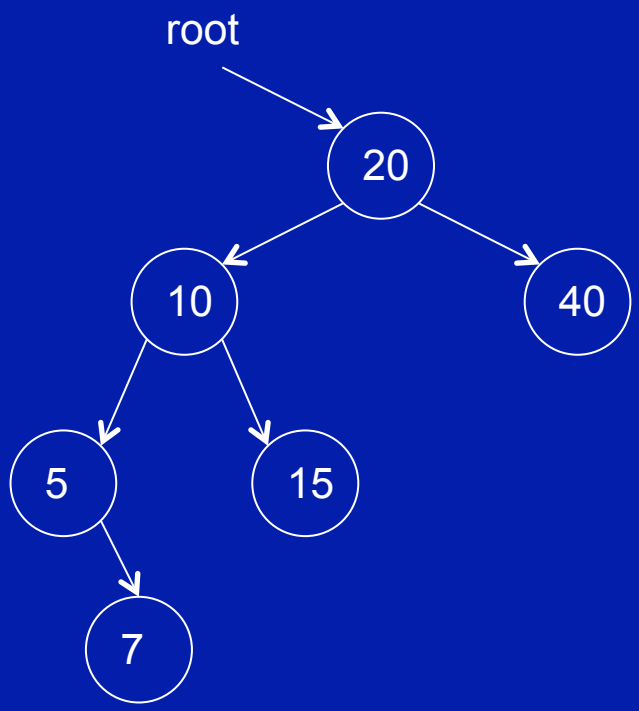
If that difference is zero everywhere, our tree is perfectly balanced.

If that difference is small everywhere, then the tree is balanced enough.

If the balance is between -1 and 1 everywhere in the tree, then the maximum height of the tree is 1.44 lg n. So we don't need a perfectly balanced BST to maintain O(lg n) time complexity for search.

* To maintain consistency with your textbook. Others may say that balance = height(left subtree) – height(right subtree). It works either way.

Balance

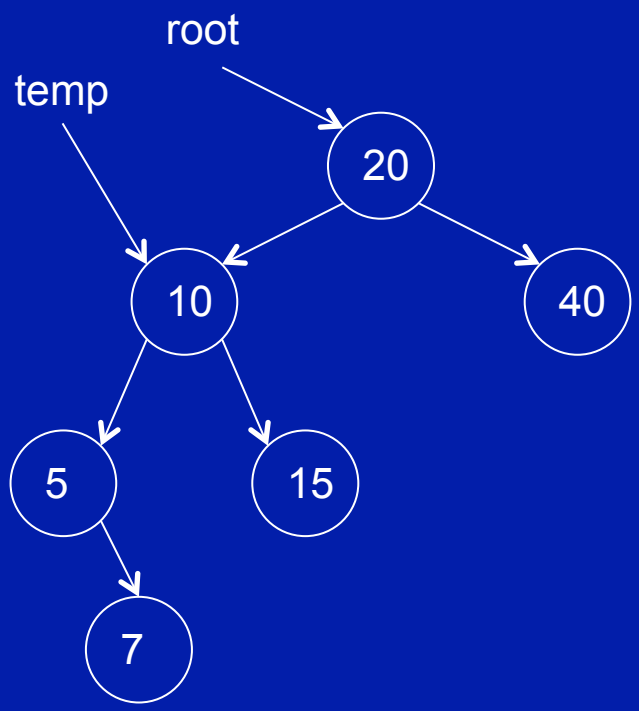


It's all just pointer manipulation.

Here's the algorithm for right rotation from your textbook:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Balance

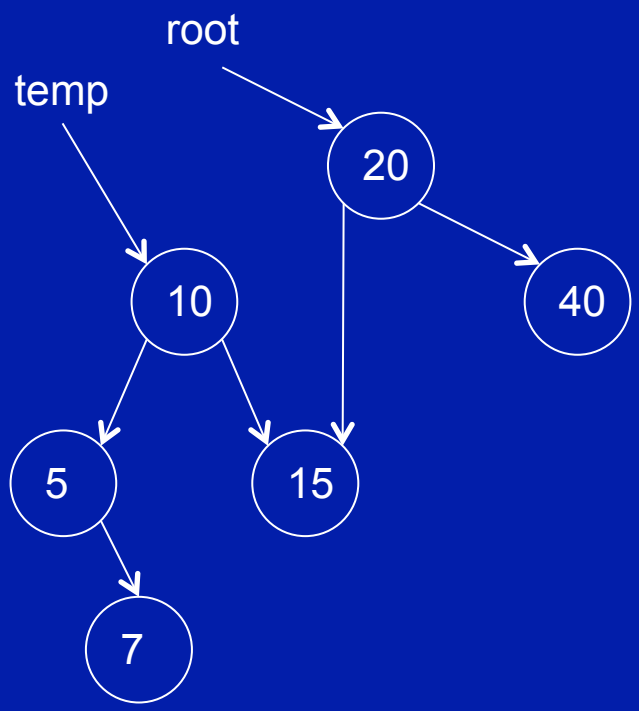


It's all just pointer manipulation.

Here's the algorithm for right rotation from your textbook:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Balance

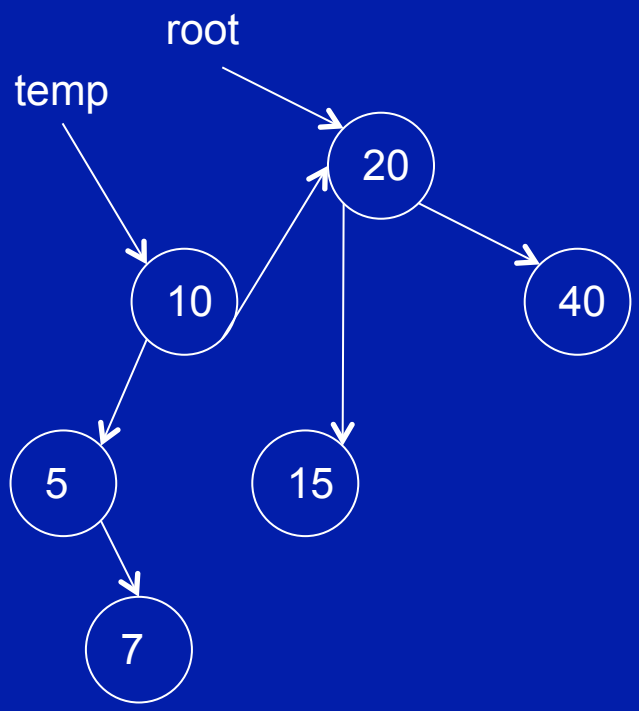


It's all just pointer manipulation.

Here's the algorithm for right rotation from your textbook:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Balance

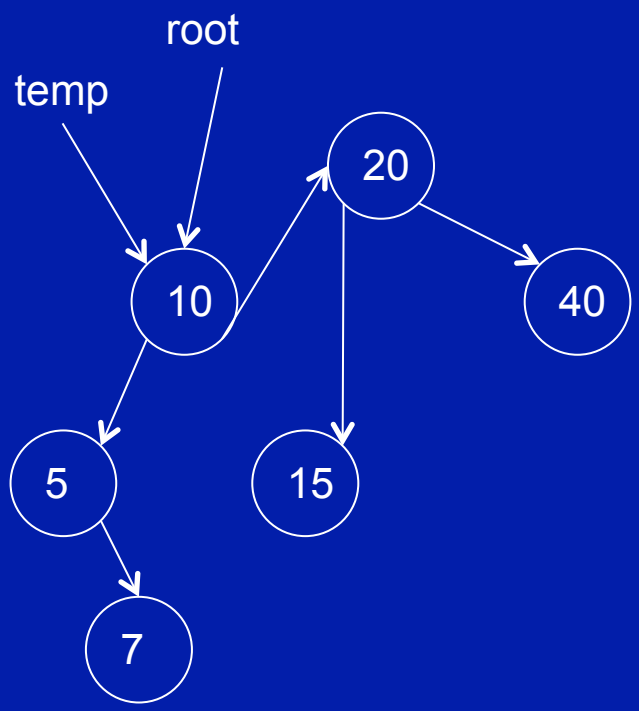


It's all just pointer manipulation.

Here's the algorithm for right rotation from your textbook:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Balance

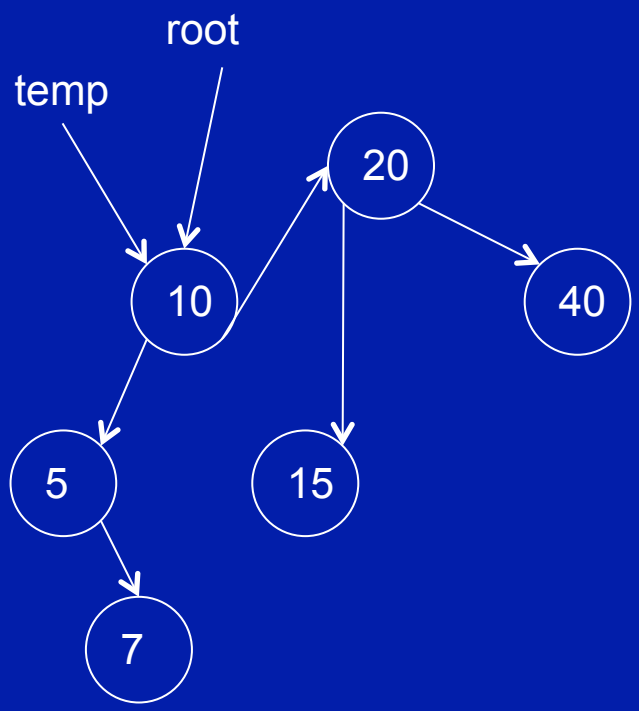


It's all just pointer manipulation.

Here's the algorithm for right rotation from your textbook:

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Balance



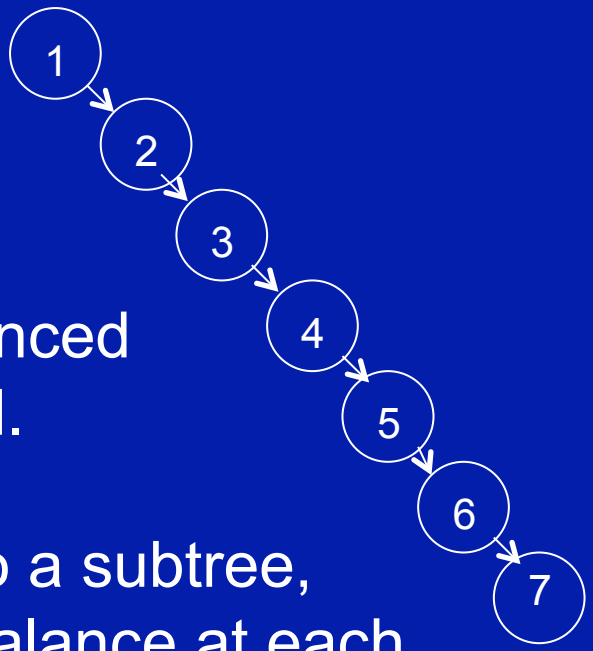
It's not magic, it's just programming.

Rotate left is just the mirror image of rotate right. We'll leave that for you to do on your own. (Or maybe on a final exam question.)

1. Remember the value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp

Self-balancing search trees

As we've seen, we can come up with a severely unbalanced BST when building the tree.



The easiest way to keep a tree balanced is never to let it become unbalanced.

So when a new node is inserted into a subtree, the insertion algorithm checks the balance at each parent node up the insertion path.

AVL trees

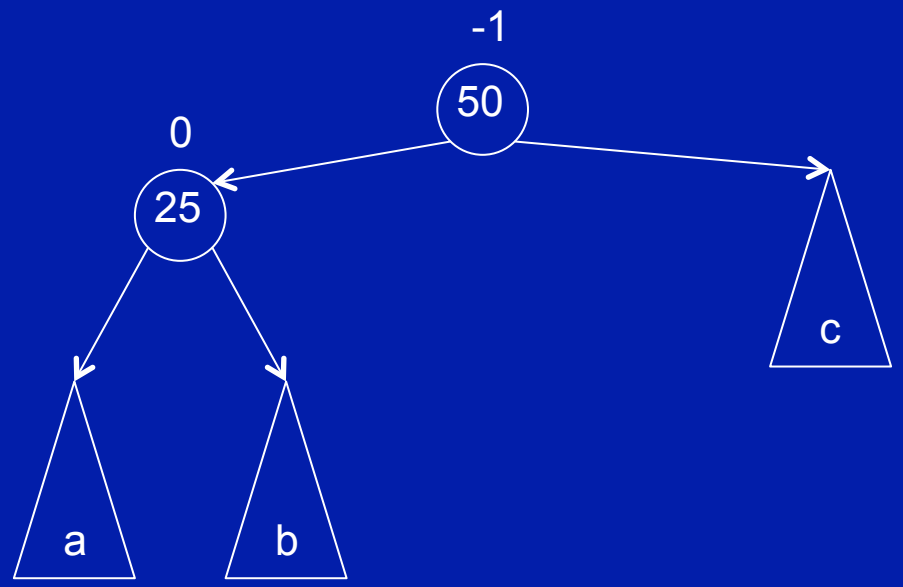
Adel'son-Vel'skii and Landis (AVL) created an algorithm for automatically maintaining the overall balance in a binary search tree.

The algorithm tracks the difference in height of each subtree. (Either by storing a balance indication at each node, or the height of the subtree whose root is at that node.)

As items are inserted in or deleted from the tree, the balance of each subtree is updated from the insertion point up to the root. If the balance ever goes critical (outside the range of -1 to 1), the subtree is rotated to bring it back to balance.

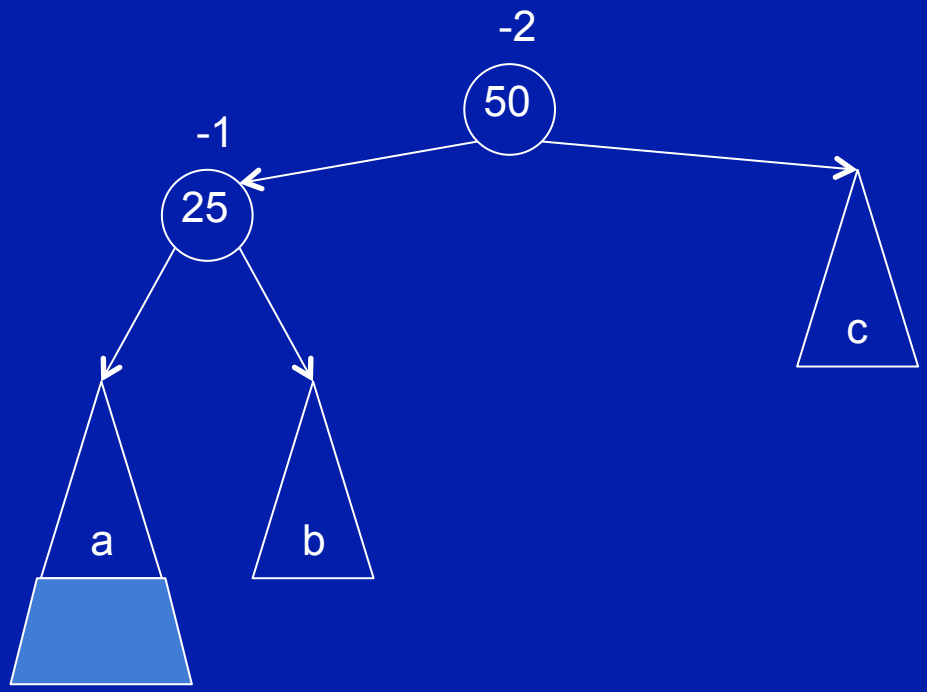
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



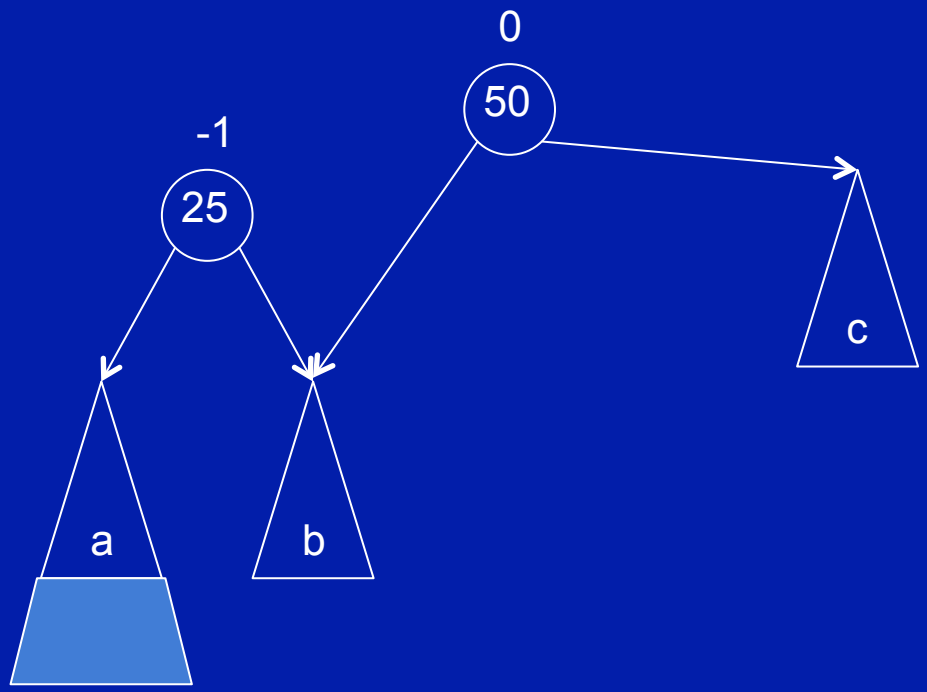
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



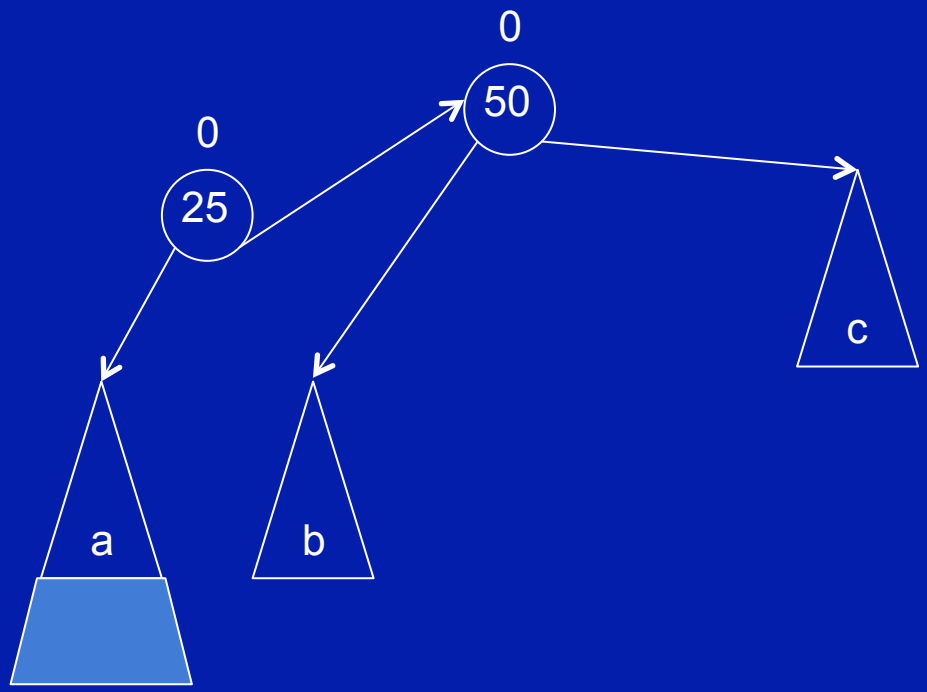
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



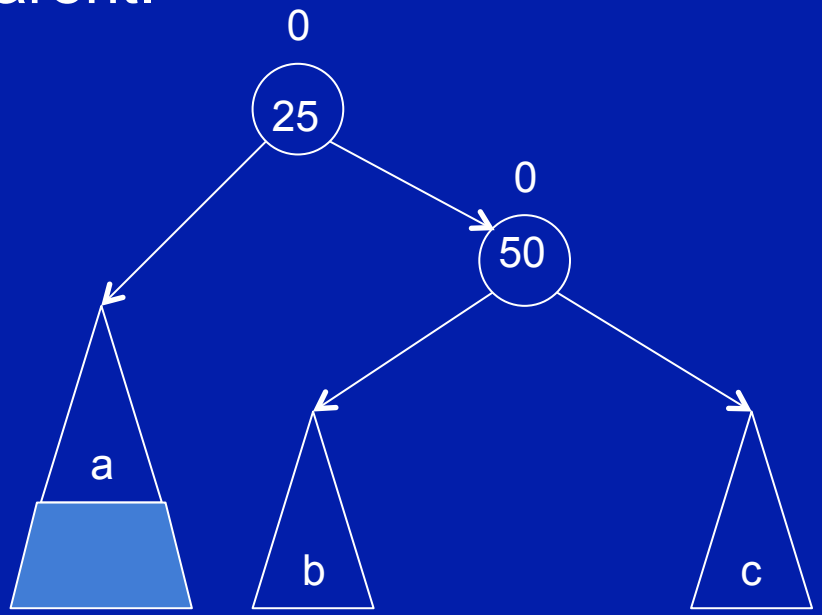
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



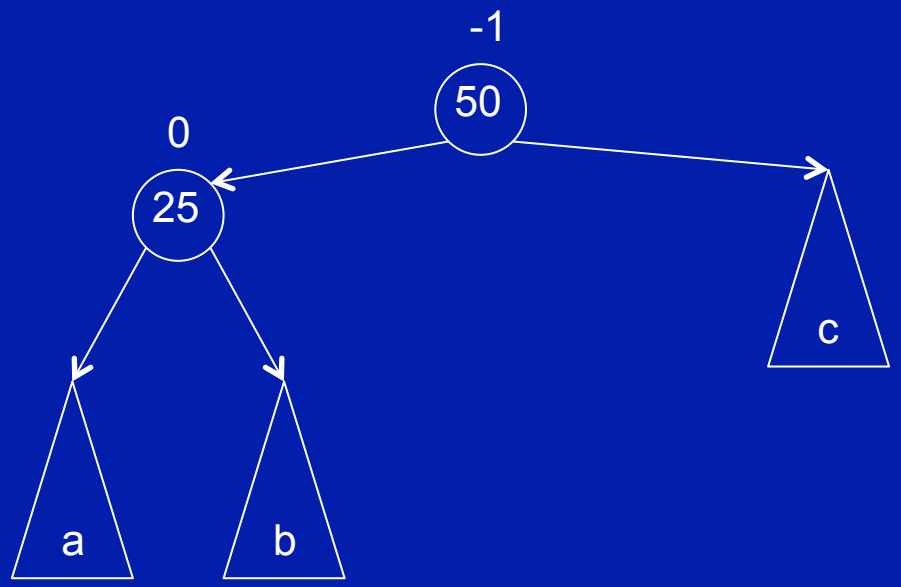
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



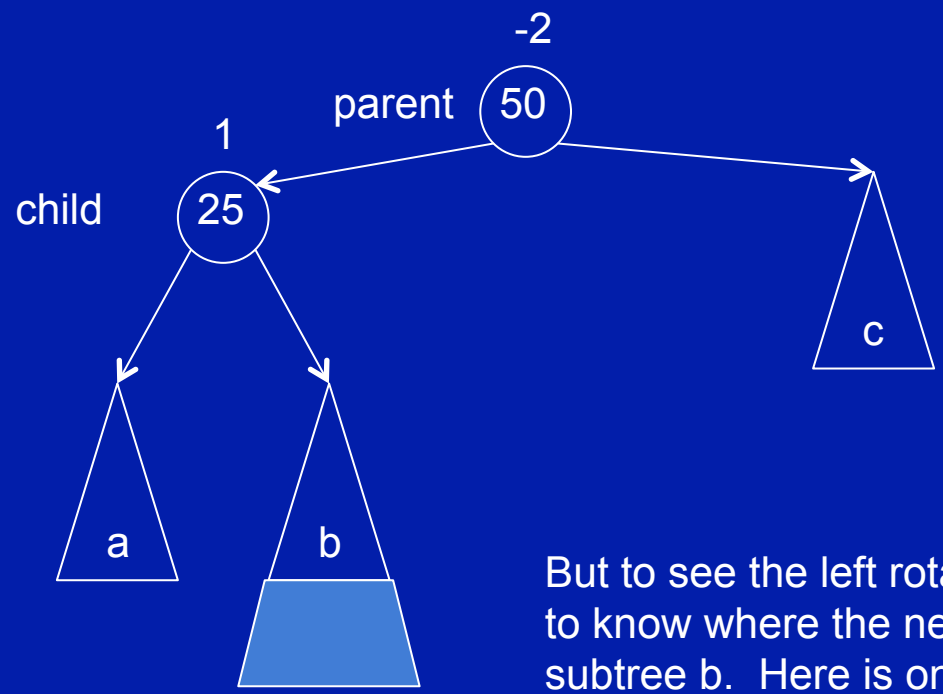
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



AVL trees

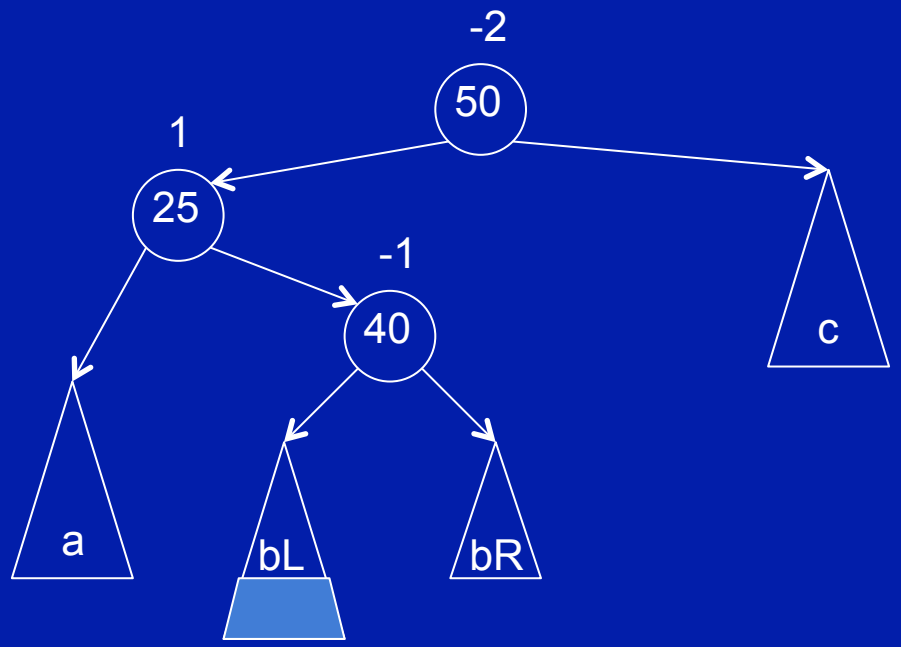
The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



But to see the left rotation details, we need to know where the new node was inserted in subtree b. Here is one possibility:

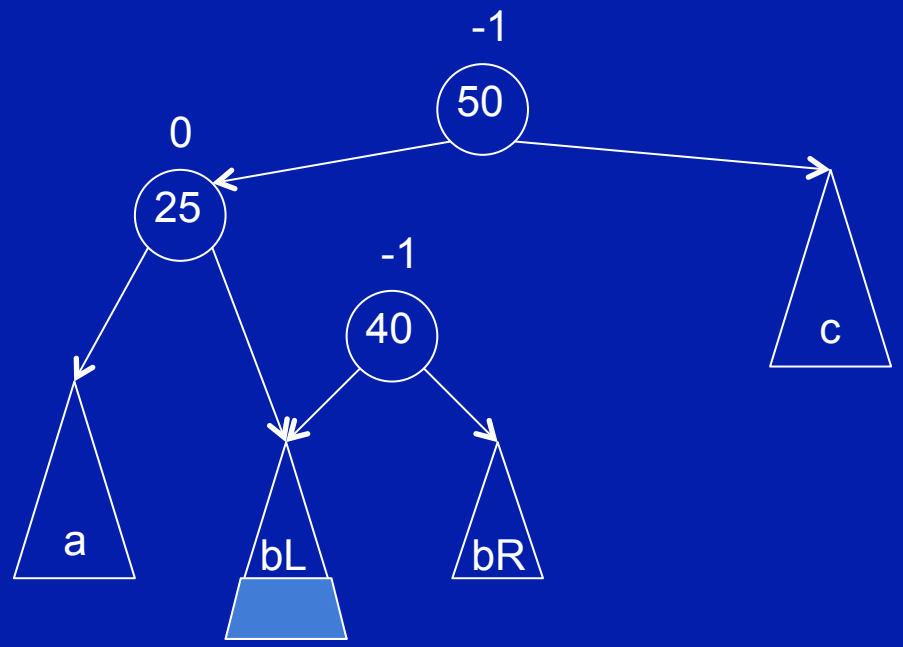
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



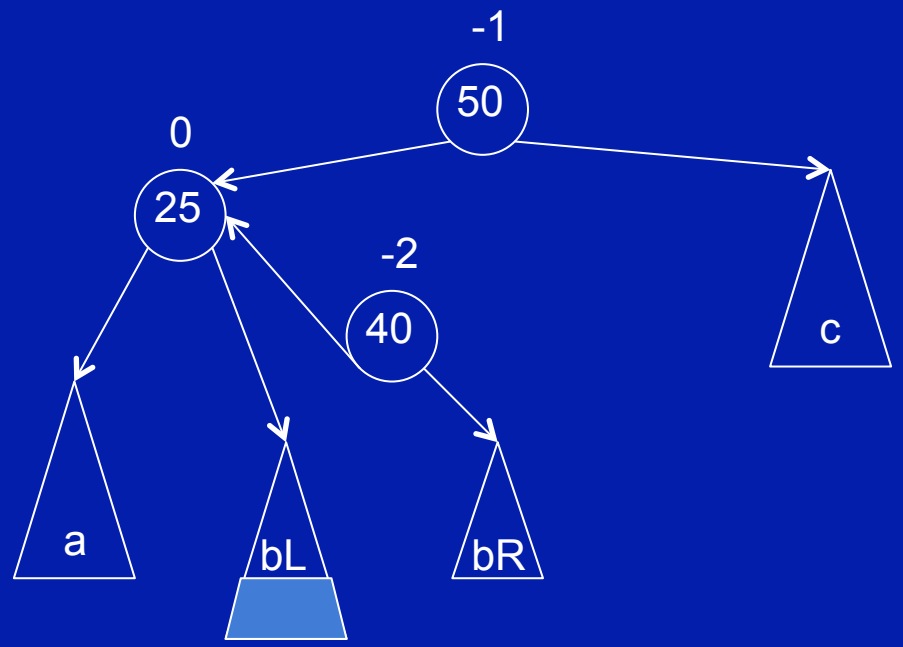
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



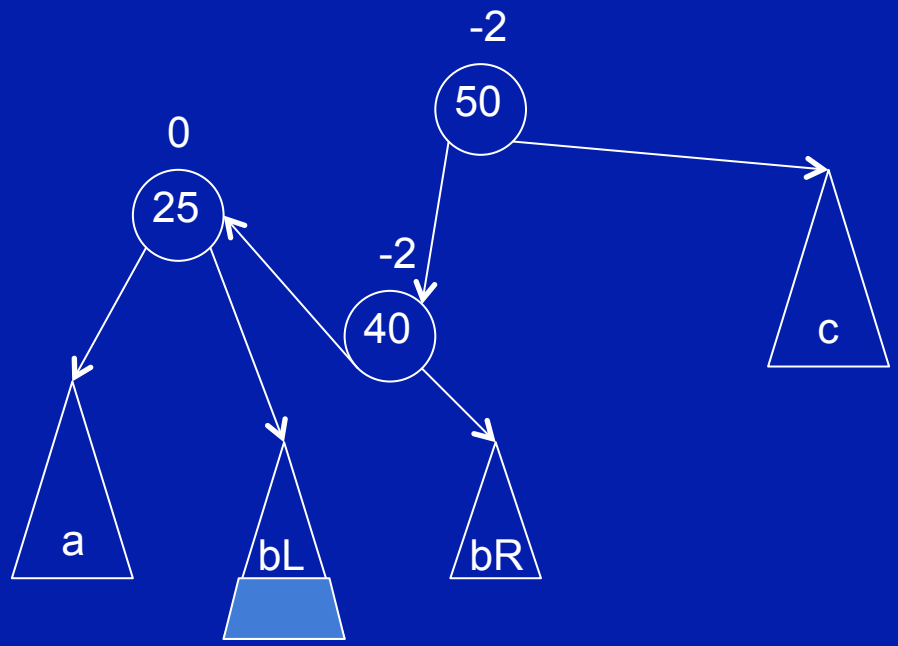
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



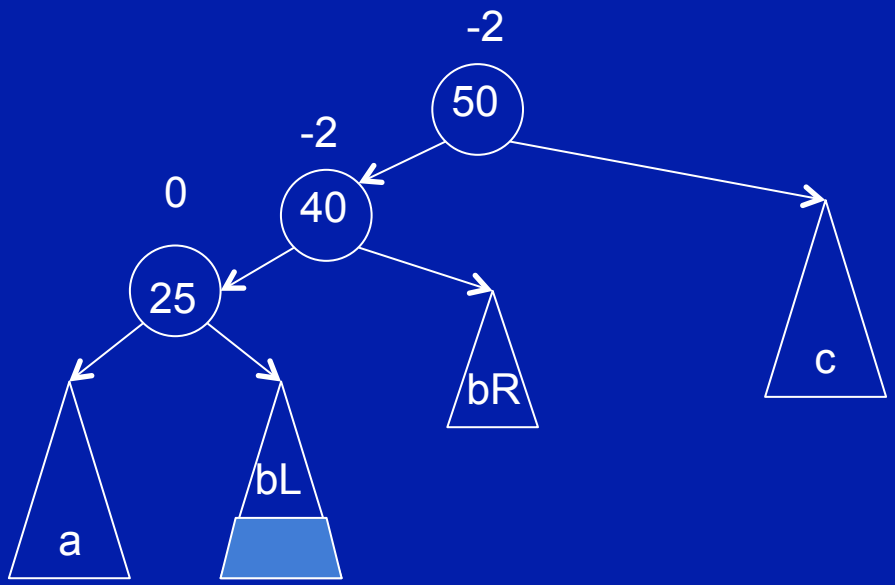
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



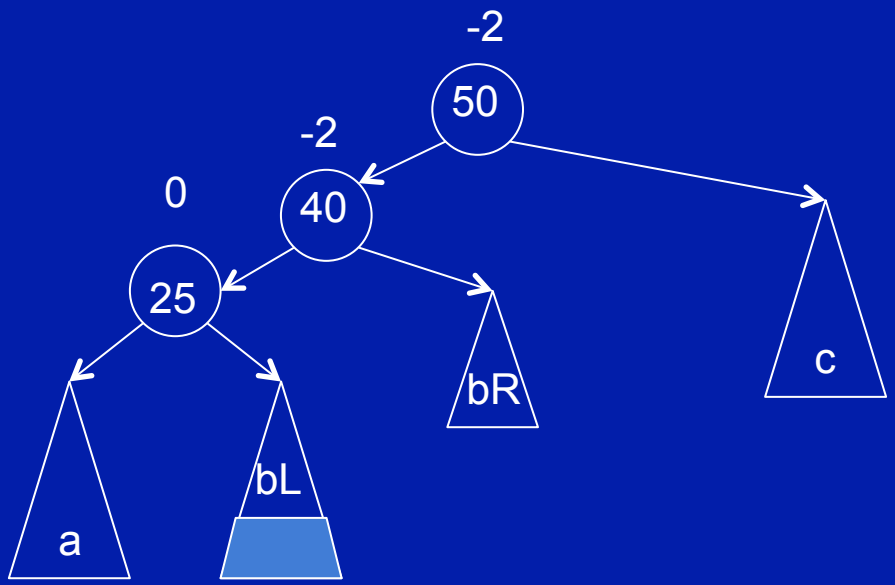
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



AVL trees

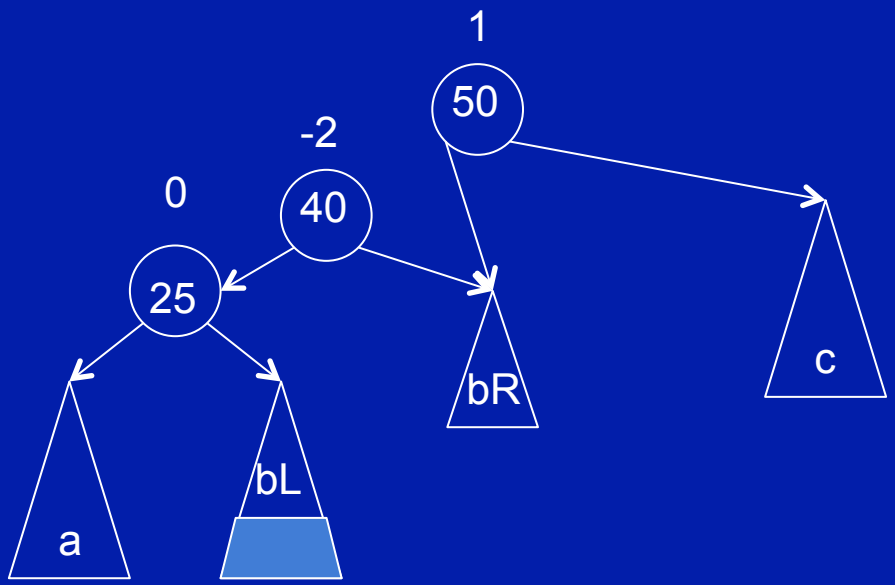
The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



Now right rotation around parent

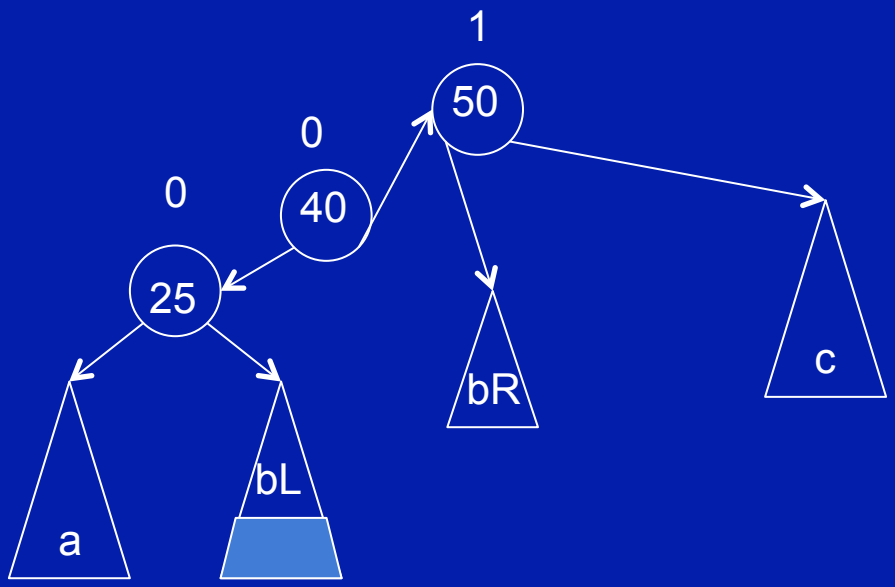
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



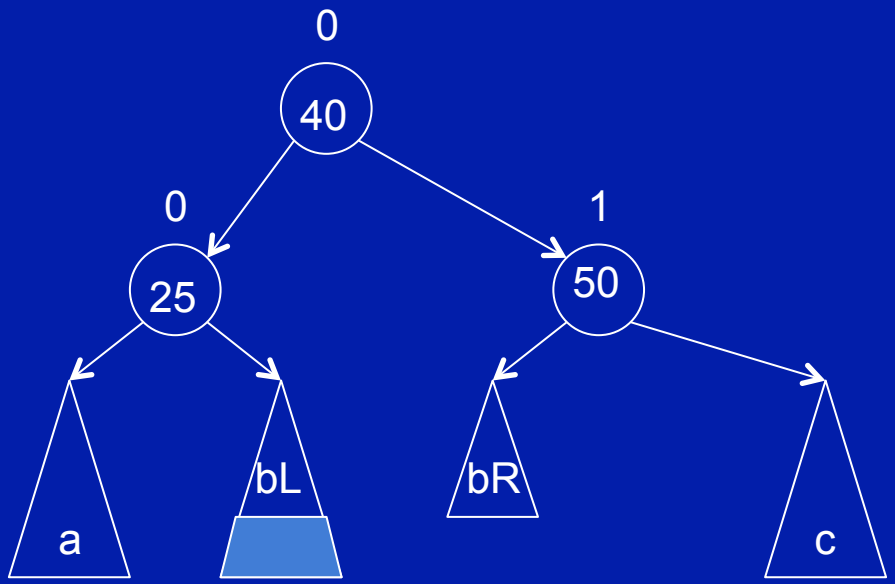
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



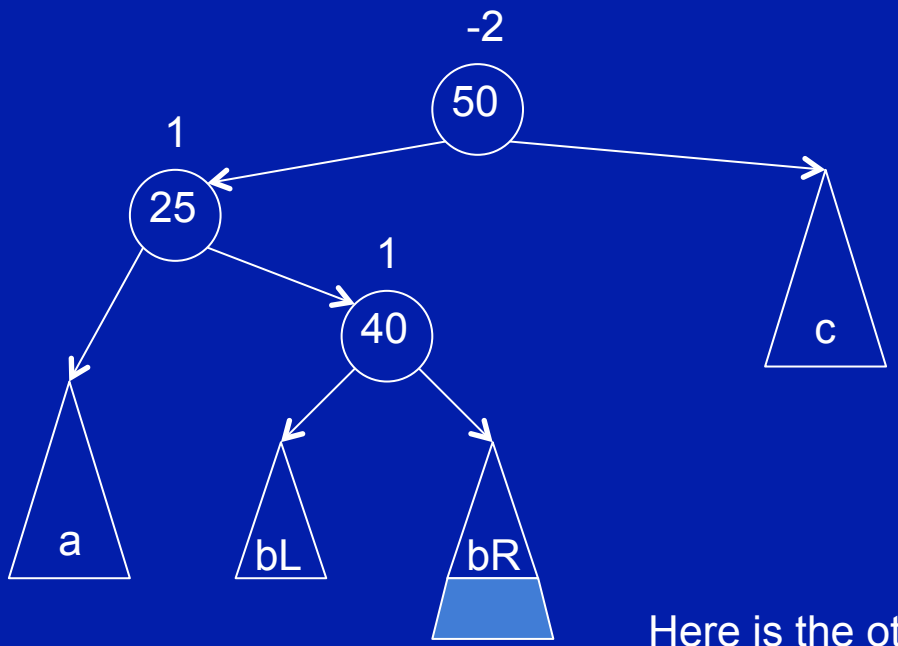
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



AVL trees

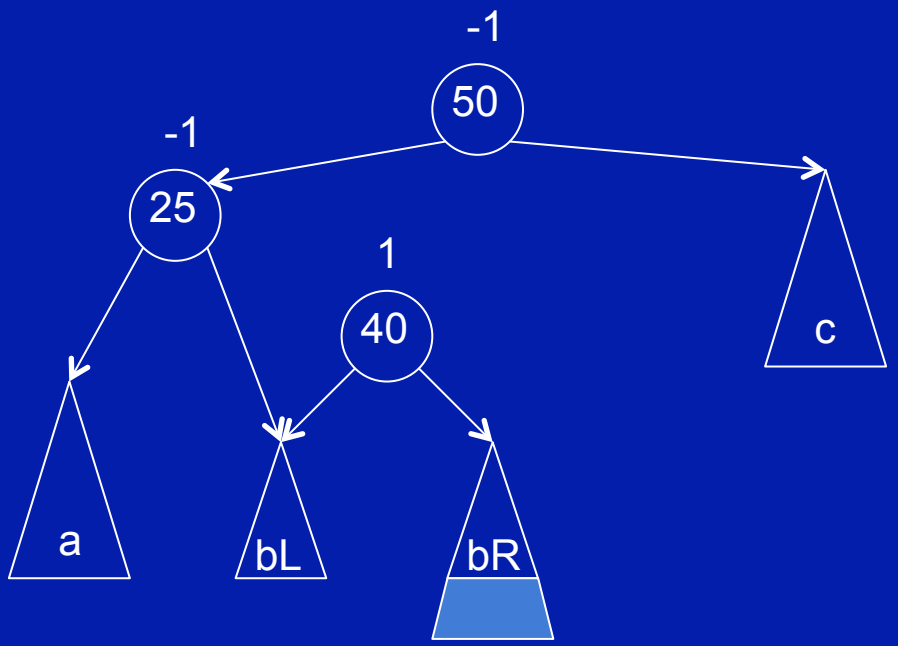
The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



Here is the other possibility. Start by rotate left around child.

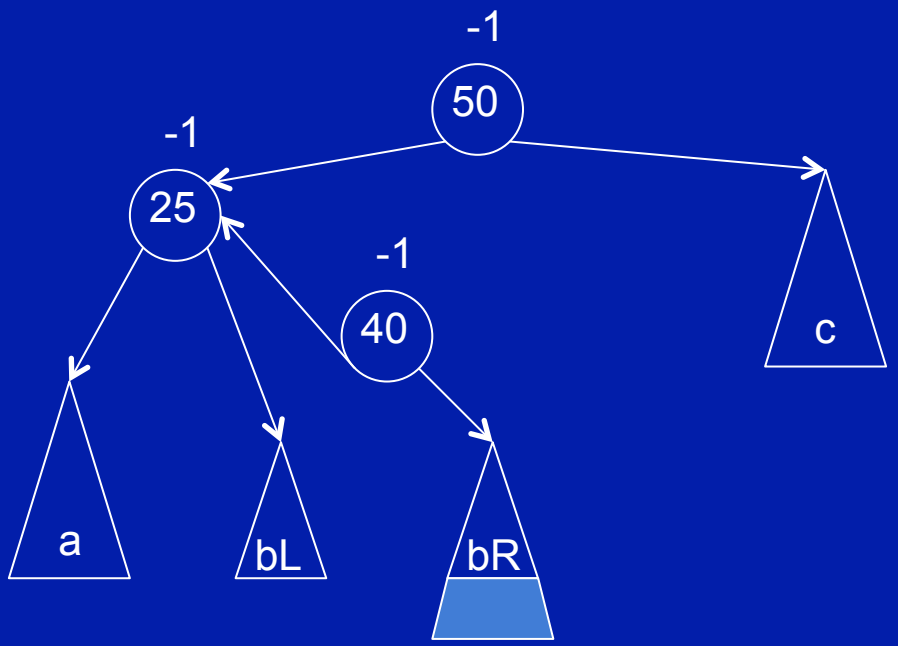
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



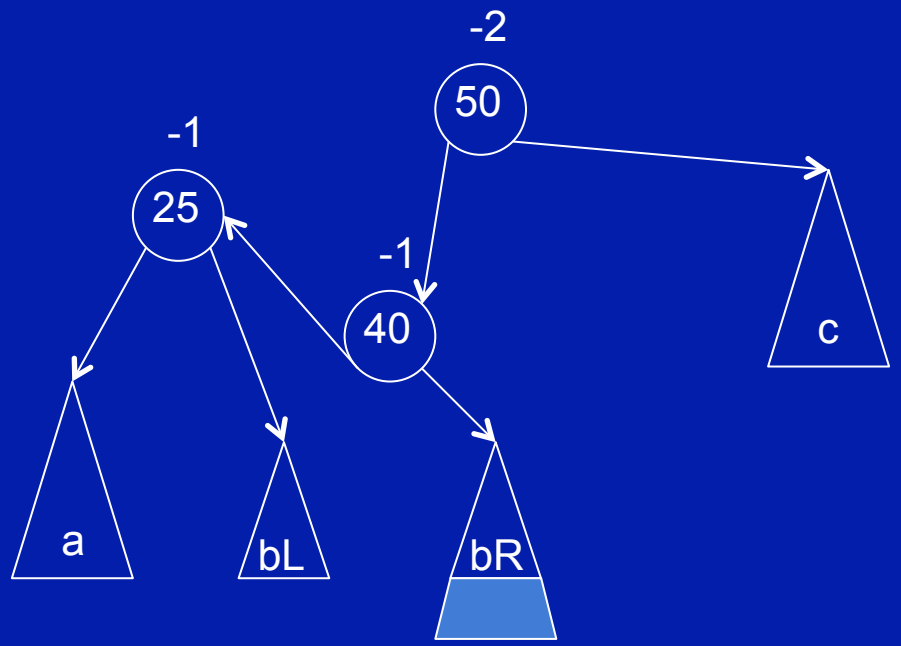
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



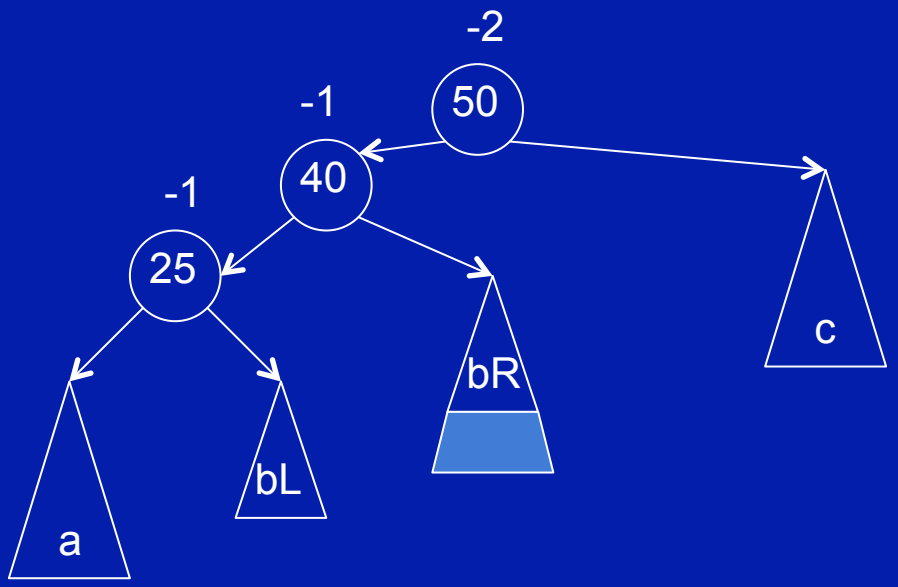
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



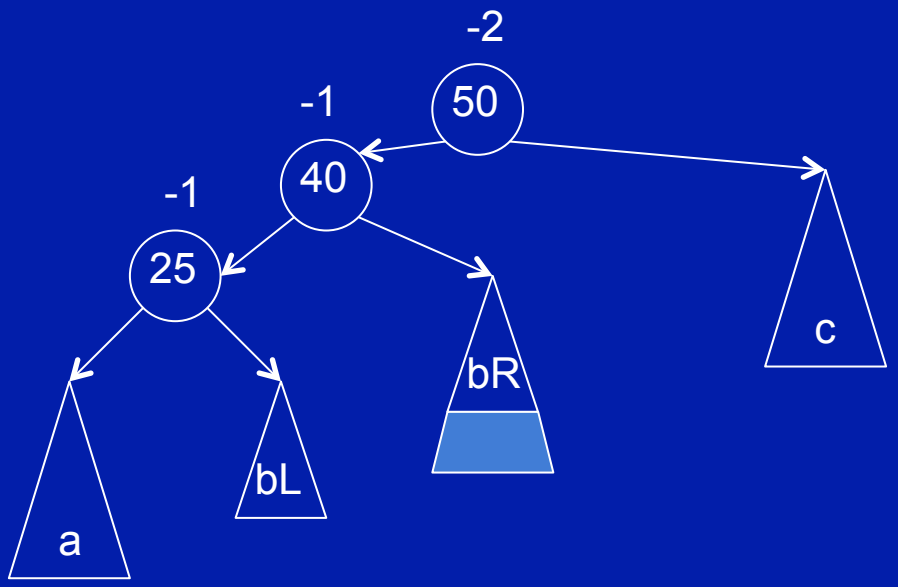
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



AVL trees

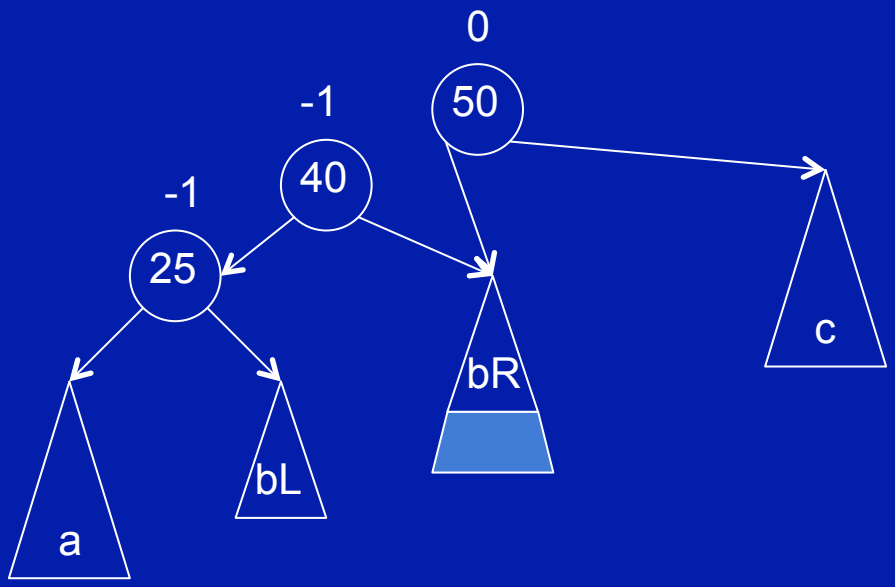
The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



Now right rotation around parent

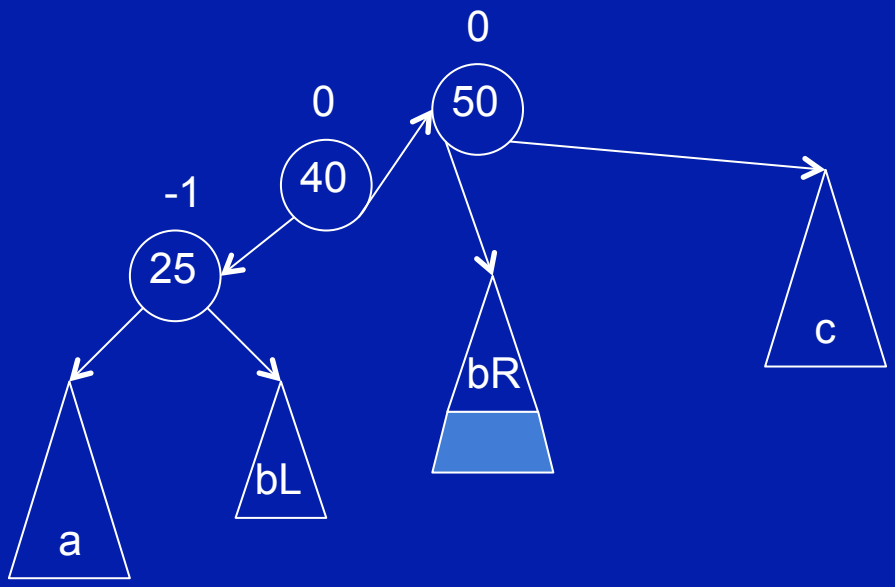
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



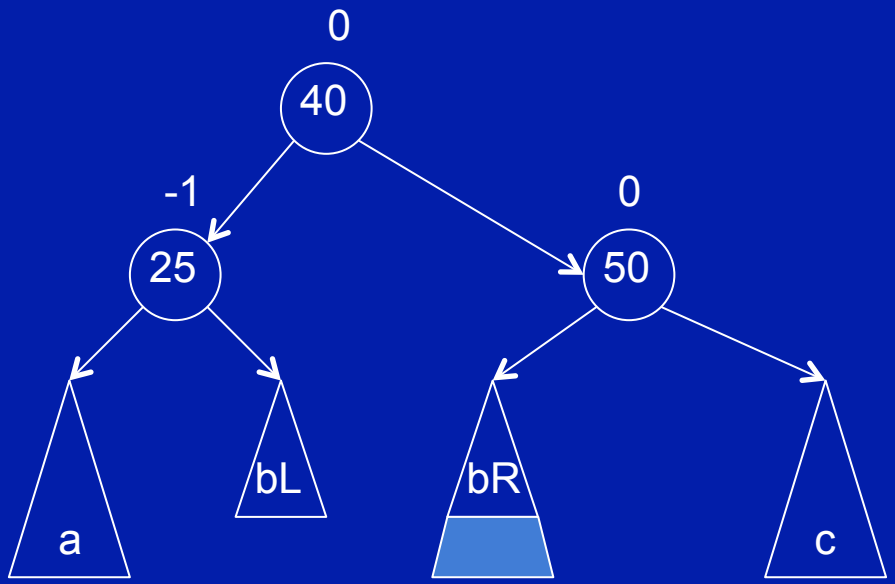
AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Right tree (parent balance is -2, child balance is +1). Fix by rotating left around the child then right around the parent.



AVL trees

The AVL algorithm looks for four kinds of unbalanced trees:

The Right-Right tree (parent and child nodes are both right-heavy, parent balance is +2, child balance is +1). Fix by rotating left around the parent.

The Right-Left tree (parent balance is +2, child balance is -1). Fix by rotating right around the child then left around the parent.

Right-Right is the mirror image of Left-Left.
Right-Left is the mirror image of Left-Right.
You should work out the details on your own.

Balanced, non-binary trees

Not all search trees have to be binary trees to be useful.

If we increase the branching factor in our search trees from 2 to m, the worst and average case search times can be improved to $O(\log_m n)$ comparisons (instead of $O(\log_2 m)$ comparisons). In really big file systems, that's a savings worth pursuing.

Increasing the branching factor of a search tree greatly reduces the number of levels which must be searched for a given key.

B-trees

B-tree properties:

A B-tree of order m is a tree with the following properties:

- the root has at least two children unless it is a leaf
- no node has more than m children
- every node except for the root and leaves has at least ceiling(m/2) children.
- all leaves appear on the same level
- an internal node with k children contains exactly k – 1 keys

“B” may stand for Bayer (one of its creators), Balanced, Broad, Bushy, or even Boeing (Bayer’s employer). but it does not stand for Binary.

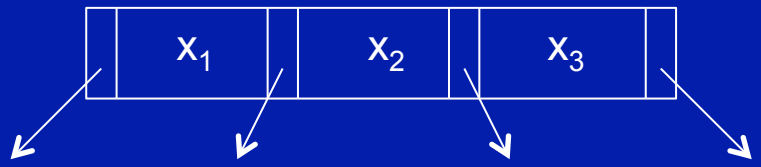
B-trees

What's in the nodes?

The literature is fuzzy and inconsistent, especially with respect to leaf nodes, but often the story goes like this:

An internal node in an order m B-tree contains an ordered set of as many as m – 1 keys and m child pointers.

The keys are "road signs" that direct traversal through the tree. A node containing keys x_1, x_2, x_3 will point to values < x_1 in its left-most subtree, >= x_1 and x_2 in the subtree pointed to between x_1 and x_2, and so on.



B-trees

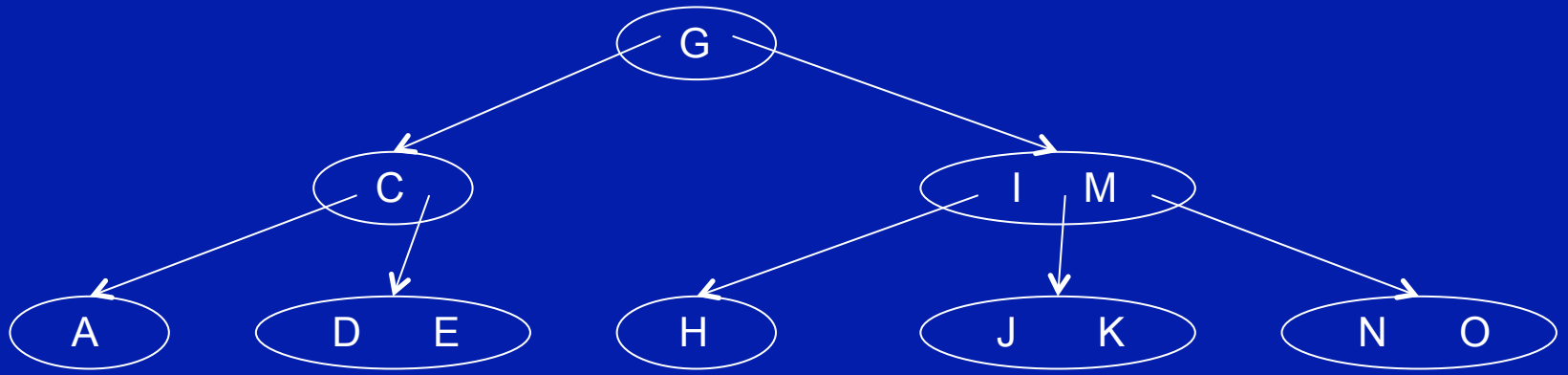
What's in the nodes?

A leaf node in an order m B-tree contains an ordered set of elements that represent data and no child pointers.

The elements may be the actual data itself or pointers to data.

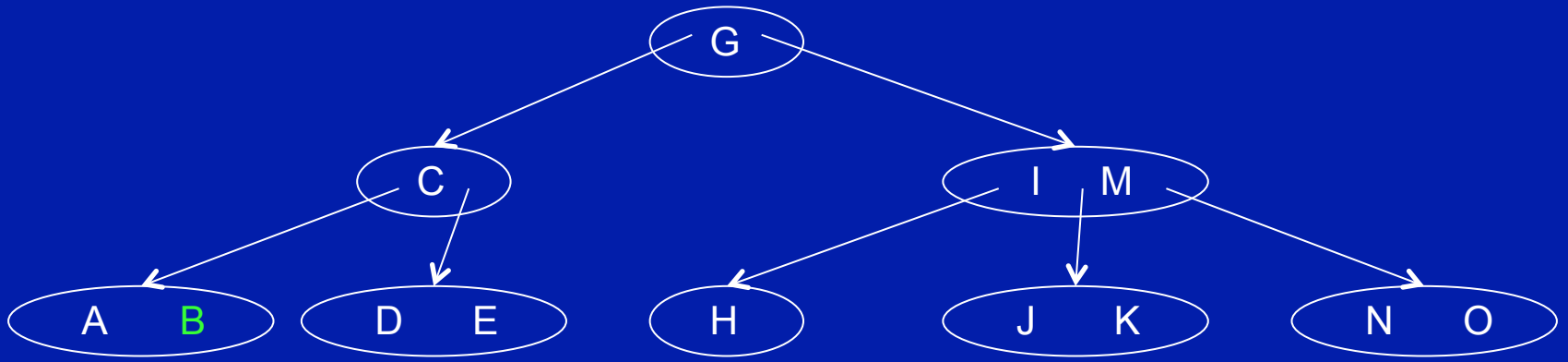
To keep things simple, we're not going to deal with the data, just like your book. We'll just talk about B-trees (and their variants) as if there were only key values and internal nodes, and we'll let the data be a detail that you deal with if/when you implement your own B-trees.

2-3 trees



The keys are stored in order such that an inorder traversal visits the keys in sorted order, just like with a binary search tree.

2-3 trees

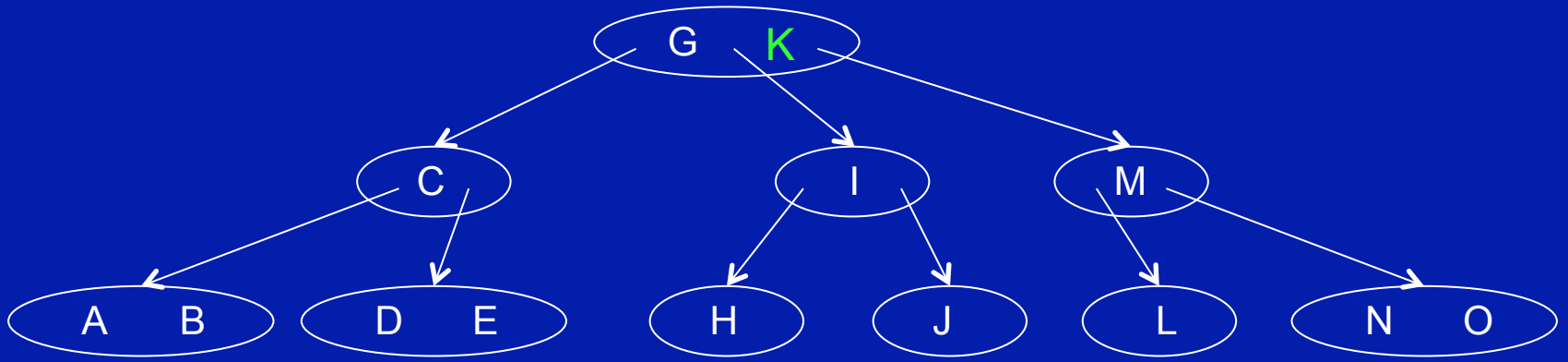


Insertion is fairly simple

To insert key B, we just search for B and add B to the node where it should be.

Depth of tree remains the same. The number of keys in a node is increased. All the B-tree properties are preserved.

2-3 trees



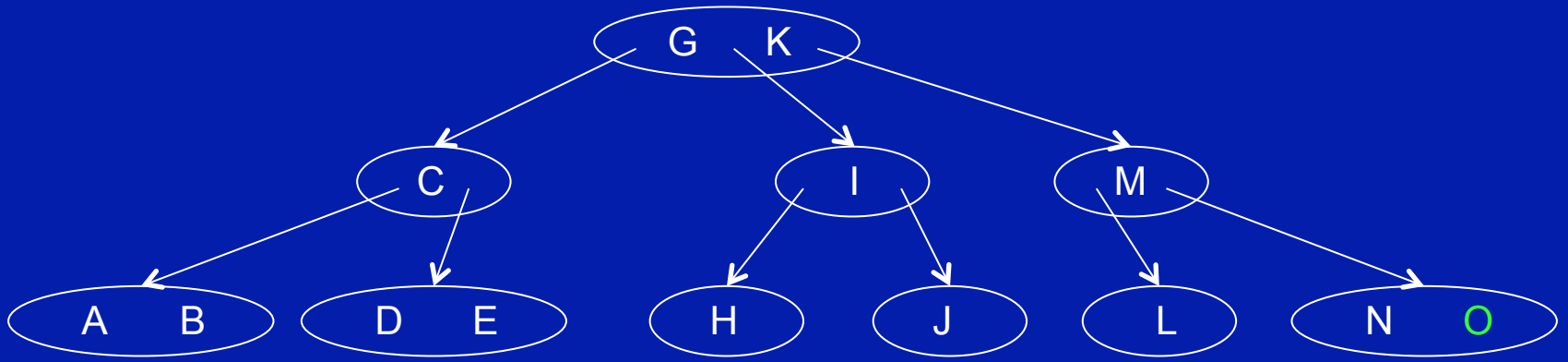
To insert key L, again we search to find the node where it should be and add it.

But this node now has too many keys, so we split the node into two nodes and send the middle key up to the parent.

And now this node has too many keys too, so we split it and send the middle key up again.

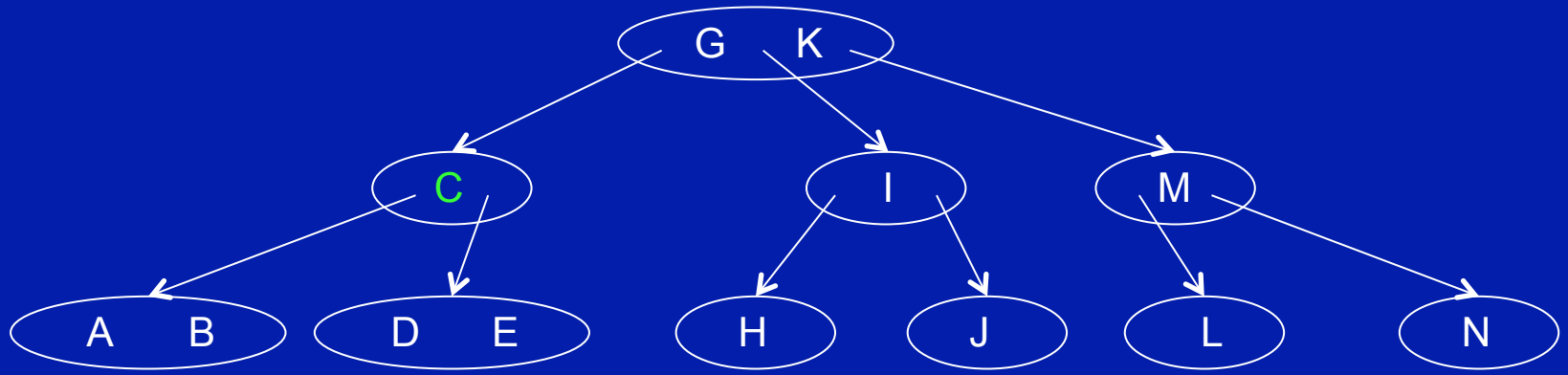
The root node is good, so we're done.

2-3 trees



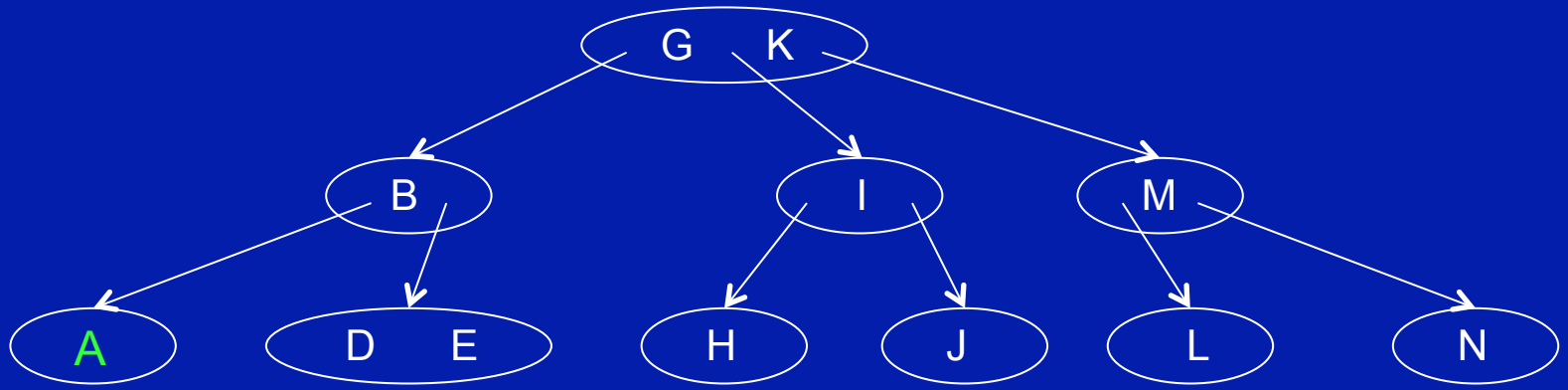
Deletion is sort of the inverse of insertion. If the item to be deleted is in a “leaf” node with two items, we just delete it.

2-3 trees



If the item is not in a “leaf”, it is swapped with its inorder predecessor from a “leaf” node and then deleted from the “leaf” node.

2-3 trees



If removing a node from a leaf causes the leaf to become empty, items from the sibling and parent can be redistributed into that leaf.

2-3 tree performance

2-3 trees require fewer complicated manipulations than AVL trees. There are no rotations.

The number of items that a 2-3 tree can hold is between $2^h - 1$ (all 2-nodes) and $3^h - 1$ (all 3-nodes).

Thus the height of a 2-3 tree is between $\log_3 n$ and $\log_2 n$.

Consequently the search time is O(log n), since logarithms are all related by a constant factor and we ignore constants in the land of Big-O.

2-3-4 trees

A 2-3-4 tree is another specialization of the B-tree. It is a B-tree where m = 4, so the 4-node is added to the 2-node and the 3-node.

Your book says that the addition of this new data item simplifies the insertion logic.

You should definitely give the 2-3-4 tree description in the book some consideration and determine whether the insertion logic really is simplified. (In other words, just because we don't talk about it here doesn't mean I don't expect you to be familiar with it.)

Heaps

A heap is a complete binary tree in which

- If A is a parent node of B, then the value (or key) at A is ordered with respect to the value (or key) of B, and the same ordering applies throughout the heap.* In other words, every subtree of a heap is a heap.

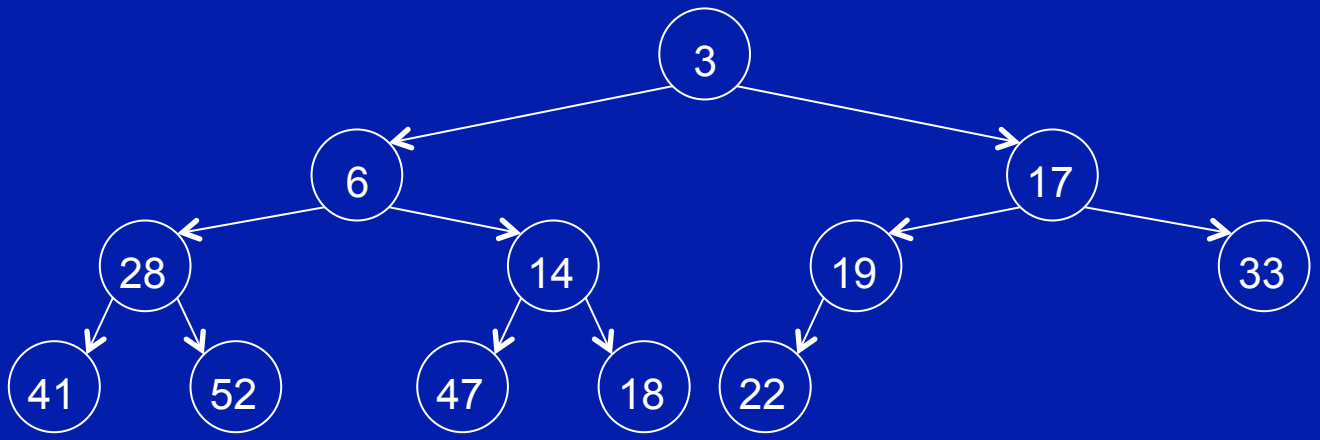
If the values (or keys) of parent nodes are always greater than or equal to those of the child nodes and the largest value (or key) is at the root node, we have a maximum binary heap or *max heap*.

If the values (or keys) of parent nodes are always less than or equal to those of the child nodes and the smallest value (or key) is at the root node, we have a minimum binary heap or *min heap*.

* see Wikipedia

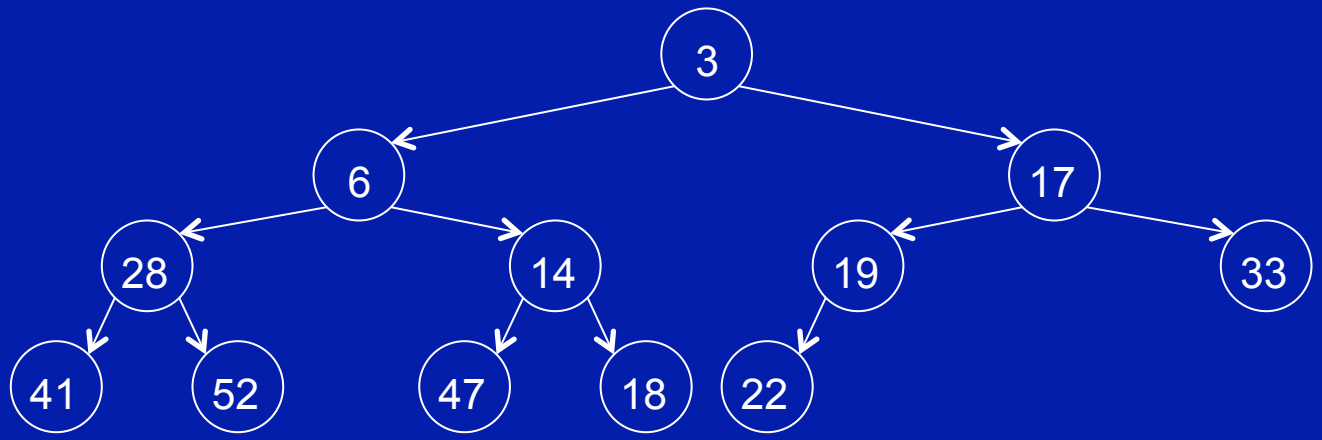
Heaps

Here is a min heap



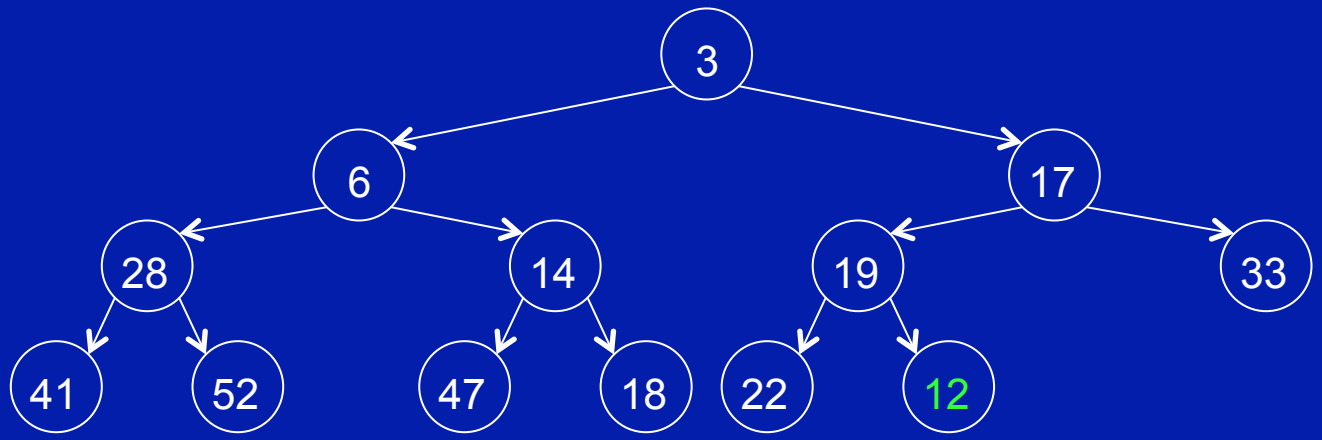
Heaps

Note that there is a top-to-bottom ordering, but there is no side-to-side ordering as we would see in a binary search tree.



Insertion in a heap

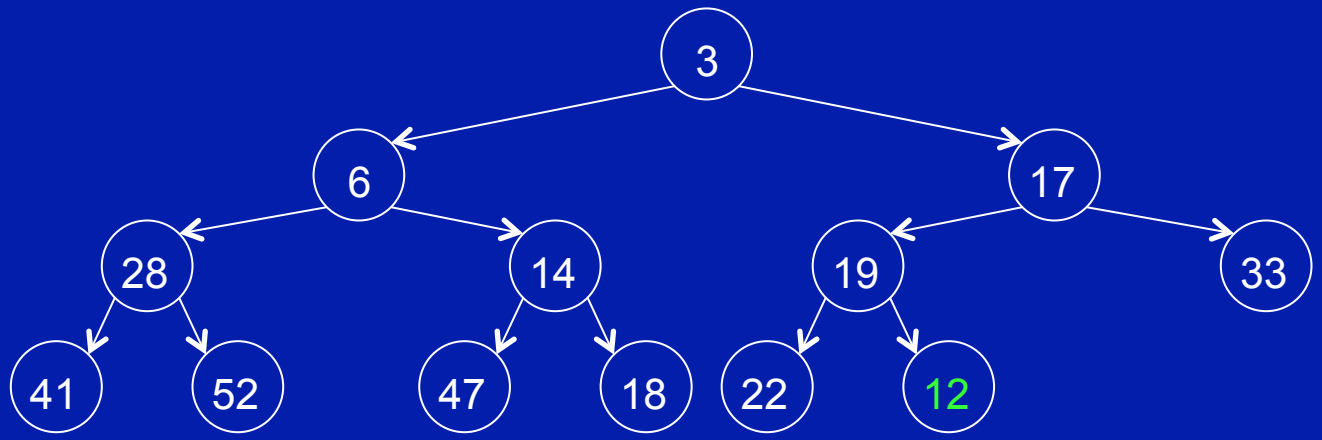
Inserting a new item into a min heap starts with adding the item to the heap at the next available location in the complete binary tree:



Insertion in a heap

Algorithm for insertion into a min heap:

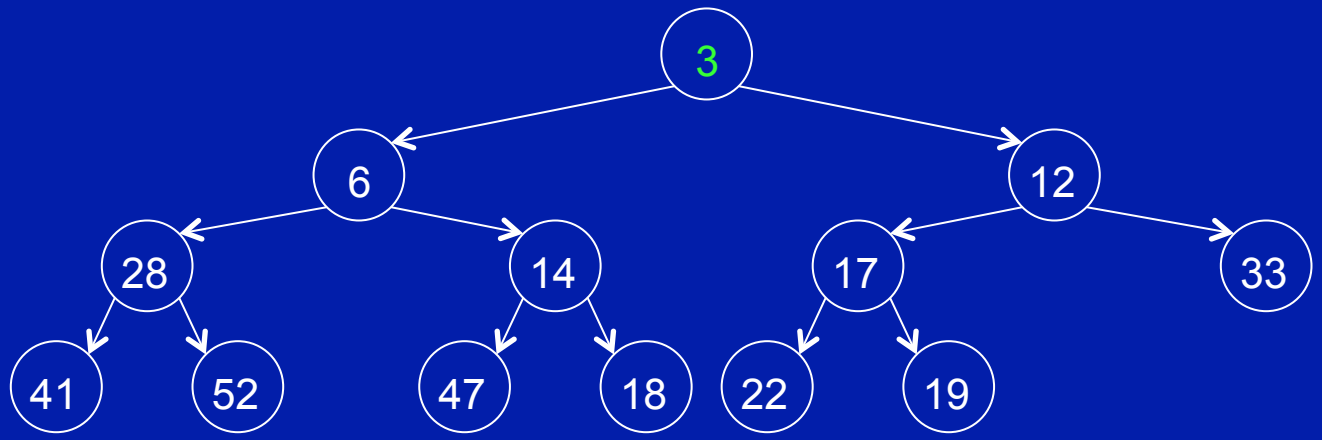
1. insert the new item in the next position at the bottom of the heap
2. while new item is not at the root and new item is smaller than its parent
3. swap the new item with its parent, moving the new item up the heap



Deletion from a heap

Algorithm for deletion from a min heap:

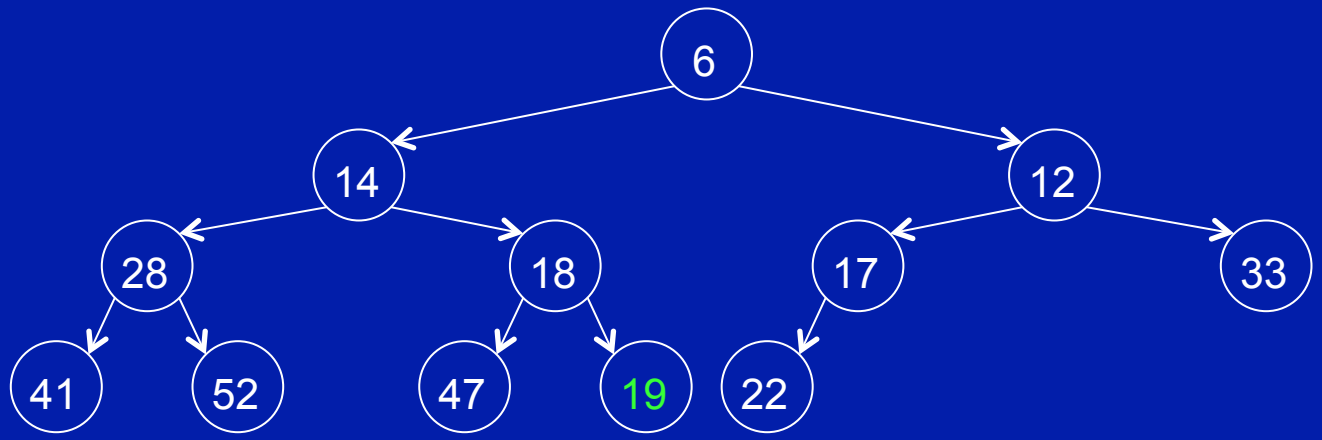
1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3. swap item LIH with the smaller of its children, moving LIH down the heap



Time complexity

When doing a ReheapUp or ReheapDown, the number of operations depends on the height of the tree.

We only ever traverse one path of the tree.

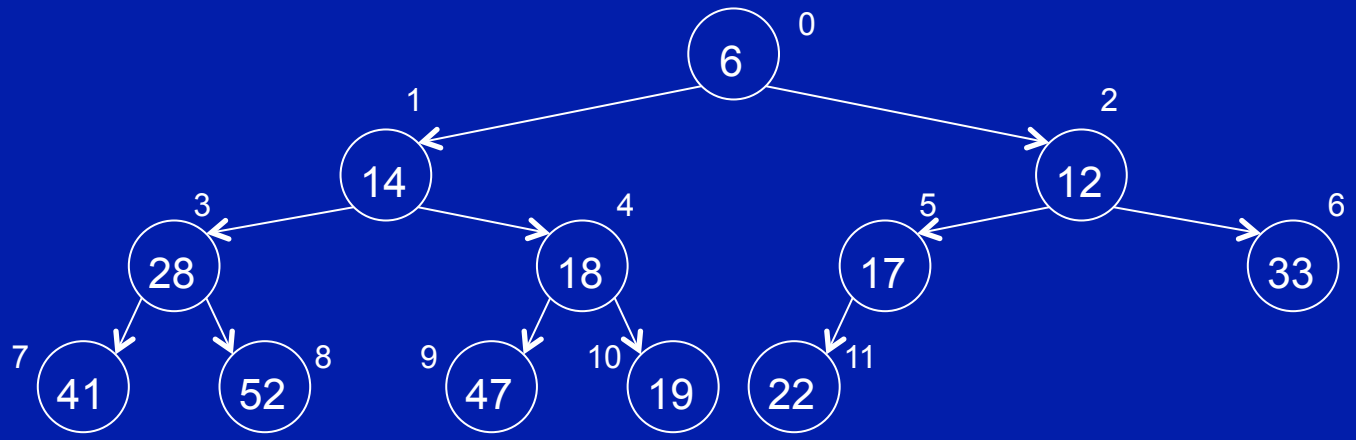


A complete binary tree of height h always has between 2^h and $2^{h+1} – 1$ nodes (= n). The height of the heap is, therefore, floor(lg n).

So the time complexity of ReheapUp and ReheapDown is O(lg n).

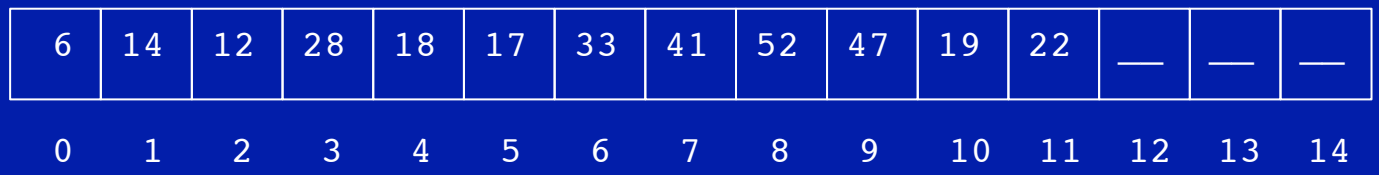
Implementing a heap

If we number the nodes in a complete binary tree by level, and left to right within a level, we get:



While other types of trees are implemented as linked list structures, a complete binary tree can be represented in contiguous storage (e.g., array, vector) using the numbering or indexing described above...

Implementing a heap



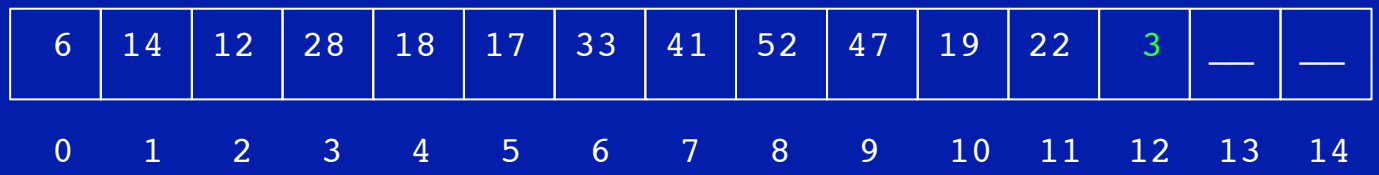
Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Implementing a heap



Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

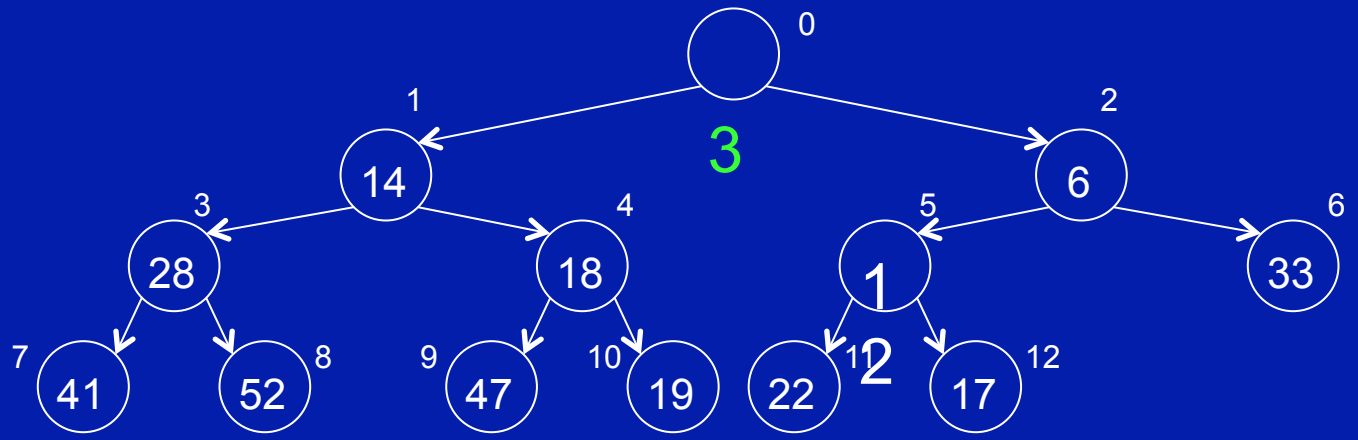
we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3
Less than or equal to its parent? (index = 5, value = 17)

Priority queues

The heap data structure provides the basis for the priority queue abstract data type. If we assume that a value in the heap represents the priority of a job (smaller numbers having higher priority, like number of pages in a print request), then as values are added, the smaller, higher priority values float toward the top, and lower priority values fall to the bottom.

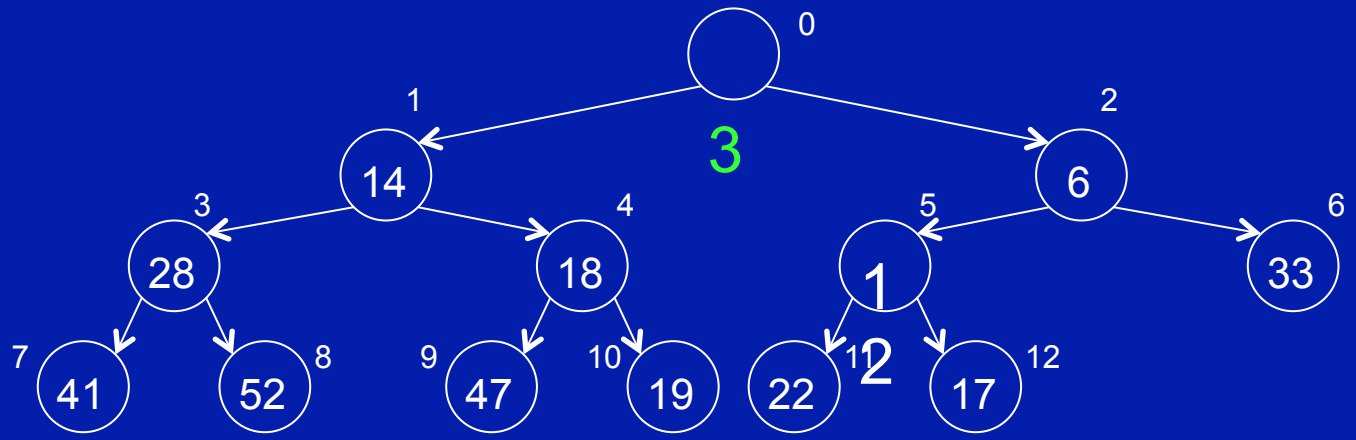


When we want to select the highest priority job in the queue, we just grab the item at the top of the heap (e.g. 3). We then move the last item in the heap to the top and perform ReheapDown as described previously.

Priority queues

Removing the minimum value (i.e., highest priority item) from the priority queue requires ReheapDown, which takes O(lg n) time.

Adding a new item to the priority queue requires ReheapUp, which also takes O(lg n) time.



Heapsort

We can use a heap as the foundation for a very efficient sorting algorithm called heapsort.

Heapsort consists of two phases:

Heapify: build a heap using the elements to be sorted

Sort: Use the heap to sort the data

This can all be done in place in the array that holds the heap, but it's easier to see if we draw the trees instead of the array. Once you see how it works with trees, make sure you understand how it works in place in the array.

Heapsort

Here's the heapify component:

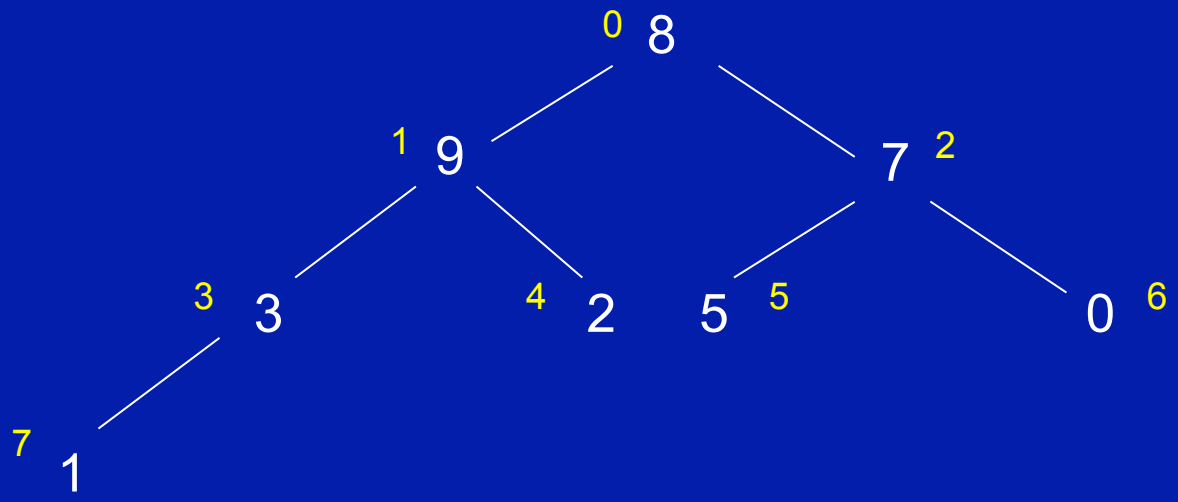
for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

Heapsort redux

Here's a faster heapify algorithm (O(n), according to Wikipedia). Just put all the elements into a complete binary tree, then apply ReheapDown from the bottom up (starting with the last parent).

Let `index` be the subscript of the last parent node in the tree.
While `index` >= 0
  Perform a ReheapDown operation starting with the node at `index`
  Decrement `index` by 1

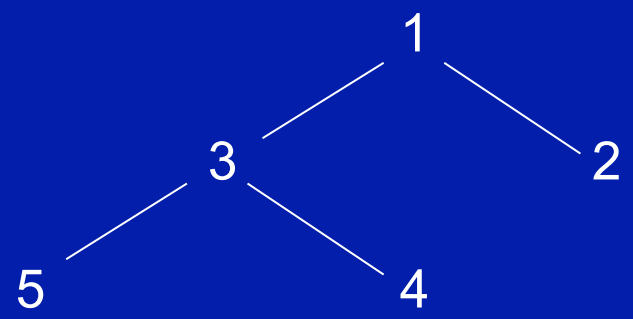


Here are the indexes. Remember that these are the same as the array indexes - heaps are implemented as arrays. Heapify and heapsort are done in place in the array.

Heapsort

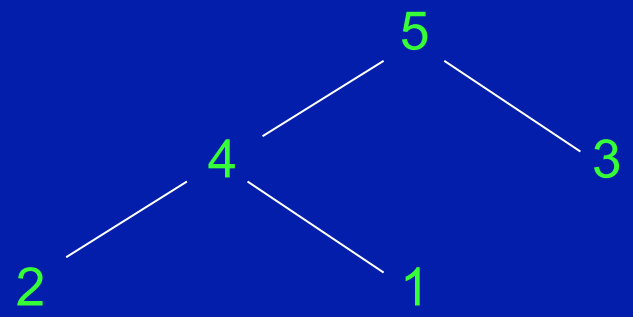
Now for the sorting:

while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort analysis

A heap of size n has lg n levels. Building a heap requires finding the correct location for each item in a heap with lg n levels. Because there are n items to insert in the heap and each insert is O(lg n), the time to heapify the original unsorted sequence using ReheapUp or Sift-up is O(n lg n). During sorting, we have n items to remove from the heap, which then is also O(n lg n). Because we can do it all in the orginal array, no extra storage is required.



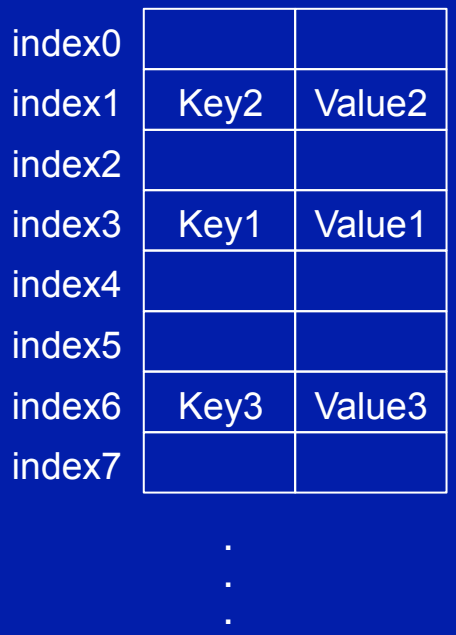
Search, revisited

To place those key-value pairs in a sequential data structure like an array, we need a mapping between the unique keys and individual array indexes.

A very simple solution would be to have the keys and the array indexes be the same (i.e., your driver's licence number is an index in ICBC's array).

In other words, if we assign driver's licence numbers in the range 0 to N-1, then we can store the data for driver k in location k of the array (of size N).

Given a driver's licence number, we could access the driver's data in O(1) time!



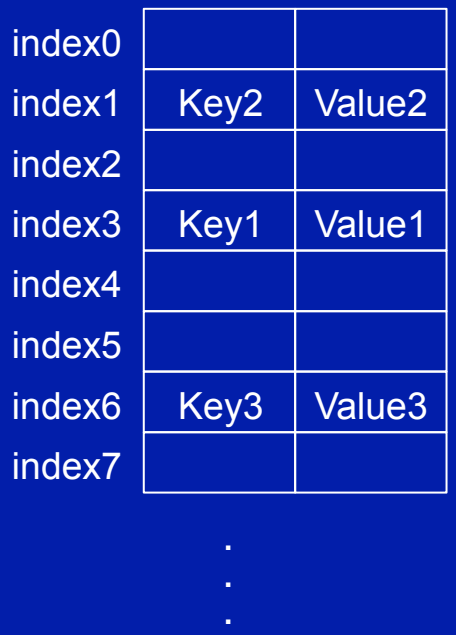
Hash functions

The solution to our problem is called hashing.

We define a function that transforms keys into numeric array indices. Such a function is called a hash function. The index is a hash index. The table is a hash table.

Given a key k, our hash function h has to generate an index i in the range of 0 to N-1 where N is the number of locations in the array.

h(Key1) ➔ index3
h(Key2) ➔ index1
h(Key3) ➔ index6



Hash functions

A hash function can be thought of as the composition of two functions:

1. The hash code map:
   h_1: keys ➔ integers

2. The compression map:
   h_2: integers ➔ [0, N – 1]

The hash code map is applied to the key first, then the compression map is applied to the result:

$h(x) = h_2(h_1(x))$

Hash functions

In this context, “to hash” means to chop something up or to make a mess of it. The main idea in building a hash function is often to “chop off” some aspects of the key and to use this partial information as the basis for searching.

Two desirable attributes of a hash function:

1. Fast to compute
2. Generate indexes that are distributed throughout the table (array) in an apparently random and uniform way.

Why the first one? Obvious. Why the second one? We’ll talk about that soon.

Strategies for building hash functions

A common, simple, and often effective mathematical expression for basic hash functions has the form:

$$h(k) = (ak + b) \bmod N$$

where a and b are integers. This works even better if N (the array size) is a prime number.

It works even better still if $N \neq r^x \pm y$ where r is the radix of the character set (e.g., 64, 128, 256) and x and y are small integers.

The pigeonhole principle

Let X and Y be finite sets where |X| > |Y|. If f:X -> Y, then $f(x_1) = f(x_2)$ for some $x_1, x_2 \in X$, where $x1 \neq x2$.

Informally: If k+1 pigeons fly into k pigeonholes, some pigeonhole must contain at least 2 pigeons.

The pigeonhole principle

In other words, if you have more key-value pairs than places to put them, they'll have to double (or more) up, even with the best, most perfect hash function you can create.

The pigeonhole principle

Also, even if you have fewer key-value pairs than places to put them, a less-than-perfect hash function may map different keys to the same index.

The pigeonhole principle

When your function maps two different keys to the same index, you have a collision.

The probability of a collision increases as more values are entered into the table.

Collision resolution

To review, if a hash function h maps two different keys x and y to the same index (i.e., $x \neq y$ and $h(x) = h(y)$), then x and y collide.

A hash function that distributes items randomly will cause collisions, even when the number of items hashed is small.

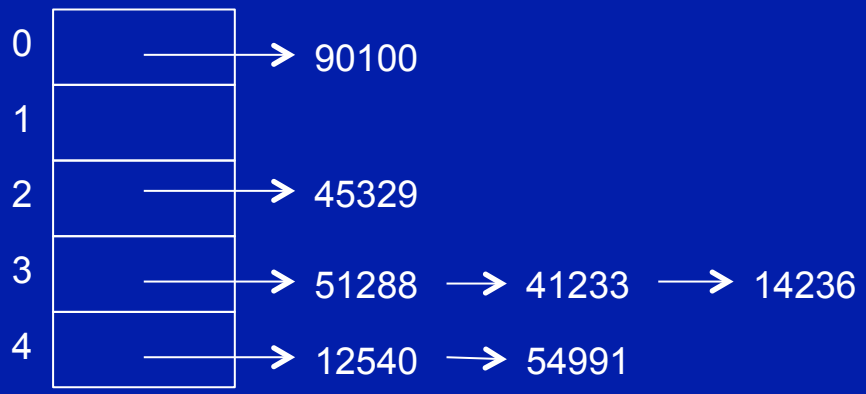
Consequently, we need a means of resolving collisions.

Chaining

Have each hash table location point to a linked list (chain) of items that hash to this location.

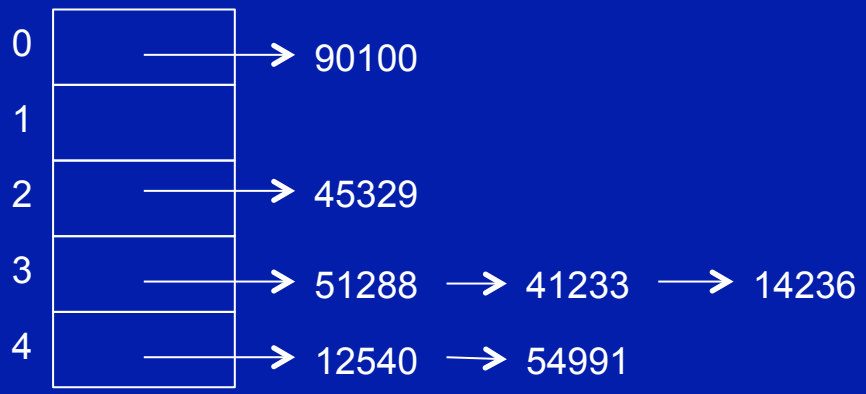
Example: Suppose h(x) = floor(x/10) mod 5

Now hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236



Chaining

When we search for an item, we apply the hash function to get to its hash index and then perform a linear search of the linked list. If the item is not in the list, then it's not in the table.



Open addressing

There is no linked list here. We have a hash table of size N containing key-value pairs, and we resolve collisions by trying a table of sequence entries until either the item is found in the table or we reach an empty location. The sequence is called a probe sequence. There are several alternatives...

Open addressing

Linear probing:

h(k), h(k) + 1, h(k) + 2, ..., h(k) + (N – 1) (all mod N)

Example: suppose h(x) = x mod 10

Hash these keys: 4, 44, 444, 6, 5

This is why you store the values ***and*** the keys.

A draw back is clustering. Adjacent clusters tend to join together to form composite clusters.

0	
1	
2	
3	
4	4
5	44
6	444
7	6
8	5
9	

Open addressing

Probing every R^{th} location:

h(k), h(k) + R, h(k) + 2R, ..., h(k) + (N – 1)R (all mod N)

To guarantee that we probe every table location, R must be relatively prime (no common factors other than 1) to N (the table size).

Example: suppose h(x) = x mod 10, R = 2

Hash these keys: 4, 14, 114, 1114, 11114

There will still be clustering, but the clusters are not consecutive locations.

0	1114
1	
2	11114
3	
4	4
5	
6	14
7	
8	114
9	

Open addressing

Quadratic probing:

$h(k), h(k) + 1^2, h(k) + 2^2, ..., h(k) + (N - 1)^2$ (all mod N)

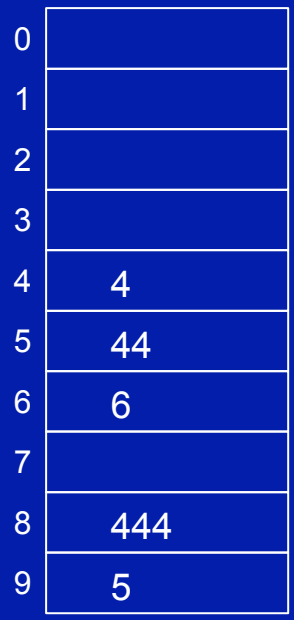
This method avoids consecutive clustering.

Drawback: $i^2 = (N - i)^2$ mod N so the probe sequence examines only half the table.

Example: suppose $h(x) = x \bmod 10$

Hash these keys: 4, 44, 444, 6, 5

Some people use:
$h(k), h(k) + 1^2, h(k) - 1^2, h(k) + 2^2, h(k) - 2^2, ...$



Open addressing

Pseudo-random probing:

$h(k), h(k) + r_1, h(k) + r_2, ..., h(k) + r_{N-1}$ (all mod N)

where r1, r1, ..., rN-1 is a sequence of previously-generated pseudo-random numbers.

Drawback: we must store the pseudo-random numbers. Why?

Open addressing

Double hashing:

Up to now, we've seen that two keys that hash to the same location have the same probe sequence. Here's a better solution, assuming the two keys are different.

$h(k)$, $h(k) + 1h_2(k)$, $h(k) + 2h_2(k)$, ..., $h(k) + (N-1)h_2(k)$ (all mod N)

...but if $h_2(k) = 0$, then set $h_2(k) = 1$

A common choice is $h_2(k) = q - (k \bmod q)$ where q is a prime < N

The hope is that if $h(x) = h(y)$, then $h_2(x) \neq h_2(y)$

Open addressing

Disadvantages of open addressing:

- Clusters can run into one another
- The hash table must be at least as large as the number of items hashed, and preferably much larger
- Deletions can be a problem. This will cause problems when searching for an existing key in a possibly long probe sequence. The solution here is to mark the space previously occupied by a key-value pair with a marker indicating that the probe sequence shouldn't stop at this now-unoccupied location. The marker is called a "tombstone". For insertion, the tombstone is considered to be an empty location.

Advantages:

- No memory is wasted on pointers

Performance of hashing

Performance depends on:

- the quality of the hash function
- the collision resolution algorithm
- the available space in the hash table

To estimate how full a table is, we calculate the load factor α:

$$\alpha = (\text{number of entries in table}) / (\text{table size})$$

For open addressing, $0 \leq \alpha \leq 1$

For chaining, $0 \leq \alpha < \infty$

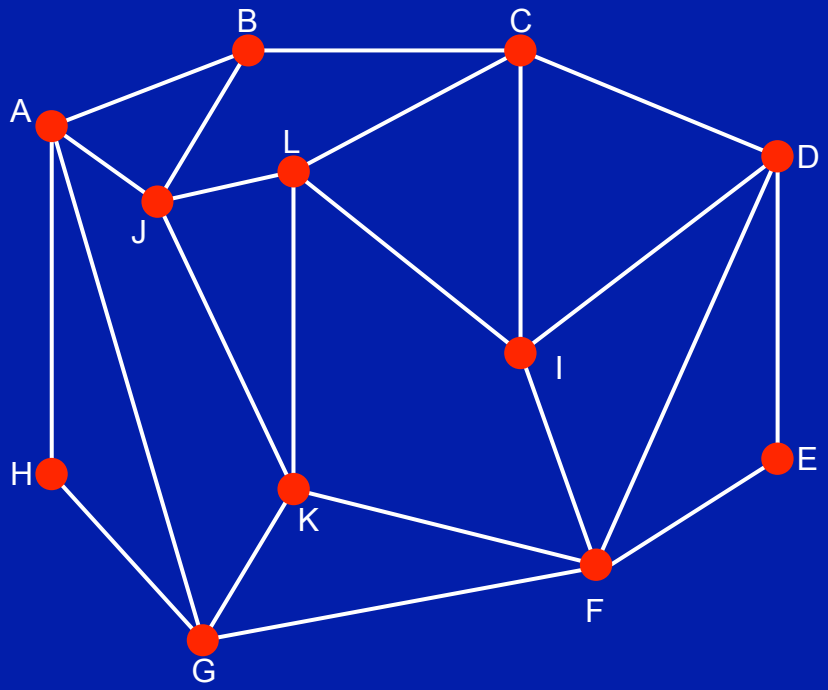
Performance of hashing

The legendary Donald Knuth has estimated the expected number of probes for a successful search as a function of the load factor α:

Linear probing:	$\frac{1}{2}(1 + 1/(1-\alpha))$
Quadratic probing or Pseudo-random probing	$(1/\alpha)\ln(1/(1-\alpha))$
Chaining	$1 + \alpha/2$

Graphs, more formally

A **graph** $G=(V,E)$ consists of:
(a) a non-empty set of **vertices** V, and
(b) a set of **edges** E between pairs of those vertices.
An edge $e \in E \subseteq V \times V$, where $e = (u,v)$ and $u,v \in V$.



G = <**V**,**E**>

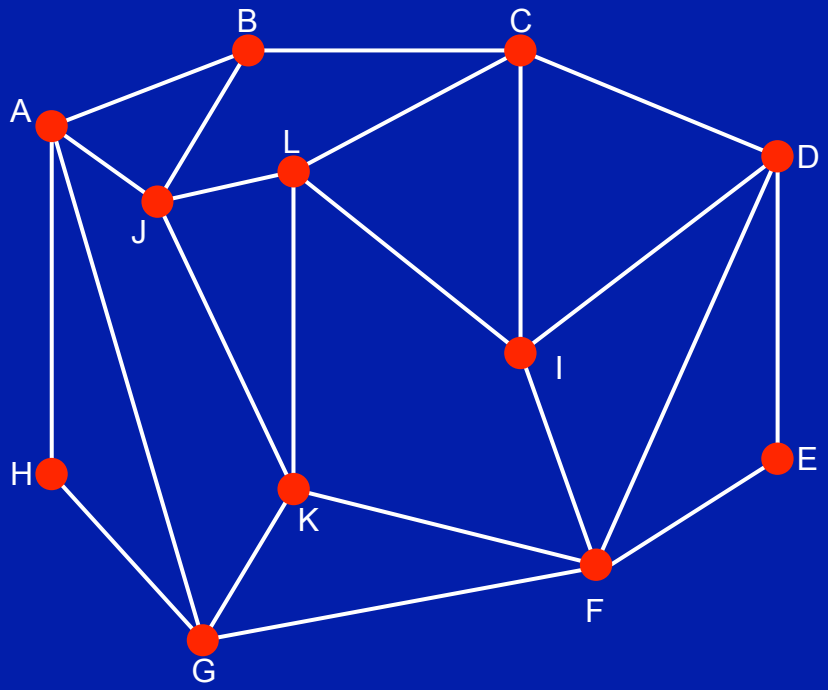
V = vertices =
{A,B,C,D,E,F,G,H,I,J,K,L}

E = edges = {(A,B),(B,C),(C,D),(D,E),
(E,F),(F,G),(G,H),(H,A),(A,J),(A,G),
(B,J),(K,F),(C,L),(C,I),(D,I),(D,F),(F,I),
(G,K),(J,L),(J,K),(K,L),(L,I)}

Graphs, more formally

Graphs may be **undirected**:

- Known as an undirected graph, or simply graph.
- $(u,v) = (v,u)$.



G = <**V**,**E**>

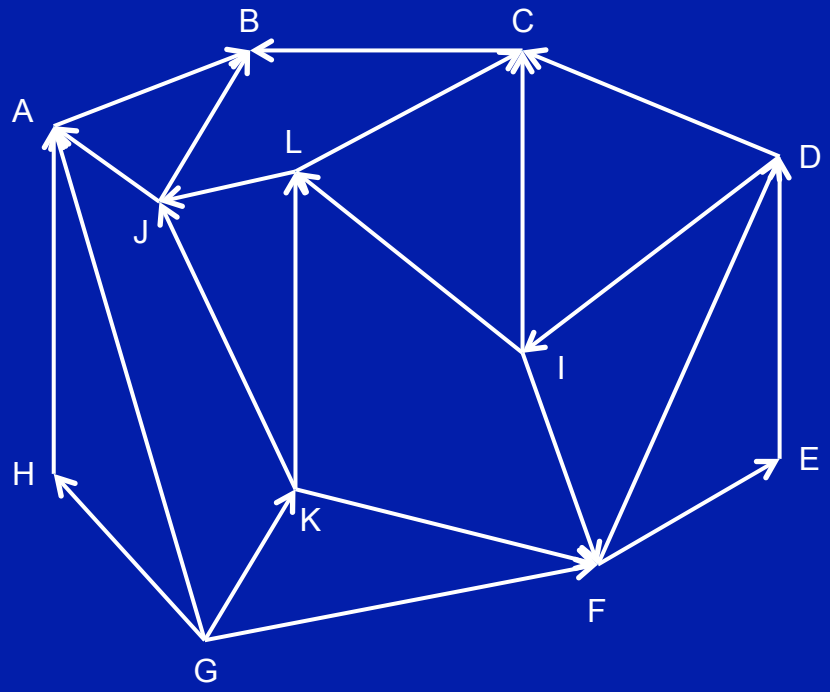
V = vertices = {A,B,C,D,E,F,G,H,I,J,K,L}

E = edges = {(A,B),(B,C),(C,D),(D,E),(E,F),(F,G),(G,H),(H,A),(A,J),(A,G),(B,J),(K,F),(C,L),(C,I),(D,I),(D,F),(F,I),(G,K),(J,L),(J,K),(K,L),(L,I)}

Graphs, more formally

Graphs may be **directed**:

- This is called a directed graph or a digraph.
- The ordered pair (u,v) implies an edge from u to v
- (red dots removed so you can see the arrows)



G = <**V**,**E**>

V = vertices = {A,B,C,D,E,F,G,H,I,J,K,L}

E = edges = {(A,B),(C,B),(D,C),(D,I),(E,D),(F,D),(F,E),(G,A),(G,F),(G,H),(G,K),(H,A),(I,C),(I,F),(I,L),(J,A),(J,B),(K,F),(K,J),(K,L),(L,C),(L,J)}

Some definitions

An edge *e* between vertices *u* and *v* is said to be incident on *u* and *v*. Edges can have weights associated with them.

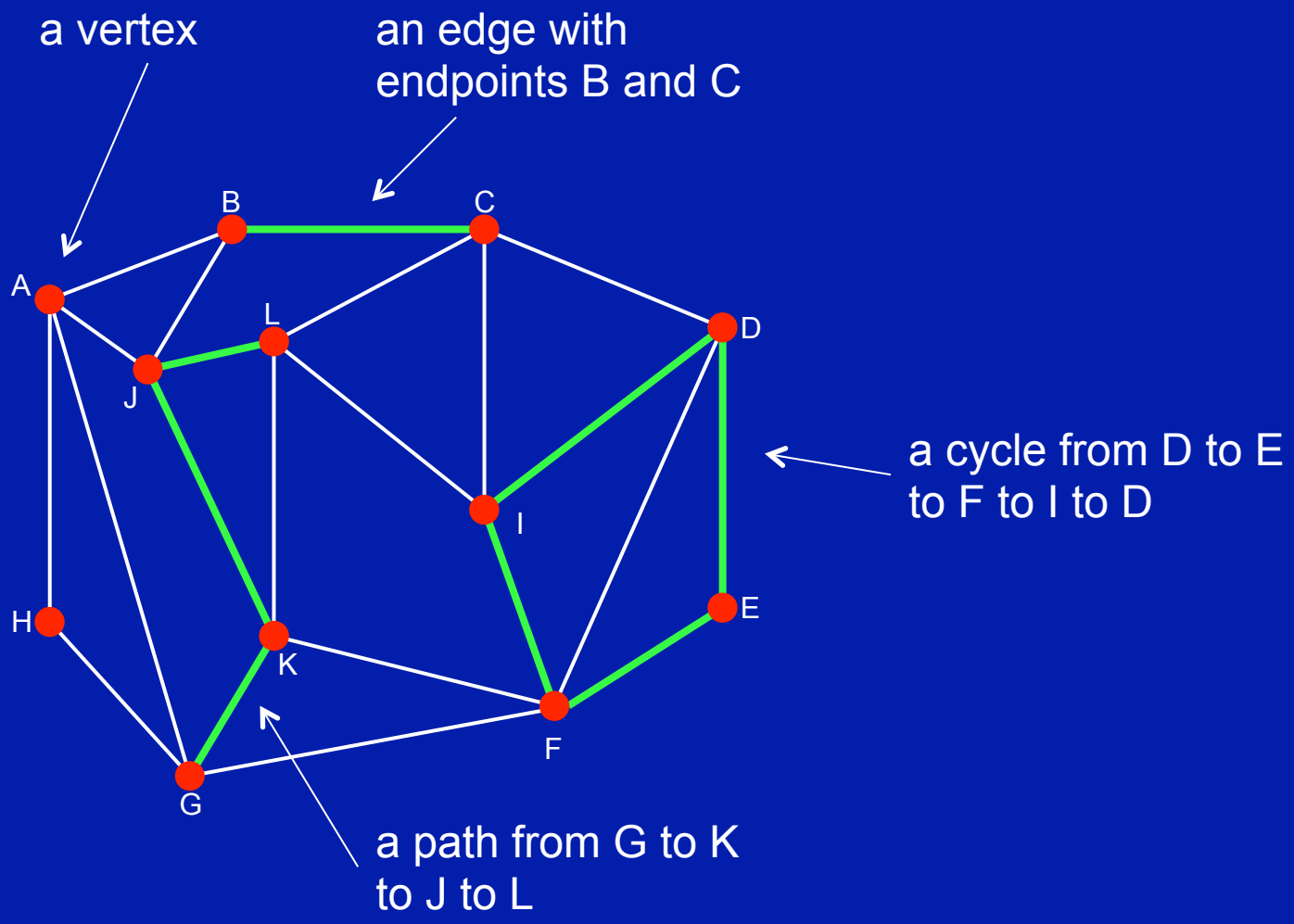
A vertex is adjacent to another if there is a single edge connecting them.

A path is a sequence of vertices in which each successive vertex is adjacent to its predecessor. A path can be directed or undirected.

If the first and last vertices are the same in a path, it is called a cycle.

A vertex that connects to itself is called a loop.

Graph anatomy

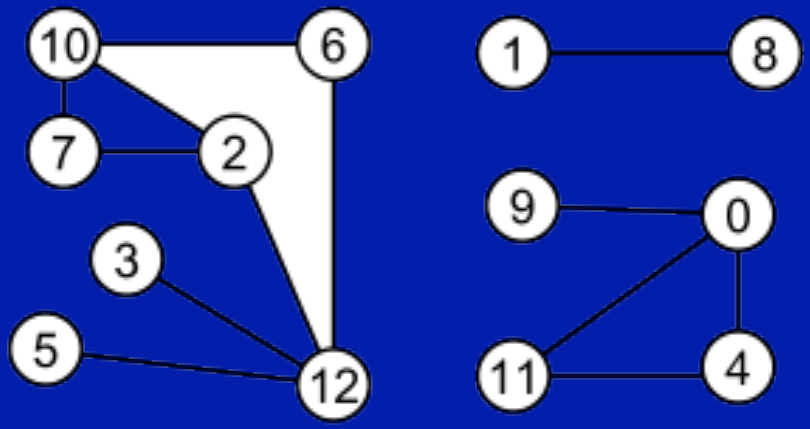


Weighted graph

- A weighted graph is a graph in which a value is associated with each of the edges.
- Weighted graphs may be either directed or undirected.

Graph isomorphism

The numbering of the vertices, and their physical arrangement are not important. The following is the same graph as the previous slide.



Concrete representation of a graph

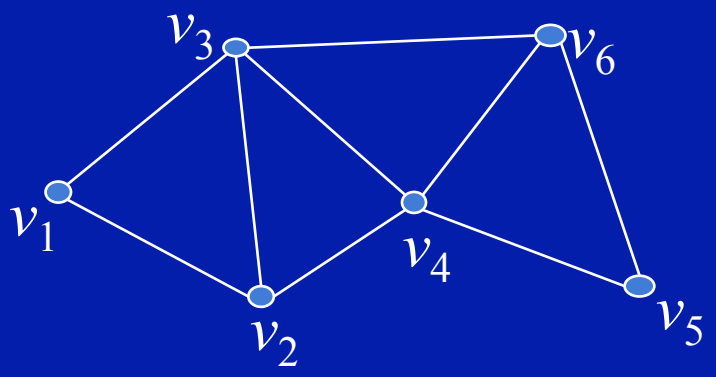
E.g. A graph with n vertices $v_1, v_2, \ldots v_n$.

The entry in row i and column j of A is the number of edges (v_i, v_j) in the graph.

The adjacency matrix of a simple graph contains only 0's and 1's.

The adjacency matrix of an undirected graph is symmetric.

For graphs with few edges, the adjacency matrix is *sparse* (i.e., contains mostly zeros).

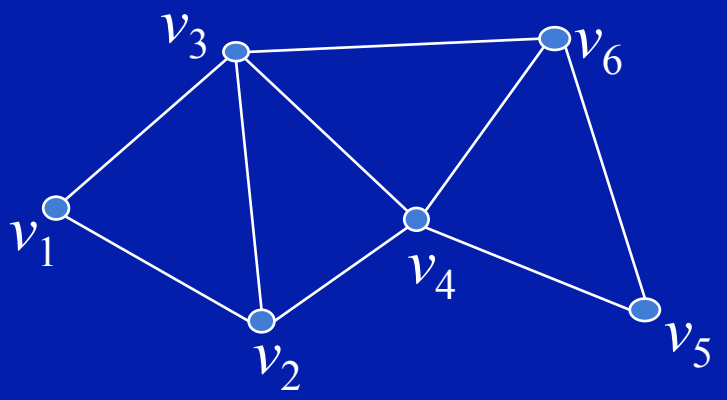


	v_1	v_2	v_3	v_4	v_5	v_6
v_1		1	1			
v_2	1		1	1		
v_3	1	1		1		1
v_4		1	1		1	1
v_5				1		1
v_6			1	1	1	

Adjacency Matrix

Adjacency list

E.g. A graph with n vertices $v_1, v_2, \ldots v_n$.



The adjacency list is never sparse (even for a graph with few edges). In other words, the space is well-used.

v_1	v_2, v_3
v_2	v_1, v_3, v_4
v_3	v_1, v_2, v_4, v_6
v_4	v_2, v_3, v_5, v_6
v_5	v_4, v_6
v_6	v_3, v_4, v_5

Adjacency List

Efficiency comparisons

In general, if the graph is dense, the adjacency matrix is better, and if the graph is sparse, the adjacency list is better. Usually, our graphs are sparse, but not always.

The math is in your textbook, but intuitively, a sparse graph will lead to a sparse matrix (few 1s, lots of 0s). The algorithm has to process every location in the matrix, whether it indicates an edge there or not, and that has an undesirable impact on processing time. Those entries don't show up in the list, however, so they will have no effect on processing time in the adjacency list implementation.

Graph traversal

Graph algorithms typically involve visiting each vertex in systematic order. The two most common traversal algorithms are breadth-first and depth-first.

Although these are traversals, the traversals are usually employed to find something in the graph (e.g., a vertex/node, or more likely a path to that vertex/node), so they're more commonly called breadth-first search and depth-first search.

Breadth-first

- Starting at a source vertex s
- Systematically explore the edges to "discover" every vertex reachable from s.
- Produces a "breadth-first tree"
  - Root of s
  - Contains all vertices reachable from *s*
  - Path from *s* to *v* is the shortest path

Breadth-first algorithm

Take a start vertex, mark it identified (colour it green), and place it into a queue.

While the queue is not empty

- Take a vertex, u, out of the queue (Begin visiting u)
- For all vertices v, adjacent to u,
  - If v has not been identified or visited
    - Mark it identified (colour it green)
    - Place it into the queue
    - (Add edge u, v to the Breadth First Search Tree)
- We are now done visiting u (colour it orange)

Depth-first

- Starting at a source vertex s
- Follow a simple path discovering new vertices until you cannot find a new vertex
- Back-up until you can start finding new vertices

Depth-first algorithm

Start at vertex *u*. Mark it visited (colour green) and put it in discovery order list.

For each vertex *v* adjacent to u

    If *v* has not been visited

        Insert *u*, *v* into DFS tree

        Recursively apply this algorithm starting at v

Mark *u* finished (color it orange) and put it in finish order list.

This demo will ignore some of these details. We'll just do what breadth-first did, but we'll use a stack instead of a queue.

Counting

In CS, we often encounter situations where we want to count the number of possible outcomes of an event. For example, we may wish to determine: the number of possible paths to follow in a directed network (graph), the number of possible 5-8 character passwords, etc.

Multiplication Principle (Product Rule)

If an operation consists of t steps and:

Step 1 can be performed in n_1 ways,

Step 2 can be performed in n_2 ways,

…

Step t can be performed in n_t ways, then the entire operation can be performed in $n_1 n_2 \ldots n_t$ different ways.

Example: How many postal codes begin with the letter V and end with the digit 4?

V					4	
1	10	26	10	26	1	$10^2(26^2)$

Addition Principle (Sum Rule)

If $X_1, X_2, \ldots, X_t$ are pairwise disjoint sets (i.e., $X_i \cap X_j = \varnothing$ $\forall i \; \forall j$ s.t. $i, j \in \{1, 2, \ldots, t\}$, $i \neq j$), then the number of ways to select an element from any of X_1 or X_2 or … or X_t is:

$$| X_1 | + | X_2 | + \ldots + | X_t |$$

disjoint means they have no elements in common

So in English, this means that the size of a union of a family of mutually disjoint sets is the sum of the sizes of the sets

Inclusion-Exclusion Principle

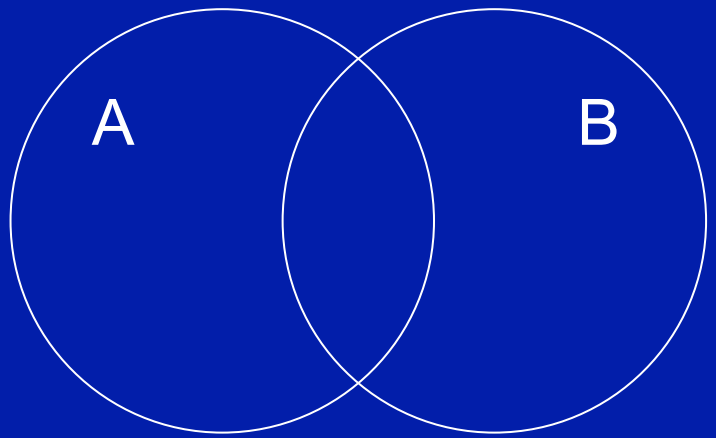
If an object can be found in 2 or more sets at the same time, then we cannot use the addition principle. Why not?

We could over count

If A and B are sets, then the total number of elements in *either* A or B is given by:

$$|A \cup B| = |A| + |B| - |A \cap B|$$

Example using a Venn Diagram:



Example: How many 8-bit strings either start with “1” or end with “00”? (assume one or the other but not both)

Example: How many 8-bit strings either start with “1” or end with “00”? (assume one or the other but not both)

There are 2^7 choices for strings that start with a 1.

There are 2^6 choices for strings that end with a 00.

...but these groups both include strings that start with a 1 AND end with a 00.

Since there are 2^5 choices for strings that start with a 1 and end with a 00...

...we remove the extra counted strings:

$$2^7 + 2^6 - 2^5 - 2^5$$

Permutations & Combinations

How many different outcomes are there if you choose *r* balls from an urn containing *n* different balls?

Example: $r = 3$ Urn = {**A,B,C,D**}	Order matters **ABC ≠ CAB**	Order doesn't **ABC = CAB**
Repetition *not* OK **e.g., AAB** is *not* counted	**ABC, ABD, ACB, ACD, ADB, ADC, BAC, BAD, BCA, BCD, BDA, BDC, CAB, CAD, CBA, CBD, CDA, CDB, DAB, DAC, DBA, DBC, DCA, DCB** *r*-permutations	**ABC, ABD, ACD, BCD** *r*-combinations
Repetition OK (ball is thrown back in urn after being picked) e.g., **AAB** is counted	**AAA, AAB, AAC, AAD, ABA, ABB, ABC, ABD, ...** *r*-permutations w/ repetitions	**AAA, AAB, AAC, AAD, ABB, ABC, ABD, ACC, ACD, ADD, BBB, BBC, BBD, BCC, BCD, BDD, CCC, CCD, CDD, DDD** *r*-combinations w/ repetitions

(1) *r*-Permutations
(Order matters, and repetition is not allowed.)

An *r*-permutation of *n* distinct elements $x_1, x_2, \ldots, x_n$ is an ordering of an *r*-element subset of $\{x_1, x_2, \ldots, x_n\}$. The number of *r*-permutations is:

$$P(n, r) = n^{(r)} = n! / (n - r)!$$

Derivation:

We need to divide out the effect of the permutations of the elements that are *not* in our chosen set.

n(n-1)(n-2)...(n-r+1)(n-r)(n-r-1)...(1) / (n-r)(n-r-1)...(1)

which simplifies (divide out common elements):

n(n-1)(n-2)...(n-r+1)

(2) *r*-Combinations (Order doesn't matter, and repetition is *not* allowed.)

An *r*-combination of *n* distinct elements $x_1, x_2, \ldots, x_n$ is an *r*-element subset of $\{x_1, x_2, \ldots, x_n\}$. The number of *r*-combinations is:

$$C(n, r) = \binom{n}{r} = n! / (n - r)!\, r!$$

Example 4: A donut shop has 10 kinds of donuts. In how many ways can 6 distinct kinds of donuts be selected?

Example 5: Show how to derive a relationship between *r*-permutations and *r*-combinations.

(3) *r*-Permutations with Repetition
(Generalized *r*-Permutations)

Here, *order matters*, and *repetition is allowed*.

Suppose we have a set of n distinct elements $x_1, x_2, \ldots, x_n$ and we select a sequence of r elements, allowing repeats. How many different sequences are possible?

$$\mathbf{n^r}$$

(4) *r*-Combinations with Repetition (Generalized r-Combinations)

Here, order *doesn't* matter, and *repetition is allowed*.

Suppose we have a set of *n* distinct elements x_1, x_2, …, x_n. The number of unordered *r*-element selections from this set, with repetition allowed, is:

(r + n - 1) choose (n-1)

or (equivalently)

(r + n - 1) choose r

Summary of Formulas

Order Matters & Repetition is *not* Allowed $n^{(r)}$	Order does *not* Matter & Repetition is *not* Allowed $\begin{bmatrix} n \\ r \end{bmatrix}$
Order Matters & Repetition is Allowed n^r	Order does *not* Matter & Repetition is Allowed $\begin{bmatrix} r+n-1 \\ n-1 \end{bmatrix}$

Note: the square brackets are really supposed to be parentheses. PowerPoint can be frustrating at times.

And that's all there is...

...to Basic Data Structures and Algorithms.

The final exam will be comprehensive, but will lean toward post-midterm material.