Algorithms & Data Structures

David Vernon

Course Content

- Introduction to Complexity of Algorithms
  - Performance of algorithms
  - Time and space tradeoff
  - Worst case and average case performance
  - The big O notation
  - Example calculations of complexity
- Complexity and Intractability
  - NP Completeness and Approximation Algorithms

Course Content

- Simple Searching Algorithms
  - Linear Search
  - Binary Search
- Simple Sorting Algorithms
  - Bubblesort
  - Quicksort

Course Content

- Abstract Data Types (ADTs)
- Lists, Stacks, and Queues
  - ADT specification
  - Array implementation
  - Linked-list implementation

Course Content

- Trees
  - Binary Trees
  - Binary Search Trees
  - Traversals
  - Applications of Trees
    - » Huffman Coding
  - Height-balanced Trees
    - » AVL Trees
    - » Red-Black Trees

Introduction to Analysis of Algorithms & Complexity Theory

Complexity

- Analysis of complexity of programs
  - Time complexity
  - Space complexity
  - Big-Oh Notation
- Introduction to complexity theory
  - P, NP, and NP-Complete classes of algorithm

Complexity

- Suppose there is an assignment statement

  x := x +1

  in your program.
- We'd like to determine:
  - The time a single execution would take
  - The number of times it is executed

    Frequency Count

Complexity

- Product of time and frequency is the total time taken
- Frequency count will vary from data set to data set

Complexity

- Since the execution time will be very machine dependent (and compiler dependent), we neglect it and concentrate on the frequency count
- Consider the following three examples:

Complexity

```
Program 1

x := x + 1
```

```
Program 2

FOR i := 1 to n
DO
   x := x + 1
END
```

```
Program 3

FOR i := 1 to n
DO
   FOR j := 1 to n
   DO
      x := x + 1
   END
END
```

Complexity

- Program 1:
  - statement is not contained in a loop (implicitly or explicitly)
  - Frequency count is 1
- Program 2
  - statement is executed *n* times
- Program 3
  - statement is executed n^2 times

Complexity

- 1, n, and n^2 are said to be different and increasing orders of magnitude (e.g. let $n = 10$)
- We are chiefly interested in determining the order of magnitude of an algorithm

Complexity

- Let's look at an algorithm to print the n^{th} term of the Fibonnaci sequence
- 0 1 1 2 3 5 8 13 21 34 …
- $t_n = t_{n-1} + t_{n-2}$
- $t_0 = 0$
- $t_1 = 1$

Complexity

```
1     procedure fibonacci
2        read(n)
3           if n<0
4              then print(error)
5              else if n=0
6                 then print(0)
7                 else if n=1
8                    then print(1)
9                    else
10                     fnm2 := 0;
11                      fnm1 := 1;
12                      FOR i := 2 to n DO
13                         fn := fnm1 + fnm2;
14                         fnm2 := fnm1;
15                         fnm1 := fn
16                      end
17                      print(fn);
```

step	n<0
1	1
2	1
3	1
4	1
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Complexity

```
1     procedure fibonacci {print nth term}
2        read(n)
3           if n<0
4              then print(error)
5              else if n=0
6                 then print(0)
7                 else if n=1
8                    then print(1)
9                    else
10                     fnm2 := 0;
11                      fnm1 := 1;
12                      FOR i := 2 to n DO
13                         fn := fnm1 + fnm2;
14                         fnm2 := fnm1;
15                         fnm1 := fn
16                      end
17                      print(fn);
```

step	n=0
1	1
2	1
3	1
4	0
5	1
6	1
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Complexity

```
1     procedure fibonacci
2        read(n)
3           if n<0
4              then print(error)
5              else if n=0
6                 then print(0)
7                 else if n=1
8                    then print(1)
9                    else
10                     fnm2 := 0;
11                      fnm1 := 1;
12                      FOR i := 2 to n DO
13                         fn := fnm1 + fnm2;
14                         fnm2 := fnm1;
15                         fnm1 := fn
16                      end
17                      print(fn);
```

step	n=1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	1
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Complexity

```
1     procedure fibonacci
2        read(n)
3           if n<0
4              then print(error)
5              else if n=0
6                 then print(0)
7                 else if n=1
8                    then print(1)
9                    else
10                     fnm2 := 0;
11                      fnm1 := 1;
12                      FOR i := 2 to n DO
13                         fn := fnm1 + fnm2;
14                         fnm2 := fnm1;
15                         fnm1 := fn
16                      end
17                      print(fn);
```

step	n>1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	0
9	1
10	1
11	1
12	n
13	n-1
14	n-1
15	n-1
16	n-1
17	1

Complexity

step	n<0	n=0	n=1	n>1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	0	0	0
5	0	1	1	1
6	0	1	0	0
7	0	0	1	1
8	0	0	1	0
9	0	0	0	1
10	0	0	0	1
11	0	0	0	1
12	0	0	0	n
13	0	0	0	n-1
14	0	0	0	n-1
15	0	0	0	n-1
16	0	0	0	n-1
17	0	0	0	1

Complexity

- The cases where $n<0$, $n=0$, $n=1$ are not particularly instructive or interesting
- In the case where $n>1$, we have the total statement frequency of:

$$9 + n + 4(n-1) = 5n + 5$$

Complexity

- $9 + n + 4(n\text{-}1) = 5n + 5$

- We write this as $O(n)$, ignoring the constants
- It means that the order of magnitude is proportional to n

Complexity

- The notation $f(n) = O(g(n))$ has a precise mathematical definition
- Read $f(n) = O(g(n))$ as
  f of *n* equals big-oh of *g* of *n*
- Definition:
  $f(n) = O(g(n))$ iff there exist two constants *c* and n_0 such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$

Complexity

- $f(n)$ will normally represent the computing time of some algorithm
  - Time complexity $T(n)$
- $f(n)$ can also represent the amount of memory an algorithm will need to run
  - Space complexity $S(n)$

Complexity

- If an algorithm has a time complexity of $O(g(n))$ it means that its execution will take no longer than a constant times $g(n)$
- n is typically the size of the data set

Complexity

- $O(1)$ Constant (computing time)
- $O(n)$ Linear (computing time)
- $O(n^2)$ Quadratic (computing time)
- $O(n^3)$ Cubic (computing time)
- $O(2^n)$ Exponential (computing time)
- $O(\log n)$ is faster than $O(n)$ for sufficiently large n
- $O(n \log n)$ is faster than $O(n^2)$ for sufficiently large n

Complexity

- *Let's look at the way these functions grow with n ….*

Complexity

n	O(1)	O(log2(n))	O(n)	O(nlog2(n)	O(n^2)	O(n^3)	O(n^4)	O(2^n)	O(n^n)
1	7	0.0	1	0.0	1	1	1	2	1
2	7	1.0	2	2.0	4	8	16	4	4
3	7	1.6	3	4.8	9	27	81	8	27
4	7	2.0	4	8.0	16	64	256	16	256
5	7	2.3	5	11.6	25	125	625	32	3125
6	7	2.6	6	15.5	36	216	1296	64	46656
7	7	2.8	7	19.7	49	343	2401	128	823543
8	7	3.0	8	24.0	64	512	4096	256	16777216
9	7	3.2	9	28.5	81	729	6561	512	3.87E+08
10	7	3.3	10	33.2	100	1000	10000	1024	1E+10
11	7	3.5	11	38.1	121	1331	14641	2048	2.85E+11
12	7	3.6	12	43.0	144	1728	20736	4096	8.92E+12
13	7	3.7	13	48.1	169	2197	28561	8192	3.03E+14
14	7	3.8	14	53.3	196	2744	38416	16384	1.11E+16
15	7	3.9	15	58.6	225	3375	50625	32768	4.38E+17
16	7	4.0	16	64.0	256	4096	65536	65536	1.84E+19
17	7	4.1	17	69.5	289	4913	83521	131072	8.27E+20
18	7	4.2	18	75.1	324	5832	104976	262144	3.93E+22
19	7	4.2	19	80.7	361	6859	130321	524288	1.98E+24
20	7	4.3	20	86.4	400	8000	160000	1048576	1.05E+26

Double click to activate spreadsheet and graph complexity function

Complexity

- Arithmetic of Big Oh notation
- if

$T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$

then

$T_1(n) + T_2(n) = O(max(f(n),g(n))$

Complexity

- if $f(n) <= g(n)$

  then

  $O(f(n) + g(n)) = O(g(n))$

Complexity

- if

  $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$

  then

  $T_1(n)\ T_2(n) = O(f(n)g(n))$

Complexity

- Rules for computing the time complexity
  - the complexity of each read, write, and assignment statement can be take as $O(1)$
  - the complexity of a sequence of statements is determined by the summation rule
  - the complexity of an if statement is the complexity of the executed statements, plus the time for evaluating the condition

Complexity

- Rules for computing the time complexity
  - the complexity of an if-then-else statement is the time for evaluating the condition plus the larger of the complexities of the then and else clauses
  - the complexity of a loop is the sum, over all the times around the loop, of the complexity of the body and the complexity of the termination condition

Complexity

- Given an algorithm, we analyse the frequency count of each statement and total the sum.
- This may give a polynomial $P(n)$:

$$P(n) = c_k\, n^k + c_{k-1}\, n^{k-1} + \ldots + c_1\, n + c_0$$

where the c_i are constants, c_k are non-zero, and n is a parameter

Complexity

- Using big-oh notation, we have:

  $P(n) = O(n^k)$

- On the other hand, if any step is executed 2^n times or more we have:

  $c\ 2^n + P(n) = O(2^n)$

Complexity

- What about computing the complexity of a recursive algorithm?
- In general, this is more difficult
- The basic technique
  - identify a recurrence relation implicit in the recursion $T(n) = f(T(k))$, $k \in \{1, 2, \ldots, n\text{-}1\}$
  - solve the recurrence relation by finding an expression for $T(n)$ in term which do not involve $T(k)$

Complexity

- Example: compute factorial *n* (*n*!)

```
int factorial(int n)
{
   int factorial_value;

   factorial_value = 0;

   /* compute factorial value recursively */

   if (n <= 1) {
      factorial_value = 1;
   }
   else {
      factorial_value = n * factorial(n-1);
   }
   return (factorial_value);
}
```

Complexity

- Let the time complexity of the function be $T(n)$
- which is what we want!
- Now, let's try to analyse the algorithm

Complexity

```
                                                    n>1
int factorial(int n)
{
   int factorial_value;                             1

   factorial_value = 0;                             1

   if (n <= 1) {                                    1
      factorial_value = 1;                          0
   }
   else {                                           1
      factorial_value = n * factorial(n-1);         T(n-1)
   }
   return (factorial_value);                        1
}
```

Complexity

- $T(n) = 5 + T(n-1)$
- $T(n) = c + T(n-1)$
- $T(n-1) = c + T(n-2)$
- $T(n) = c + c + T(n-2)$
  $= 2c + T(n-2)$
- $T(n-2) = c + T(n-3)$
- $T(n) = 2c + c + T(n-3)$
  $= 3c + T(n-3)$
- $T(n) = ic + T(n-i)$

Complexity

- $T(n) = ic + T(n-i)$
- Finally, when $i = n-1$
- $T(n) = (n-1)c + T(n-(n-1))$
  $= (n-1)c + T(1)$
  $= (n-1)c + d$
- *Hence,* $T(n) = O(n)$

Complexity

- Space Complexity
  - Compute the space complexity of an algorithm by analysing the storage requirements (as a function on the input size) in the same way

Complexity

- Space Complexity
  - For example
    - » if you read a stream of *n* characters
    - » and only ever store a constant number of them,
    - » then it has space complexity *O*(1)

Complexity

- Space Complexity
  - For example
    - » if you read a stream of n records
    - » and store all of them,
    - » then it has space complexity O(n)

Complexity

- Space Complexity
  - For example
    - » if you read a stream of n records
    - » and store all of them,
    - » and each record causes the creation of (a constant number) of other records,
    - » then it still has space complexity O(n)

Complexity

- Space Complexity
  - For example
    - » if you read a stream of n records
    - » and store all of them,
    - » and each record causes the creation of a number of other records (and the number is proportional to the size of the data set n)
    - » then it has space complexity $O(n^2)$

Complexity

- Time vs Space Complexity
  - In general, we can often decrease the time complexity but this will involve an increase in the space complexity
  - and *vice versa* (decrease space, increase time)
  - This is the time-space tradeoff

Complexity

- Time vs Space Complexity
  - For example, the average time complexity of an iterative sort (e.g. bubble sort) is $O(n^2)$
  - but we can do better: the average time complexity of the Quicksort is $O(n \log n)$
  - But the Quicksort is recursive and the recursion causes a increase in memory requirements (*i.e.* an increase in space complexity)

Complexity

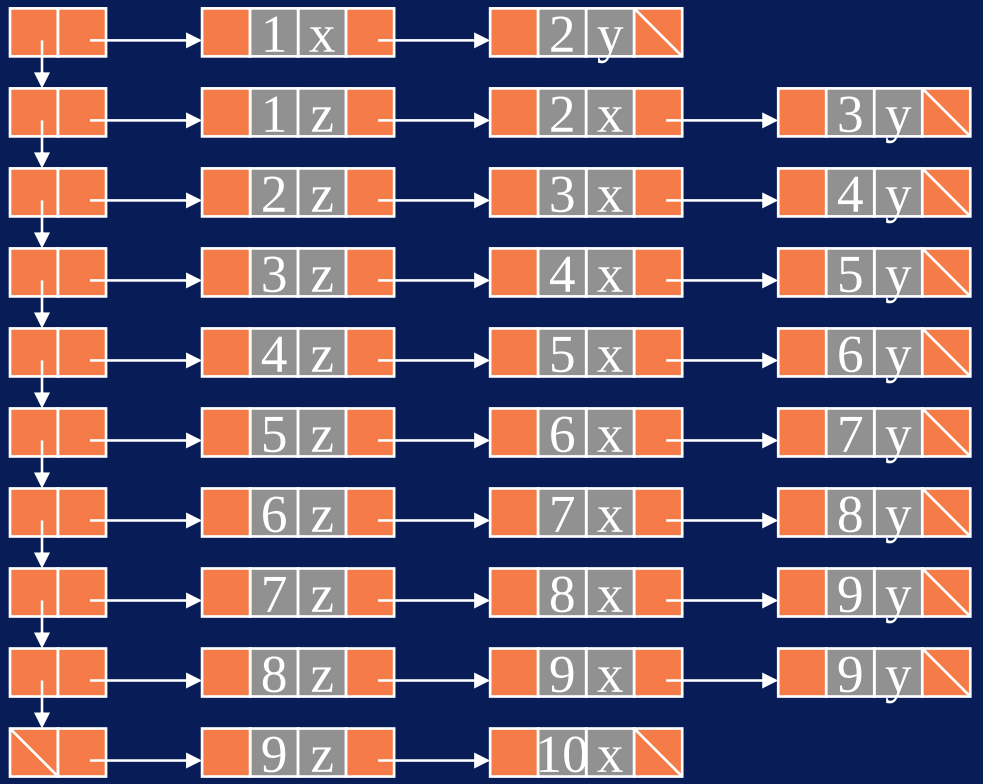
- Time vs Space Complexity
  - For example, the space complexity of 2-D matrix is O(n^2)
  - but if the matrix is sparse we can do better: we can represent the matrix as a 2-D linked list and often reduce the space complexity to O(n)
  - But the time taken to access each element will rise (*i.e.* the time complexity will rise)

Complexity

x	y	0	0	0	0	0	0	0	0
z	x	y	0	0	0	0	0	0	0
0	z	x	y	0	0	0	0	0	0
0	0	z	x	y	0	0	0	0	0
0	0	0	z	x	y	0	0	0	0
0	0	0	0	z	x	y	0	0	0
0	0	0	0	0	z	x	y	0	0
0	0	0	0	0	0	z	x	y	0
0	0	0	0	0	0	0	z	x	y
0	0	0	0	0	0	0	0	z	x

nxn matrix:

$O(n^2)$ space complexity



2x(2 + 4 + 4) + (n-2)x(2 + 4 + 4 + 4)
= 20 + 14n - 28 = 14n - 8:

O(n) space complexity

Complexity

- Order of space complexity for the matrix representation of the banded matrix is $O(n^2)$ >>
  Order of space complexity for the linked list representation $O(n)$
- However, the matrix implementation will sometimes be more effective:

Complexity

- $n^2 <= 14n - 8$
- $n^2 - 14n + 8 <= 0$
- $n = \pm 13$ is the cutoff at which the list representation is more efficient in terms of storage space.
- Typically, in real engineering problems, *n* can be much greater than 100 and the saving is very significant

Complexity

n	O(1)	O(log2(n))	O(n)	O(nlog2(n)	O(n^2)	O(n^3)	O(n^4)	O(2^n)	O(n^n)
1	7	0.0	1	0.0	1	1	1	2	1
2	7	1.0	2	2.0	4	8	16	4	4
3	7	1.6	3	4.8	9	27	81	8	27
4	7	2.0	4	8.0	16	64	256	16	256
5	7	2.3	5	11.6	25	125	625	32	3125
6	7	2.6	6	15.5	36	216	1296	64	46656
7	7	2.8	7	19.7	49	343	2401	128	823543
8	7	3.0	8	24.0	64	512	4096	256	16777216
9	7	3.2	9	28.5	81	729	6561	512	3.87E+08
10	7	3.3	10	33.2	100	1000	10000	1024	1E+10
11	7	3.5	11	38.1	121	1331	14641	2048	2.85E+11
12	7	3.6	12	43.0	144	1728	20736	4096	8.92E+12
13	7	3.7	13	48.1	169	2197	28561	8192	3.03E+14
14	7	3.8	14	53.3	196	2744	38416	16384	1.11E+16
15	7	3.9	15	58.6	225	3375	50625	32768	4.38E+17
16	7	4.0	16	64.0	256	4096	65536	65536	1.84E+19
17	7	4.1	17	69.5	289	4913	83521	131072	8.27E+20
18	7	4.2	18	75.1	324	5832	104976	262144	3.93E+22
19	7	4.2	19	80.7	361	6859	130321	524288	1.98E+24
20	7	4.3	20	86.4	400	8000	160000	1048576	1.05E+26

Double click to activate spreadsheet and graph complexity function

Complexity

- *Worst-case complexity and Average-case complexity*
  - so far we have looked only at worst-case complexity (i.e. we have developed an upper-bound on complexity)
  - however, there are times when we are more interested in the average-case complexity (especially it differs significantly)

Complexity

- *Worst-case complexity and Average-case complexity*
  - for example, the Quicksort algorithm has $T(n) = O(n^2)$, worst case (for inversely sorted data)
  - $T(n) = O(n \log_2 n)$, average case (for randomly ordered data)

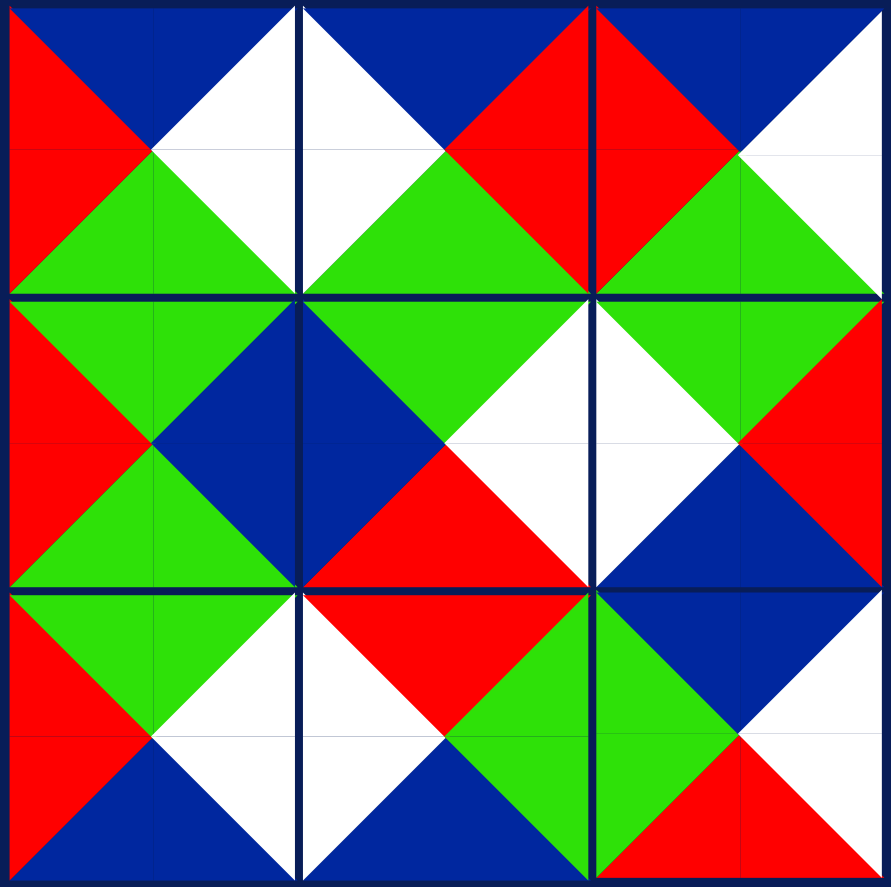
Complexity

n	O(1)	O(log2(n))	O(n)	O(nlog2(n)	O(n^2)	O(n^3)	O(n^4)	O(2^n)	O(n^n)
1	7	0.0	1	0.0	1	1	1	2	1
2	7	1.0	2	2.0	4	8	16	4	4
3	7	1.6	3	4.8	9	27	81	8	27
4	7	2.0	4	8.0	16	64	256	16	256
5	7	2.3	5	11.6	25	125	625	32	3125
6	7	2.6	6	15.5	36	216	1296	64	46656
7	7	2.8	7	19.7	49	343	2401	128	823543
8	7	3.0	8	24.0	64	512	4096	256	16777216
9	7	3.2	9	28.5	81	729	6561	512	3.87E+08
10	7	3.3	10	33.2	100	1000	10000	1024	1E+10
11	7	3.5	11	38.1	121	1331	14641	2048	2.85E+11
12	7	3.6	12	43.0	144	1728	20736	4096	8.92E+12
13	7	3.7	13	48.1	169	2197	28561	8192	3.03E+14
14	7	3.8	14	53.3	196	2744	38416	16384	1.11E+16
15	7	3.9	15	58.6	225	3375	50625	32768	4.38E+17
16	7	4.0	16	64.0	256	4096	65536	65536	1.84E+19
17	7	4.1	17	69.5	289	4913	83521	131072	8.27E+20
18	7	4.2	18	75.1	324	5832	104976	262144	3.93E+22
19	7	4.2	19	80.7	361	6859	130321	524288	1.98E+24
20	7	4.3	20	86.4	400	8000	160000	1048576	1.05E+26
20	7	4.3	20	86.4	400				
20	7	4.3	20	86.4	400				
20	7	4.3	20	86.4	400				
20	7	4.3	20	86.4	400				

Double click to activate spreadsheet and graph complexity function

Complexity and Intractability

- The complexity of some algorithms is such that, in effect, they become intractable
- Consider the monkey puzzle problem:
  - given nine square cards whose sides are imprinted with the upper and lower halves of coloured figures
  - the objective is to arrange the cards in a 5x5 square such that halves match and colours are identical wherever edges meet



Complexity and Intractability

- Assume *n*, the number of cards, is 25
- The size of the final square is 5x5

Complexity and Intractability

- Brute force solution:
  - Go through all possible arrangements of the cards
  - pick a card and place it - there are 25 possibilities for the first placement
  - pick the next card and place it - there are 24 possibilities,
  - Pick the next card, there are 23 possibilities ...

Complexity and Intractability

- there are 25x24x23x22x.....x2x1 possible arrangements
- That is, there are factorial 25 possible arrangements (25!)
- 25! contains 26 digits
- If we make 1000000 arrangements per second, the algorithm will take 490 000 000 000 years to complete

Complexity and Intractability

- The order of complexity of this algorithm is $O(n!)$
- $n!$ grows at a rate which is orders of magnitude larger than the growth rate of the other functions we mentioned before

Complexity and Intractability

- Other functions exist that grow even faster, e.g. n^n
- Even functions like 2^n exhibit unacceptable sizes even for modest values of *n*

Complexity and Intractability

- We classify functions as ‘good’ and ‘bad’
- Polynomial functions are good
- Super-polynomial (or exponential) functions are bad

Complexity and Intractability

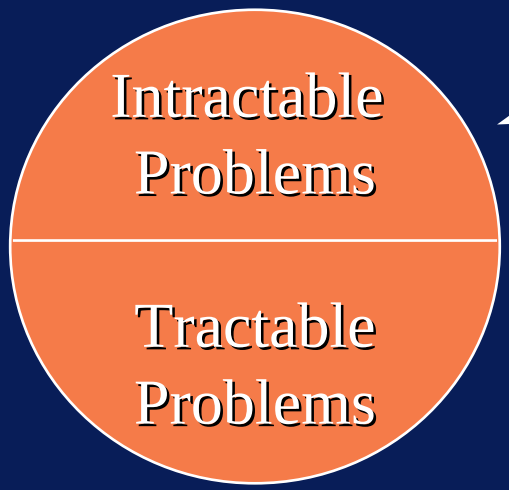
- A polynomial function is one that is bounded from above by some function n^k for some fixed value of k (*i.e.* $k \neq f(n)$)
- An exponential function is one that is bounded from above by some function k^n for some fixed value of k (*i.e.* $k \neq f(n)$)
- (Strictly speaking n^n is not exponential but super-exponential)

Complexity and Intractability

- Polynomial-time algorithm
  - Order-of-magnitude time performance bounded from above by a polynomial function of *n*
  - Reasonable algorithm
- Super-polynomial, exponential, time algorithm
  - Order-of-magnitude time performance bounded from above by a super-polynomial, exponential, function of *n*
  - Unreasonable algorithm

Complexity and Intractability

- Tractable problem
  - admits a polynomial-time or reasonable solution
- Intractable problem
  - admits only an exponential or unreasonable solution



Problems not admitting reasonable algorithms

Problems admitting reasonable (polynomial-time) algorithms

Complexity and Intractability

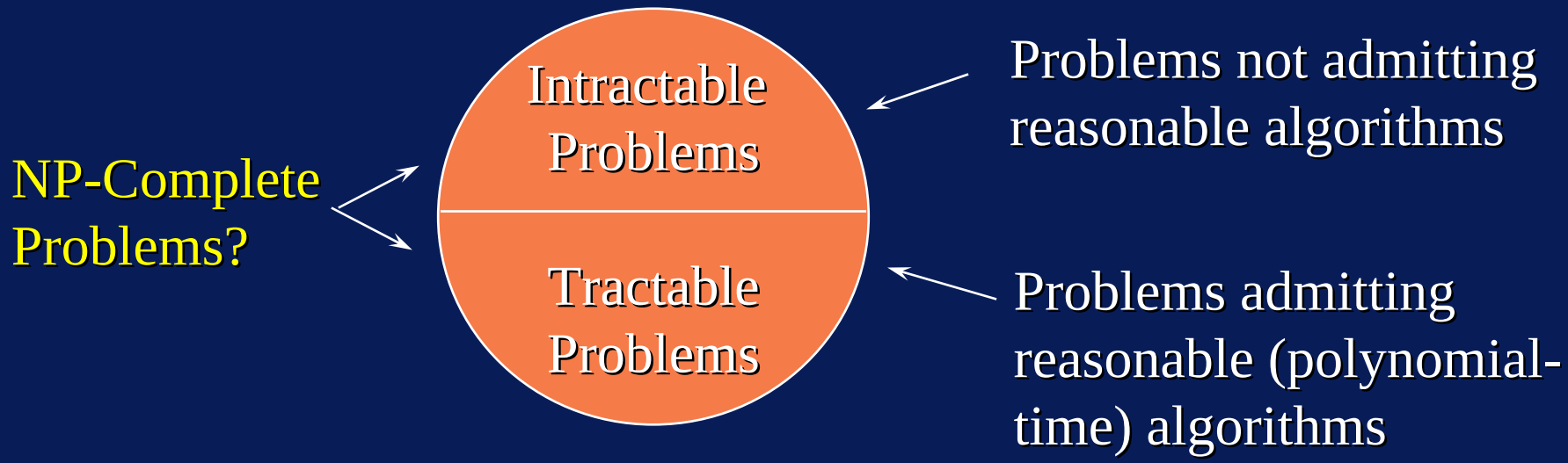
- There are many (approx. 1000) important and diverse problems which exhibit the same properties as the monkey puzzle problem (*e.g.* TSP)
- All admit unreasonable, exponential-time, solutions
- None are known to admit reasonable ones

Complexity and Intractability

- But no-one has been able to prove that any of then REQUIRE super-polynomial time
- Best known lower-bounds are O(n)

Complexity and Intractability

- This class of problems are known as NP-Complete
- Lower bounds are linear and upper bounds are exponential



Complexity and Intractability

- Examples of NP-Complete Problems
  - 2-D arrangments
  - Path-finding (e.g. travelling salesman TSP; Hamiltonian)
  - Scheduling and matching (e.g. time-tabling)
  - Determining logical truth in the propositional calculus
  - Colouring maps and graphs

Complexity and Intractability

- All NP-Complete problems seem to require construction of partial solutions (and then backtracking when we find they are wrong) in the development of the final solution
- However, if we could ‘guess’ at each point in the construction which partial solutions were to lead to the ‘right’ answer then we could avoid the construction of these partial solutions and construct only the correct solution

Complexity and Intractability

- This approach would allow
  - a polynomial-time solution
  - but it would be non-deterministic
  - since it requires some guessing
- NP - Nondeterministic Polynomial
- NP-Complete problems admit
  - Unreasonable exponential time solution
  - Reasonable non-deterministic polynomial time solutions

Complexity and Intractability

- Important property of NP-Compete problems
  - Either all NP-Complete problems are tractable or none of them are!
  - If there exists a polynomial-time algorithm for any single NP-Complete problem, then there would be necessarily a polynomial-time algorithm for all NP-Complete problems
  - If there is an exponential lower bound for any NP-Complete problem, they all are intractable!

Complexity and Intractability

- NP - class of problems which admit non-deterministic polynomial-time algorithms
- P - class of problems which admit (deterministic) polynomial-time algorithms
- NP-Complete - the hardest of the NP problems (every NP problem can be transformed to an NP-Complete problem in polynomial time)
- So, is NP = P or not?

Complexity and Intractability

- We don’t know!
- The NP=P? problem has been open since it was posed in 1971 and is one of the most difficult unresolved problems in computer science

Searching

Linear (Sequential) Search

- Linear (Sequential) Search
- Begin at the beginning of the list
- Proceed through the list, sequentially and element by element,
- Until the key is encountered
- Or the end of the list is reached

Linear (Sequential) Search

- Note: we treat a list as a general concept, decoupled from its implementation
- The order of complexity is O(n)
- The list does not have to be in sorted order

Binary Search

- This is exactly the same search strategy which we met in the section on binary search trees.
- In this instance, however, we will be using arrays.
- The main point to note here is that the elements of the array must be sorted
  - just as the binary search tree was

Binary Search

- The essential idea is to begin in the beginning of the list
- Check to see whether the key is
  - equal to
  - less than
  - greater than
- the middle element

Binary Search

- If key is equal to the middle element, then terminate
- If key is less than the middle element, then search the left half
- If key is greater than the middle element, then search the right half
- Continue until either
  - the key is found or
  - there are no more elements to search

Implementation of Binary_Search

```
Pseudo-code first

Binary_Search(list, key, upper_bound, index, found)

identify sublist to be searched by setting bounds on
   search

REPEAT
    get middle element of list
    if middle element < key
       then reset bounds to make the right sublist
            the list to be searched
       else reset bounds to make the left sublist
            the list to be searched
UNTIL list is empty or key is found
```

Implementation of Binary_Search in Modula

```
CONST n = 100;

TYPE bounds_type = 1..n;
     key_type    = INTEGER;
     list_type   = ARRAY[bounds_type] OF key_type;

PROCEDURE binary_search(list: list_type,
                        key: key_type,
                        bounds: bounds_type,
                        VAR index: bounds_type,
                        VAR found: BOOLEAN);

VAR first, last, mid : bounds_type
```

Implementation of Binary_Search in Modula

```
(* assume at least one element in the list *)
BEGIN
   first :=  1;
   last  := bounds;

   REPEAT
      mid :=  (first + last) DIV 2;
      IF list[mid] < key
         THEN
            first := mid + 1
         ELSE
            last := mid - 1
      END
   UNTIL (first > last) OR (list[mid]  = key);
```

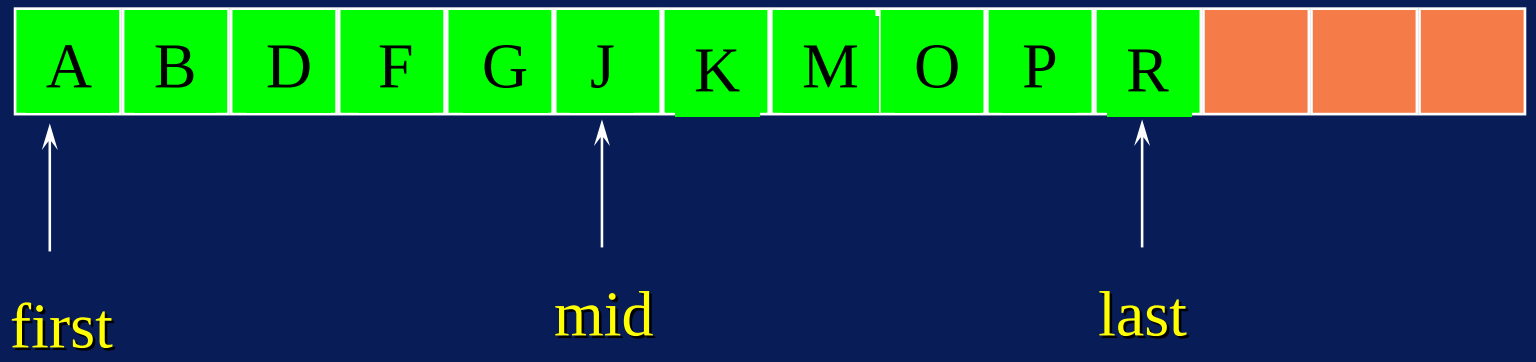
Implementation of Binary_Search in Modula

```
    found := key = list[mid];
    index := mid
END binary_search
```

Binary Search

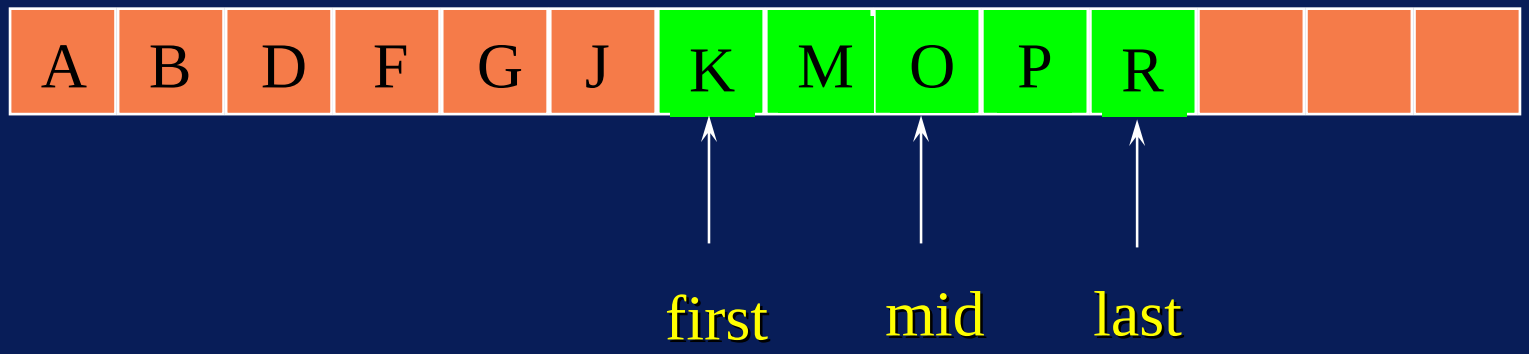

```
first:
last:
mid:
list[mid]:
key:        P
```

Binary Search



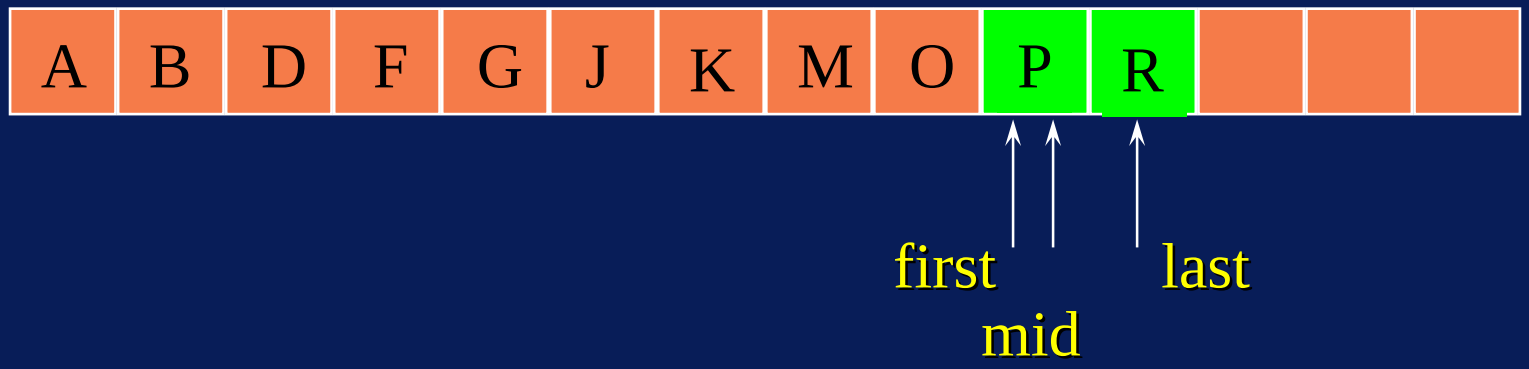

```
first:      1
last:      11
mid:        6
list[mid]: J
key:        P
```

Binary Search




```
first:      1   7
last:      11  11
mid:        6   9
list[mid]: J   O
key:        P   P
```

Binary Search

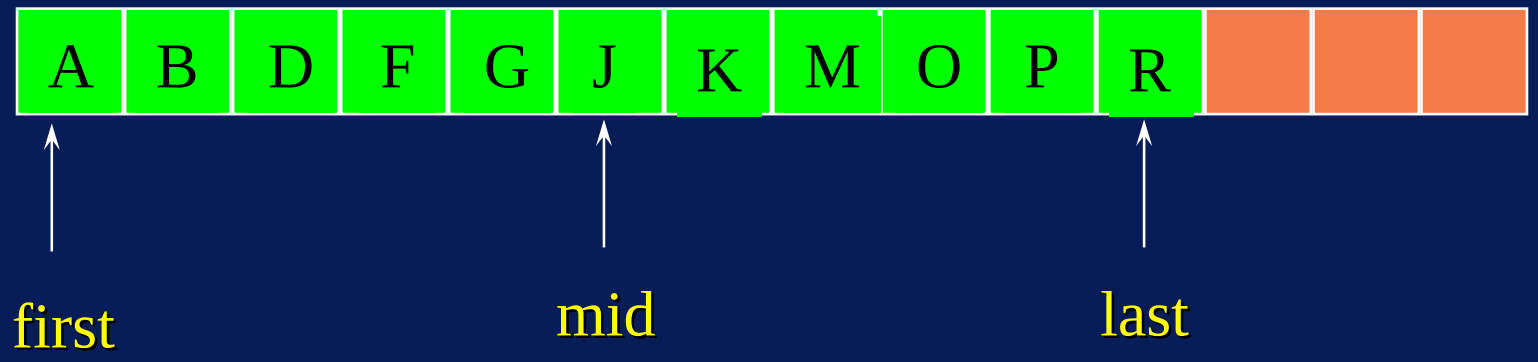



```
first:      1   7   10
last:      11  11   11
mid:        6   9   10
list[mid]: J   O    P   <-- FOUND!
key:        P   P    P
```

Binary Search

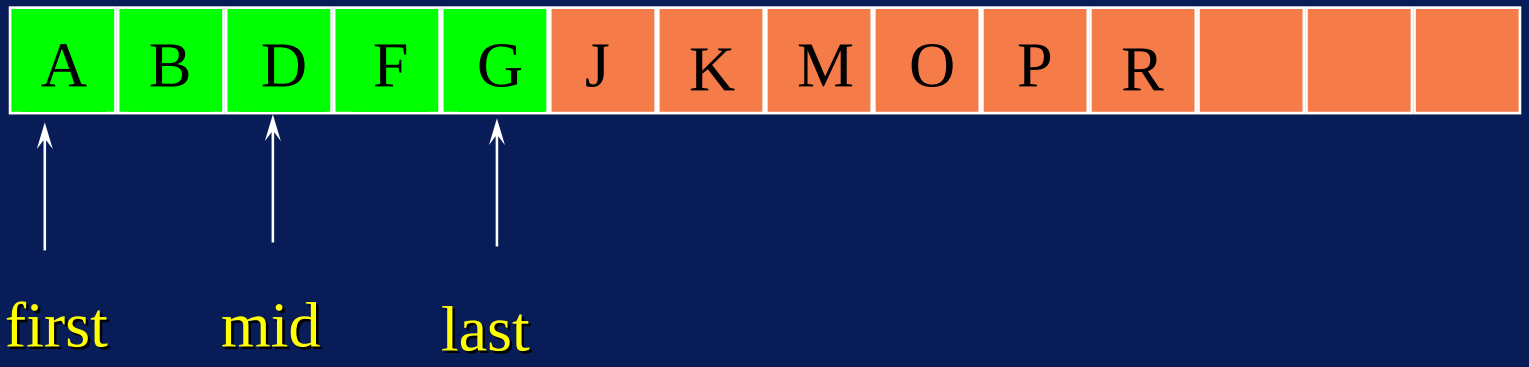

```
first:
last:
mid:
list[mid]:
key:        E
```

Binary Search



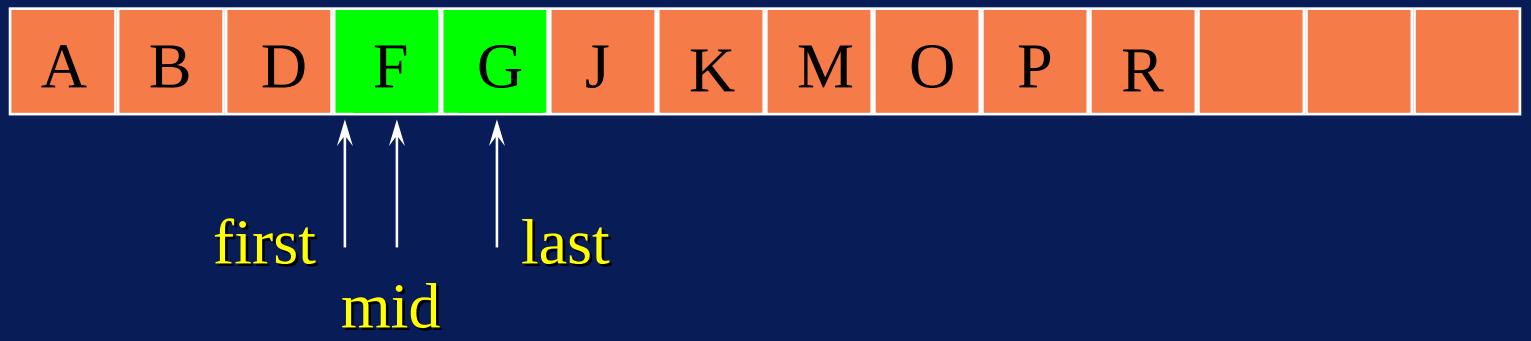

```
first:      1
last:      11
mid:        6
list[mid]: J
key:        E
```

Binary Search



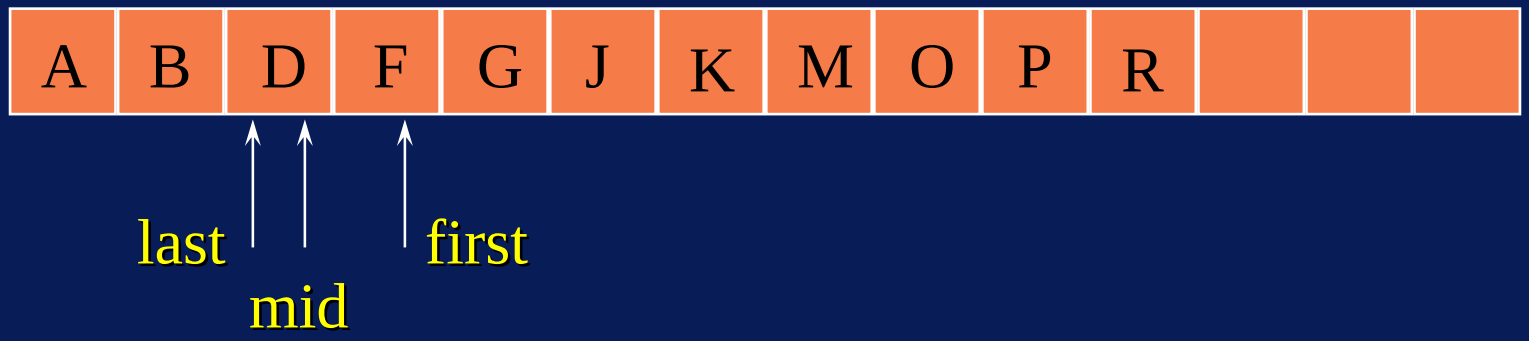

```
first:      1   1
last:      11   5
mid:        6   3
list[mid]:  J   D
key:        E   E
```

Binary Search




```
first:      1   1   4
last:      11   5   5
mid:        6   3   4
list[mid]: J   D   F
key:       E   E   E
```

Binary Search




```
first:      1   1   4   4
last:      11   5   5   3
mid:        6   3   4   3
list[mid]: J   D   F   D
key:       E   E   E   E
```

first > last: NOT FOUND!

Sorting Algorithms

Sorting Algorithms

- Bubble Sort
- Quick Sort

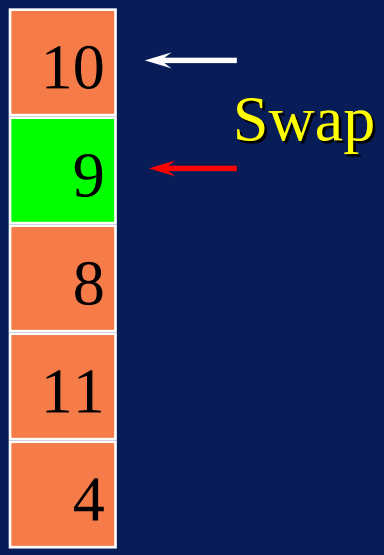
Bubble Sort

- Assume we are sorting a list represented by an array A of n integer elements
- Bubble sort algorithm in pseudo-code

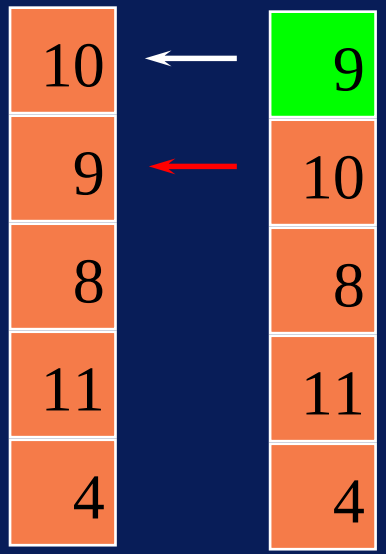
```
FOR every element in the list,
    proceeding for the first to the last
DO
   WHILE list element > previous list element
      bubble element back (up) the list
      by successive swapping with
      the element just above/prior it
```

Bubble Sort

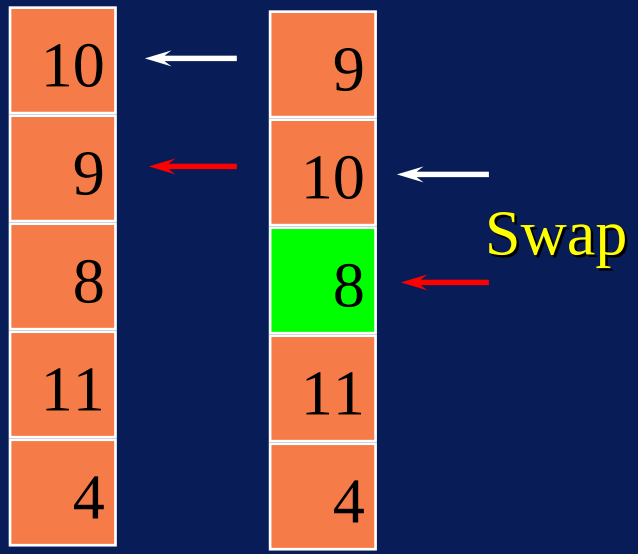
Bubble Sort



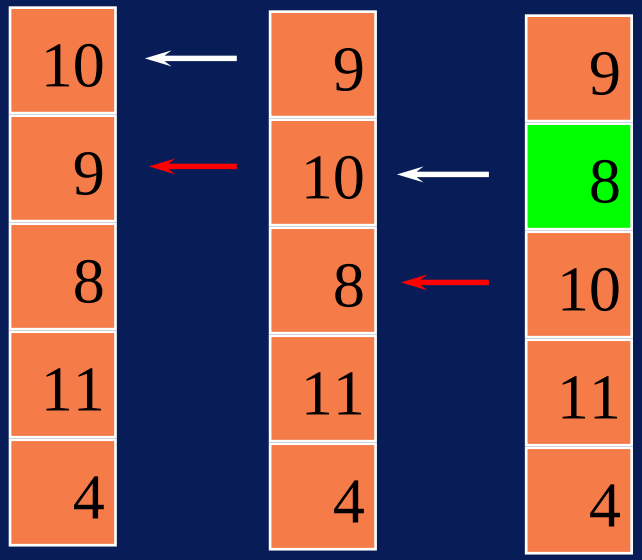
Bubble Sort



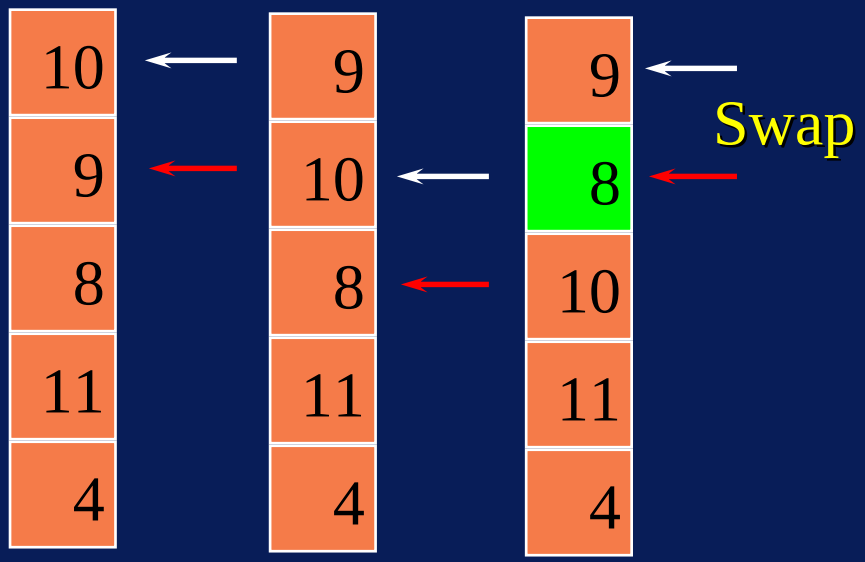
Bubble Sort



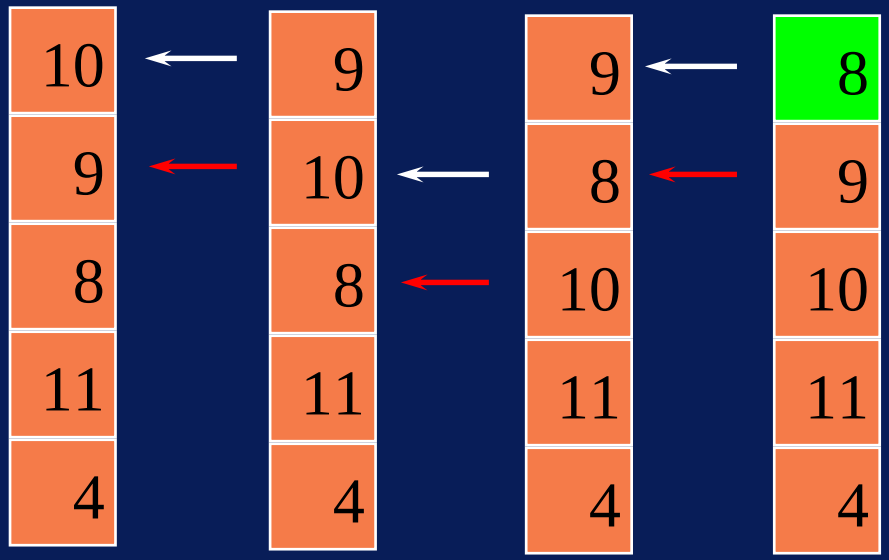
Bubble Sort



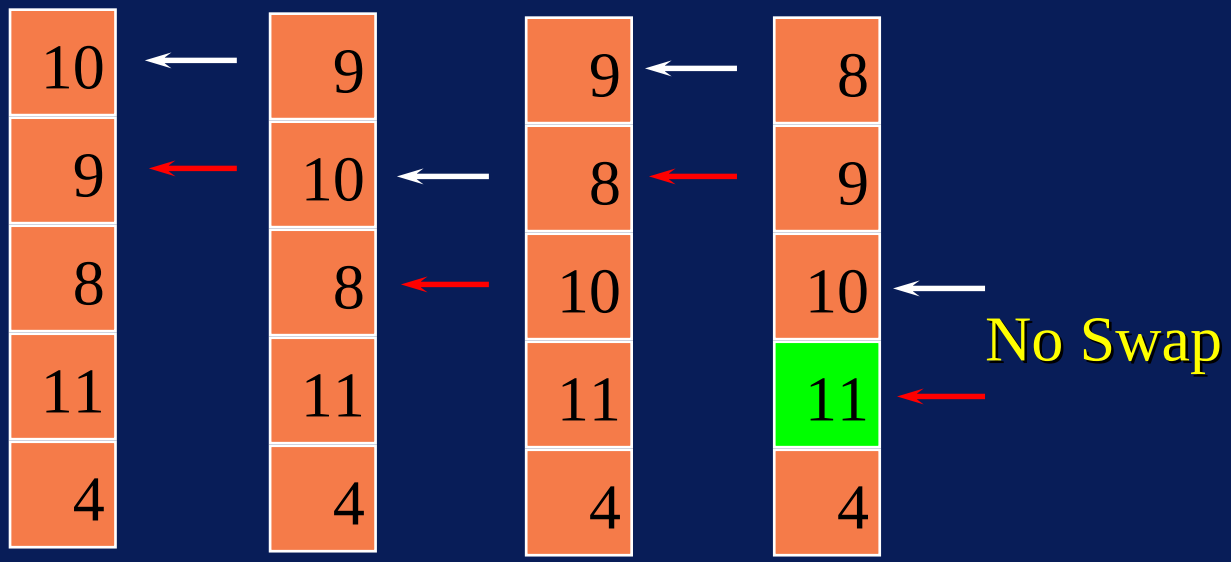
Bubble Sort



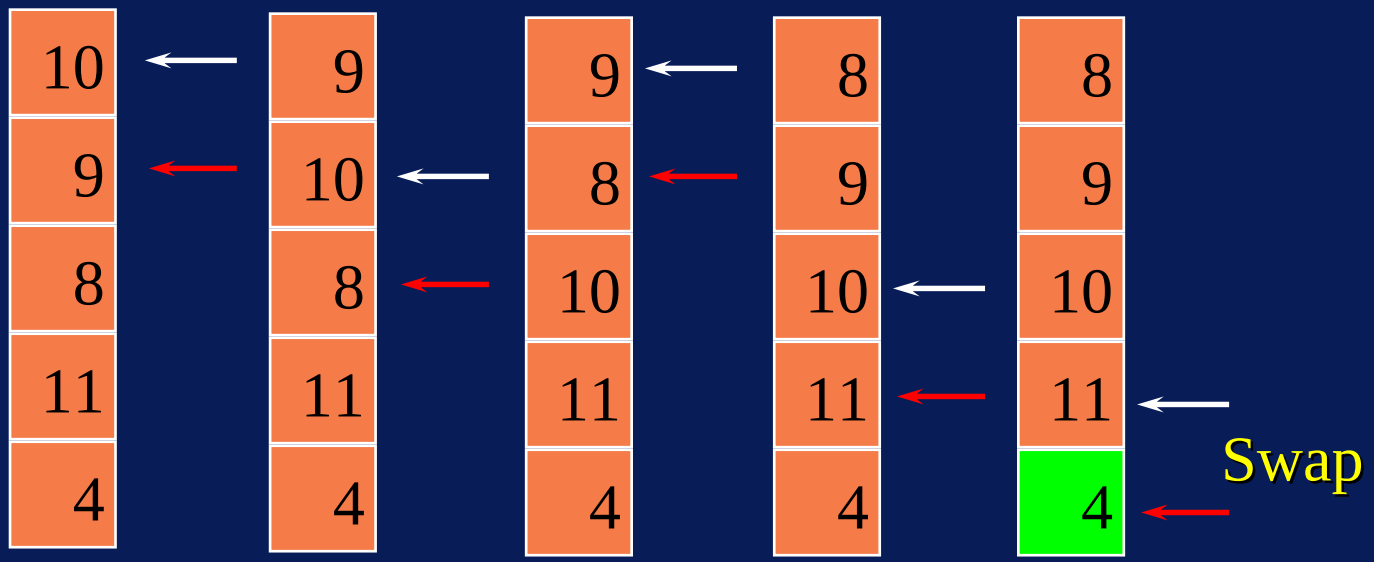
Bubble Sort



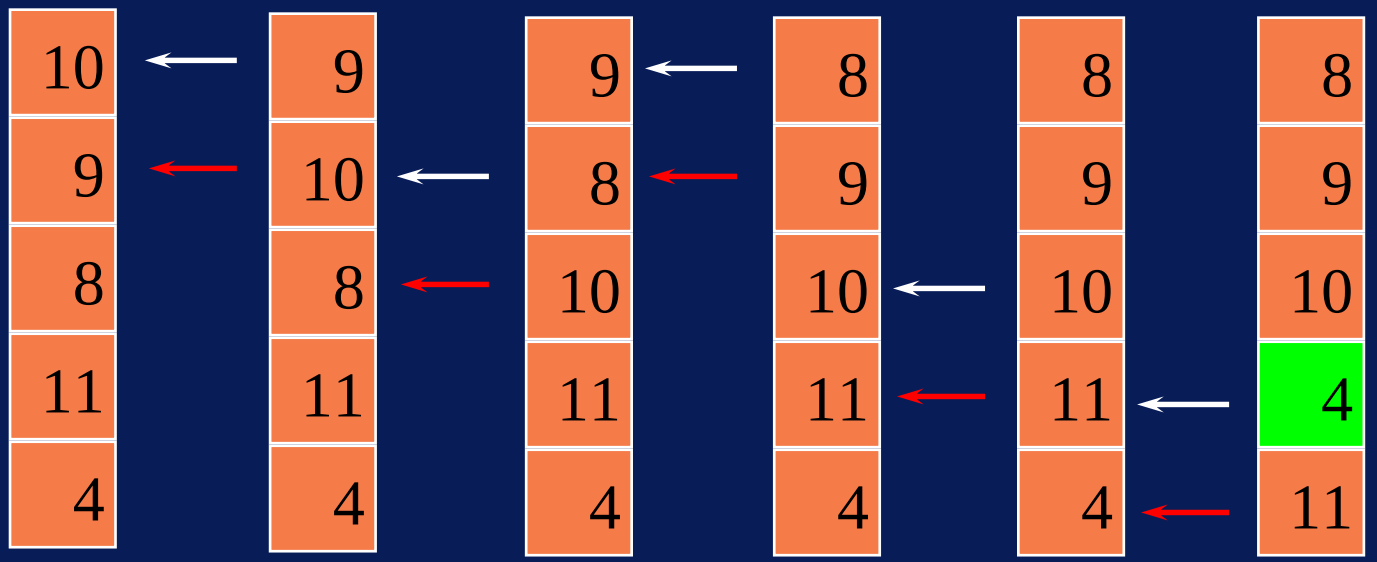
Bubble Sort



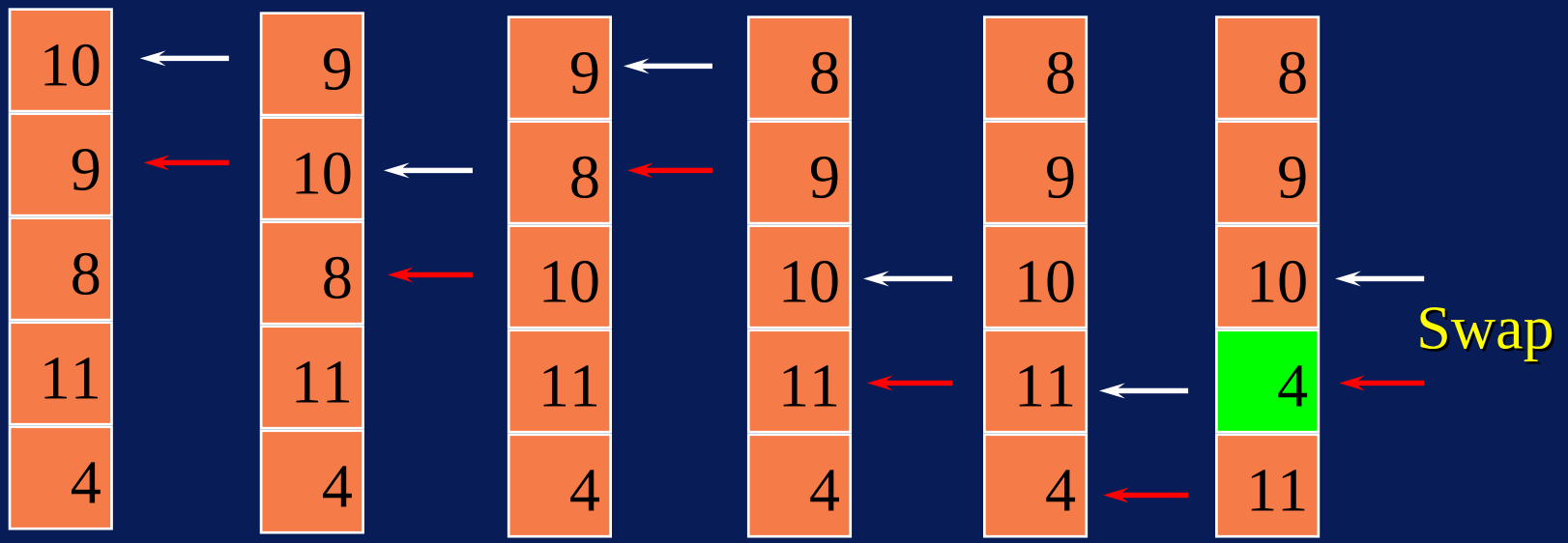
Bubble Sort



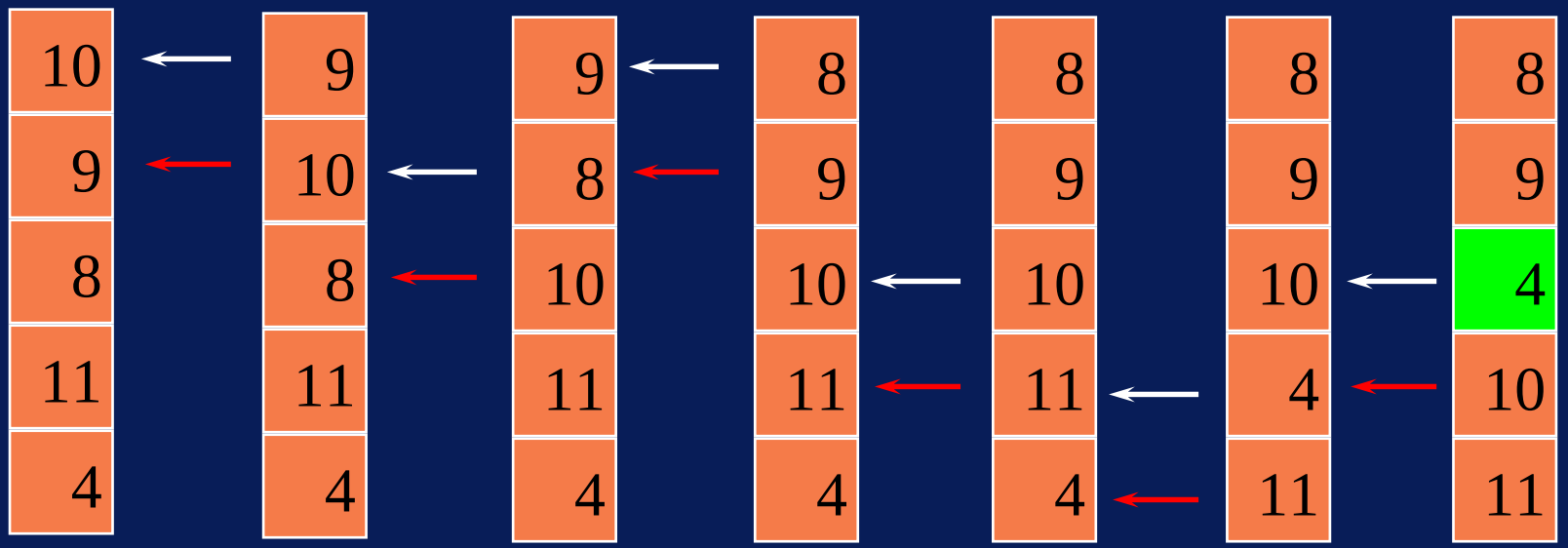
Bubble Sort



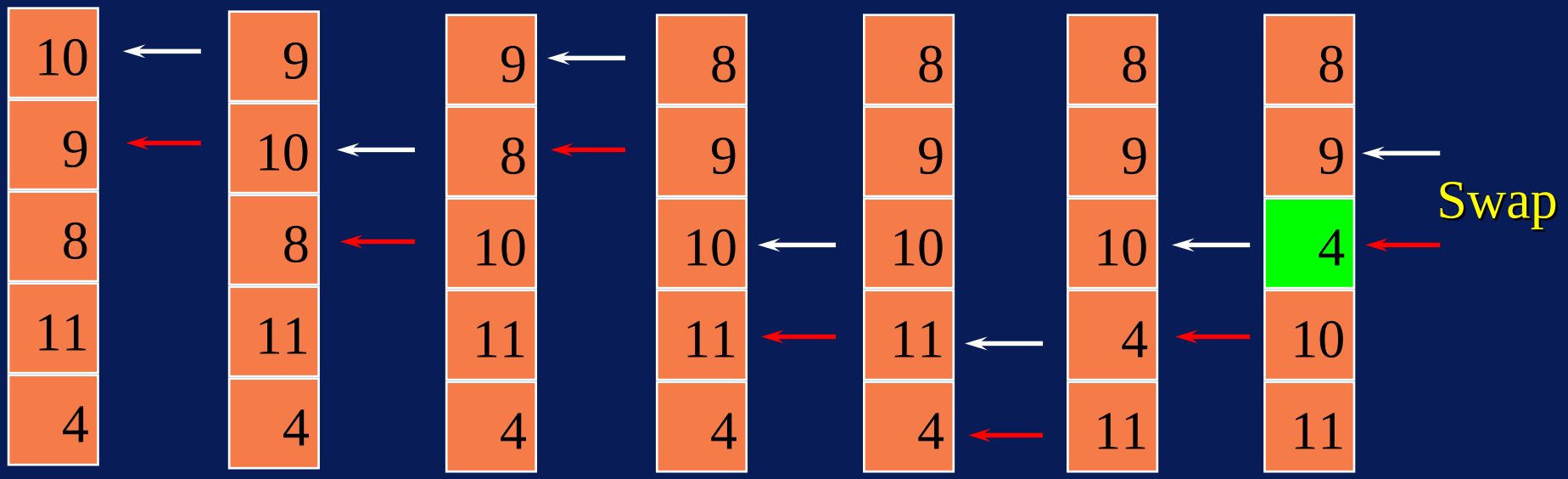
Bubble Sort



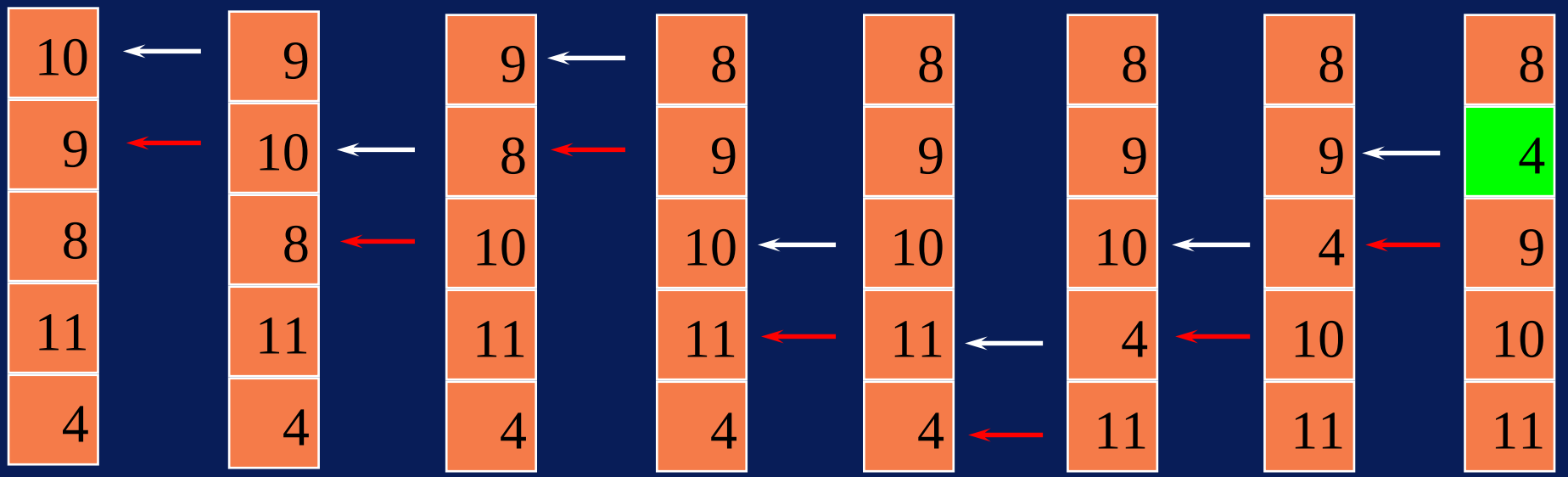
Bubble Sort



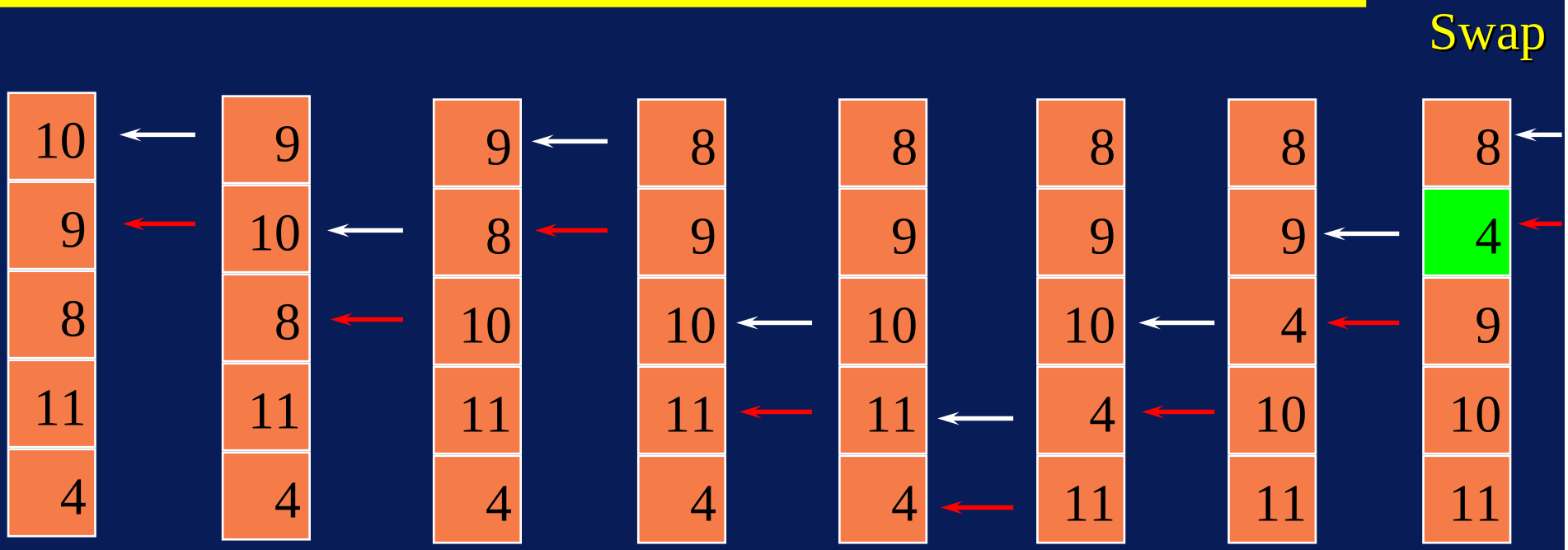
Bubble Sort



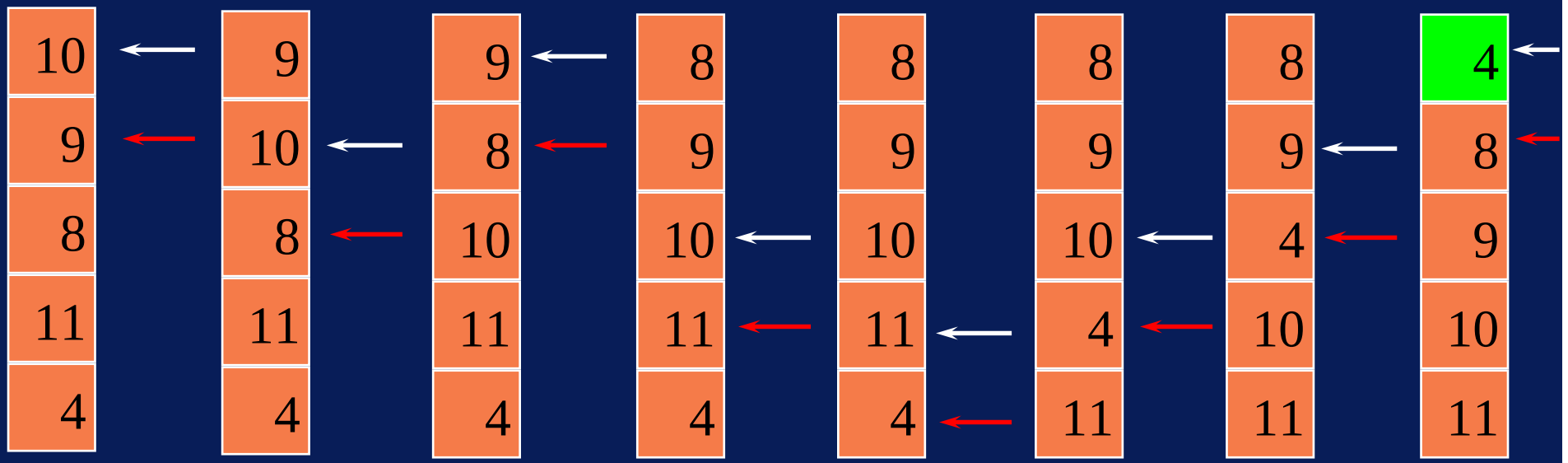
Bubble Sort



Bubble Sort



Bubble Sort



Implementation of Bubble_Sort()

```
Int bubble_sort(int *a, int size) {
   int i,j, temp;

   for (i=0; i < size-1; i++) {
      for (j=i; j >= 0; j--) {
         if (a[j] > a[j+1]) {

            /* swap */
            temp = a[j+1];
            a[j+1] = a[j];
            a[j] = temp;
         }
      }
   }
}
```

Bubble Sort

- A few observations:
  - we don't usually sort numbers; we usually sort records with keys
    - » the key can be a number
    - » or the key could be a string
    - » the record would be represented with a `struct`
  - The swap should be done with a function (so that a record can be swapped)
  - We can make the preceding algorithm more efficient. How? (hint: do we always have to bubble back to the top?)

Bubble Sort

- Exercise: implement these changes and write a driver program to test:
  - the original bubble sort
  - the more efficient bubble sort
  - the bubble sort with a swap function
  - the bubble sort with structures
  - compute the order of time complexity of the bubble sort

Quicksort

- The Quicksort algorithm was developed by C.A.R. Hoare. It has the best average behaviour in terms of complexity:

  Average case: $O(n \log_2 n)$
  Worst case: $O(n^2)$

Quicksort

- Given a list of elements,
- take a partitioning element
- and create a (sub)list
  - such that all elements to the left of the partitioning element are less than it,
  - and all elements to the right of it are greater than it.
- Now repeat this partitioning effort on each of these two sublists

Quicksort

- And so on in a recursive manner until all the sublists are empty, at which point the (total) list is sorted
- Partitioning can be effected simultaneously, scanning left to right and right to left, interchanging elements in the wrong parts of the list
- The partitioning element is then placed between the resultant sublists (which are then partitioned in the same manner)

Implementation of Quicksort()

```
In pseudo-code first

If anything to be partitioned
   choose a pivot
   DO
      scan from left to right until we find an element
      > pivot: i points to it

      scan from right to left until we find an element
      < pivot: j points to it

      IF  i < j
         exchange ith and jth element
   WHILE i <= j
```

Implementation of Quicksort()

exhange pivot and j^{th} element

partition from 1^{st} to j^{th} elements

partition from i^{th} to r^{th} elements

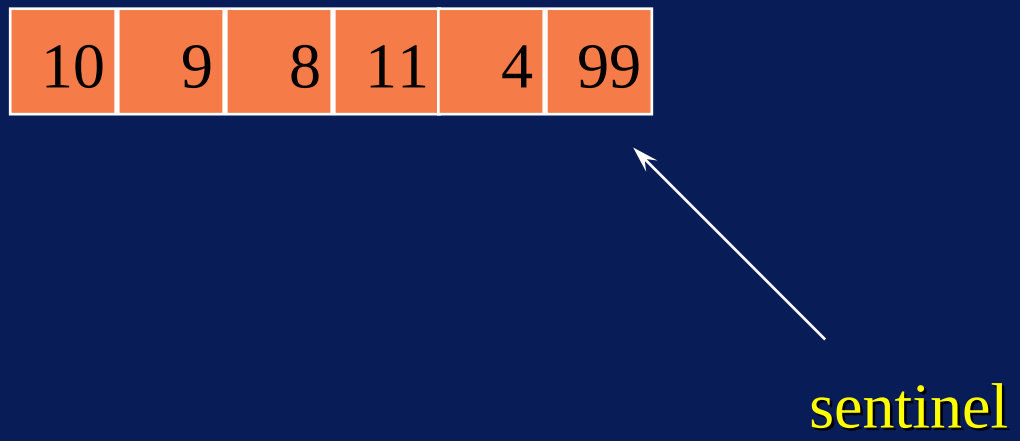
Implementation of Quicksort()

```
/* simple quicksort to sort an array of integers */

void quicksort (int A[], int L, int R)
{
   int i, j, pivot;

/* assume A[R] contains a number > any element,    */
/* i.e. it is a sentinel.                          */
```

Implementation of Quicksort()

```
if ( R > L) {
   i = L; j = R;
   pivot = A[i];
   do {
      while (A[i] <= pivot) i=i+1;
      while ((A[j] >= pivot) && (j>l)) j=j-1;
      if (i < j) {
         exchange(A[i],A[j]); /*between partitions*/
         i = i+1; j = j-1;
      }
   } while (i <= j);
   exchange(A[L], A[j]); /* reposition pivot */
   quicksort(A, L, j);
   quicksort(A, i, R);   /*includes sentinel*/
}
```

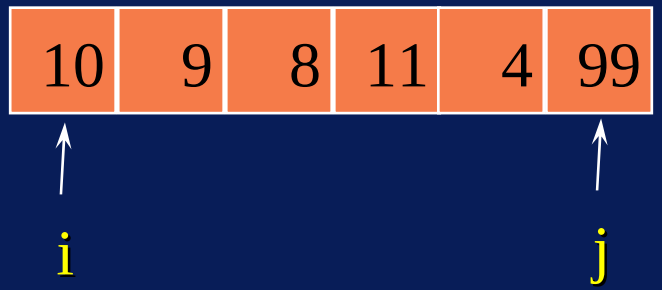
Quicksort



Quicksort

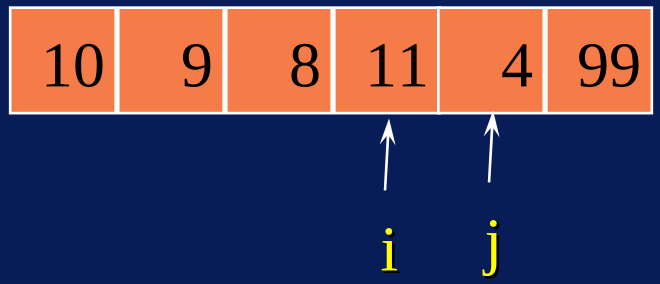

```
QS(A, , )

L:
R:
i:
j:
pivot:
```

Quicksort



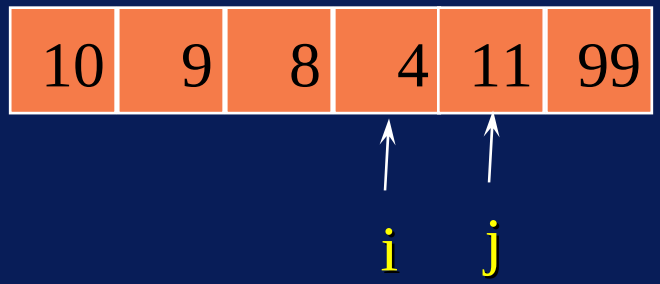

```
QS(A,1,6)

L:      1
R:      6
i:      1
j:      6
pivot: 10
```

Quicksort



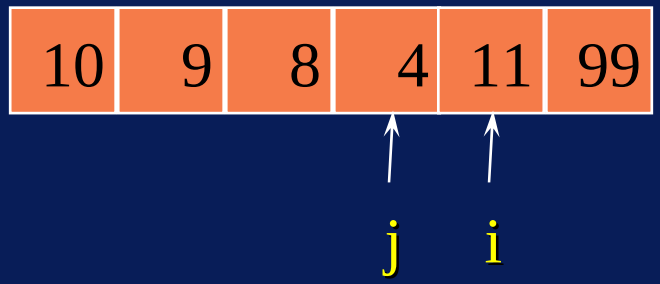

```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4
j:      6 5
pivot: 10
```

Quicksort



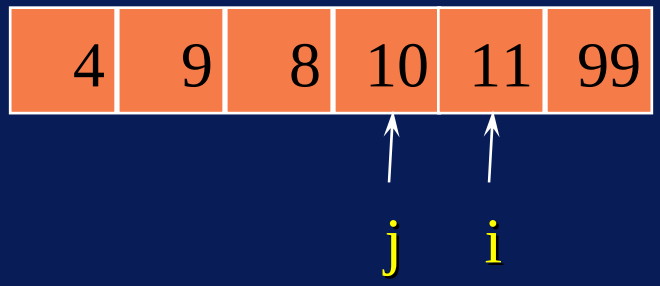

```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4
j:      6 5
pivot: 10
```

Quicksort




```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4 5
j:      6 5 4
pivot: 10
```

Quicksort




```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4 5
j:      6 5 4
pivot: 10
```

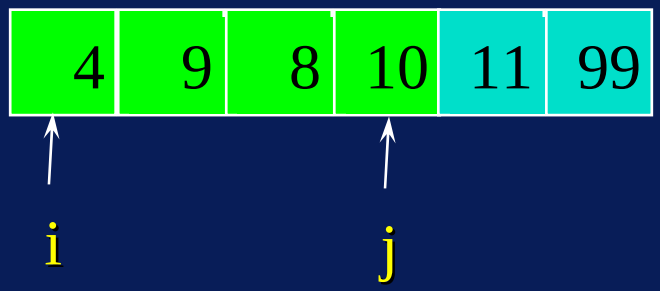
Quicksort


```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4 5
j:      6 5 4
pivot: 10
```

```
QS(A,1,4)

L:      1
R:      4
i:
j:
pivot:  4
```

```
QS(A,5,6)

L:      5
R:      6
i:
j:
pivot:  11
```

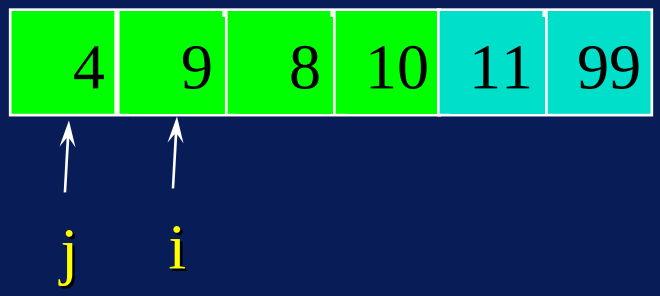
Quicksort




```
QS(A,1,4)

L:      1
R:      4
i:      1
j:      4
pivot:  4
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort




```
QS(A,1,4)

L:      1
R:      4
i:      1 2
j:      4 3 2 1
pivot:  4
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort

QS(A,1,4)		QS(A,1,1)		QS(A,2,4)		QS(A,5,6)	
L:	1	L:	1	L:	2	L:	5
R:	4	R:	1	R:	4	R:	6
i:	1 2	i:		i:		i:	5
j:	4 3 2 1	j:		j:		j:	6
pivot:	4	pivot:	4	pivot:	9	pivot:	11

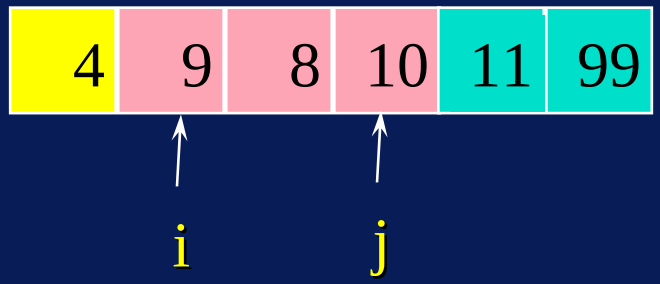
Quicksort


```
QS(A,1,1)

L:      1
R:      1
i:
j:
pivot:  4
```

```
QS(A,2,4)

L:      2
R:      4
i:
j:
pivot:  9
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

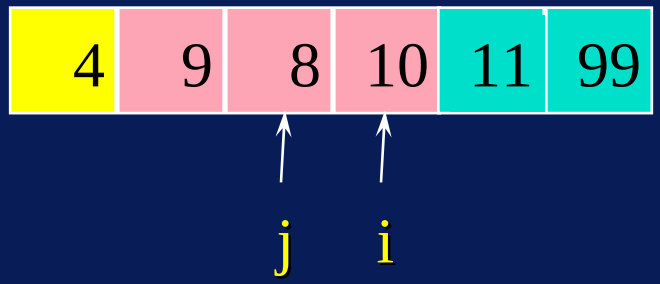
Quicksort




```
QS(A,2,4)

L:      2
R:      4
i:      2
j:      4
pivot:  9
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

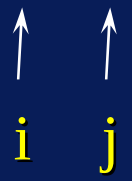
Quicksort




```
QS(A,2,4)

L:      2
R:      4
i:      2 3 4
j:      4 3
pivot:  9
```

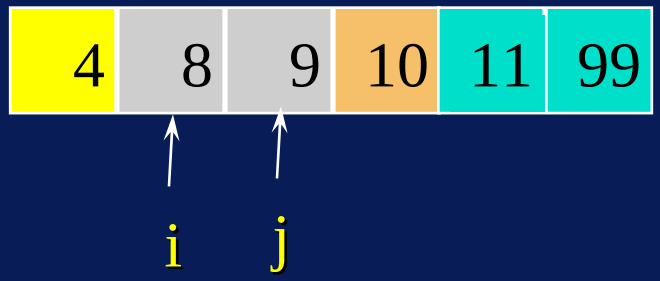
```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort



QS(A,2,4)		QS(A,2,3)		QS(A,4,4)		QS(A,5,6)	
L:	2	L:	2	L:	4	L:	5
R:	4	R:	3	R:	4	R:	6
i:	2 3 4	i:		i:		i:	5
j:	4 3	j:		j:		j:	6
pivot:	9	pivot:	8	pivot:	10	pivot:	11

Quicksort

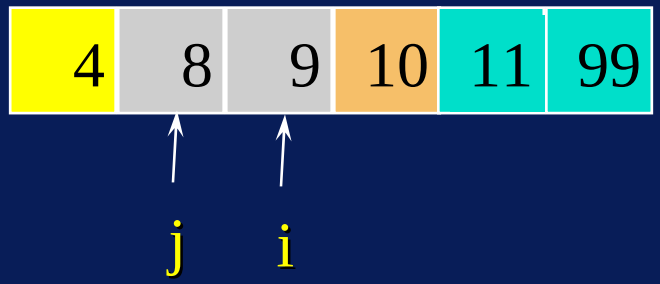



```
QS(A,2,3)

L:      2
R:      3
i:      2
j:      3
pivot:  8
```

```
QS(A,4,4)

L:      4
R:      4
i:
j:
pivot:  10
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort




```
QS(A,2,3)

L:      2
R:      3
i:      2 3
j:      3 2
pivot:  8
```

```
QS(A,4,4)

L:      4
R:      4
i:
j:
pivot:  10
```

```
QS(A,5,6)

L:      5
R:      6
i:      5
j:      6
pivot:  11
```

Quicksort

QS(A,2,3)		QS(A,2,2)		QS(A,3,3)	QS(A,4,4)	QS(A,5,6)
L:	2	L:	2	L: 3	L: 4	L: 5
R:	3	R:	2	R: 3	R: 4	R: 6
i:	2 3	i:		i:	i:	i: 5
j:	3 2	j:		j:	j:	j: 6
pivot:	8	pivot:	8	pivot: 9	pivot: 10	pivot: 11

Quicksort

QS(A,4,4)		QS(A,5,6)	
L:	4	L:	5
R:	4	R:	6
i:		i:	5
j:		j:	6
pivot:	10	pivot:	11

Quicksort


```
QS(A,5,6)     QS(A,5,5)     QS(A,6,6)

L:       5    L:       5    L:       6
R:       6    R:       5    R:       6
i:       5    i:            i:
j:       6    j:            j:
pivot:  11    pivot:  11    pivot:  99
```

Data Structures

Data Structures

- Lists
- Stacks (special type of list)
- Queues (another type of list)
- Trees
  - General introduction
  - Binary Trees
  - Binary Search Trees (BST)
- Use *Abstract Data Types* (ADT)

Abstract Data Types

- ADTs are an old concept
  - Specify the complete set of values which a variable of this *type* may assume
  - Specify completely the set of all possible operations which can be applied to values of this *type*

Abstract Data Types

- It's worth noting that object-oriented programming gives us a way of combining (or **encapsulating**) both of these specifications in one logical definition
  - **Class** definition
  - **Objects** are instantiated classes
- Actually, object-oriented programming provides much more than this (e.g. inheritance and polymorphism)

Lists

Lists

- A list is an ordered sequence of zero or more elements of a given type

  $a_1, a_2, a_3, \ldots a_n$

  - a_i is of type *elementtype*
  - a_i precedes a_{i+1}
  - a_{i+1} succeeds or follows a_i
  - If n=0 the list is empty: a null list
  - The position of a_i is i

Lists

List element w
(w is of type windowtype:
w could be, but is not necessarily,
the integer sequence position of
the element in the list)

Element of type elementtype

LIST: An ADT specification of a list type

- Let L denote all possible values of type LIST (*i.e.* lists of elements of type *elementtype)*
- Let E denote all possible values of type *elementtype*
- Let B denote the set of Boolean values *true* and *false*
- Let W denote the set of values of type *windowtype*

LIST Operations

- *Syntax of ADT Definition*:

  Operation:

  What_You_Pass_It → What_It_Returns :

LIST Operations

- *Declare*: $\rightarrow$ L :

  The function value of *Declare(L)* is an empty list

  – alternative syntax: *LIST L*

LIST Operations

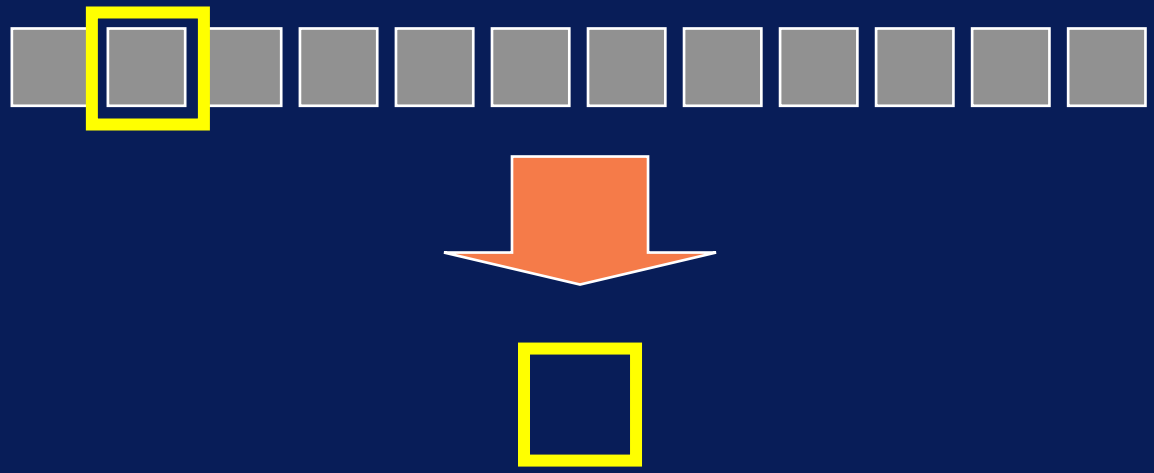
- *End*: L $\rightarrow$ W :

  The function *End(L)* returns the position after the last element in the list
  (i.e. the value of the function is the window position after the last element in the list)

LIST Operations

- *Empty*: L $\rightarrow$ L x W :

  The function *Empty* causes the list to be emptied and it returns position *End(L)*



LIST Operations

- *IsEmpty*: L $\rightarrow$ B :

  The function value *IsEmpty*(*L*) is *true* if *L* is empty; otherwise it is *false*

LIST Operations

- *First*: L $\rightarrow$ W :

  The function value *First*(*L*) is the window position of the first element in the list;

  if the list is empty, it has the value *End(L)*

LIST Operations

- *Next*: L $\times$ W $\rightarrow$ W :

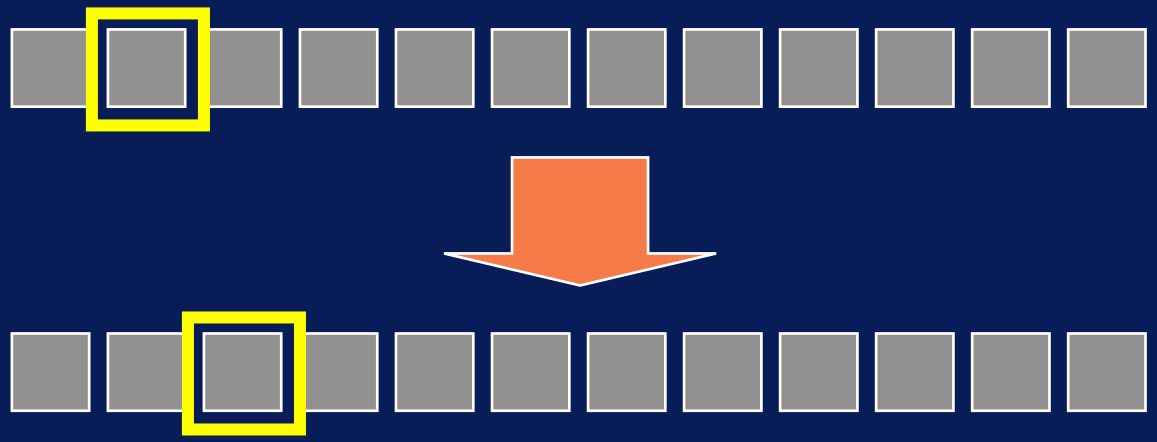
  The function value *Next*(*w*,*L*) is the window position of the next successive element in the list;

  if we are already at the end of the list then the value of *Next*(*w,L*) is *End(L);*

  if the value of *w* is *End(L)*, then the operation is undefined

LIST Operations

Next(*w*,*L*)



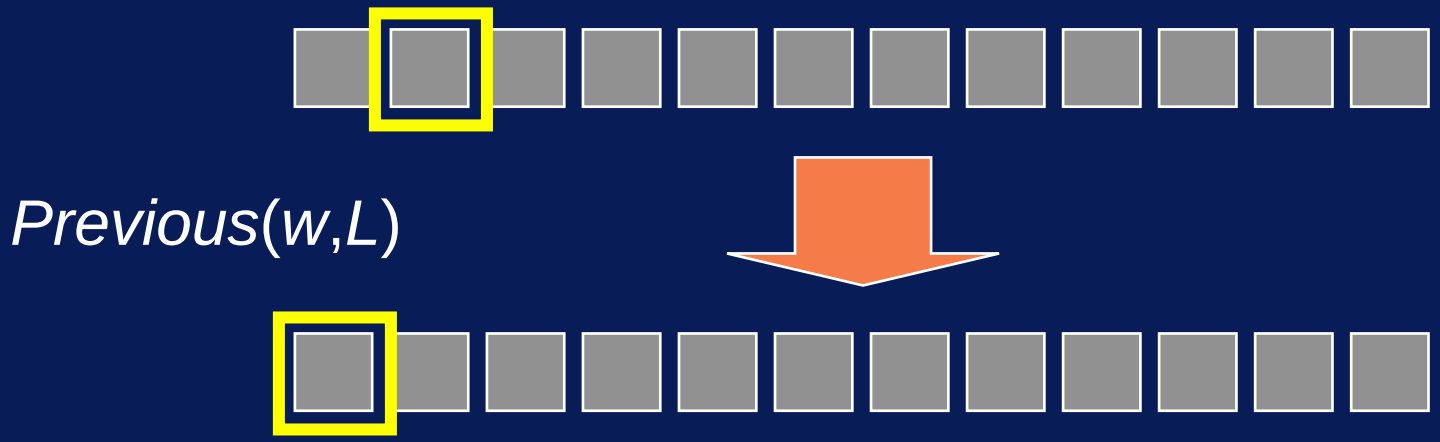
LIST Operations

- *Previous*: $L \times W \rightarrow W$:

  The function value *Previous*(w, *L*) is the window position of the previous element in the list;

  if we are already at the beginning of the list (*w=First*(*L*)), then the value is undefined

LIST Operations



LIST Operations

- *Last*: L → W :

  The function value *Last*(*L*) is the window position of the last element in the list;

  if the list is empty, it has the value *End(L)*

LIST Operations

- *Insert*: $E \times L \times W \rightarrow L \times W$:

  Insert(e, w, L)
  Insert an element *e* at position *w* in the list *L*, moving elements at *w* and following positions to the next higher position

  $a_1, a_2, \ldots a_n \rightarrow a_1, a_2, \ldots, a_{w-1}, e, a_w, \ldots, a_n$

LIST Operations

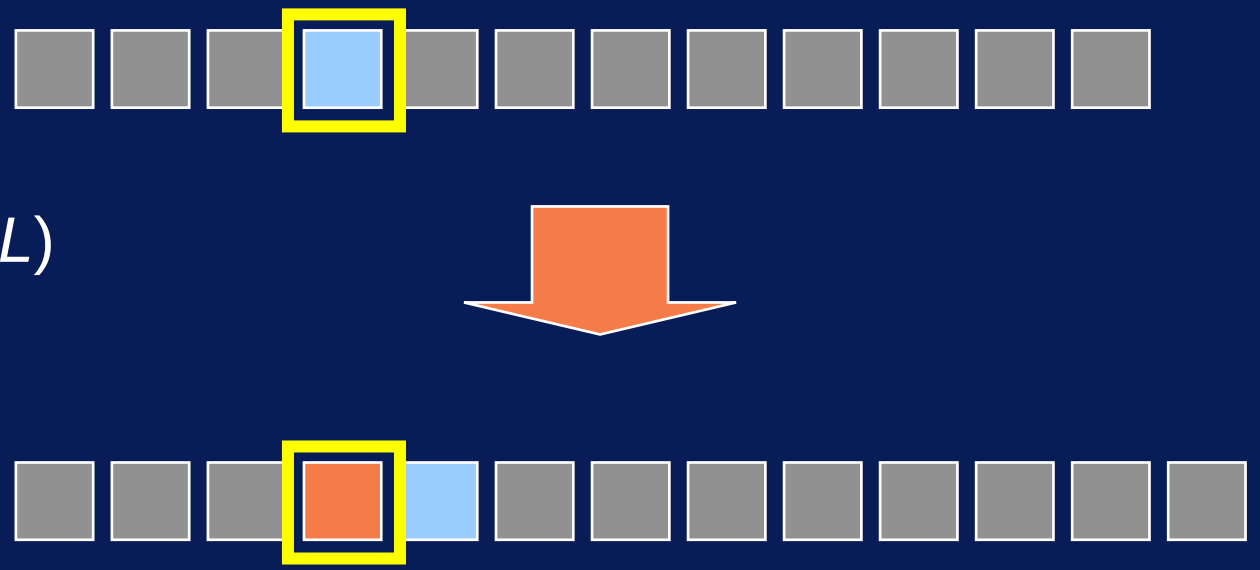
If w = *End(L)* then

$a_1, a_2, \ldots a_n \rightarrow a_1, a_2, \ldots, a_n, e$

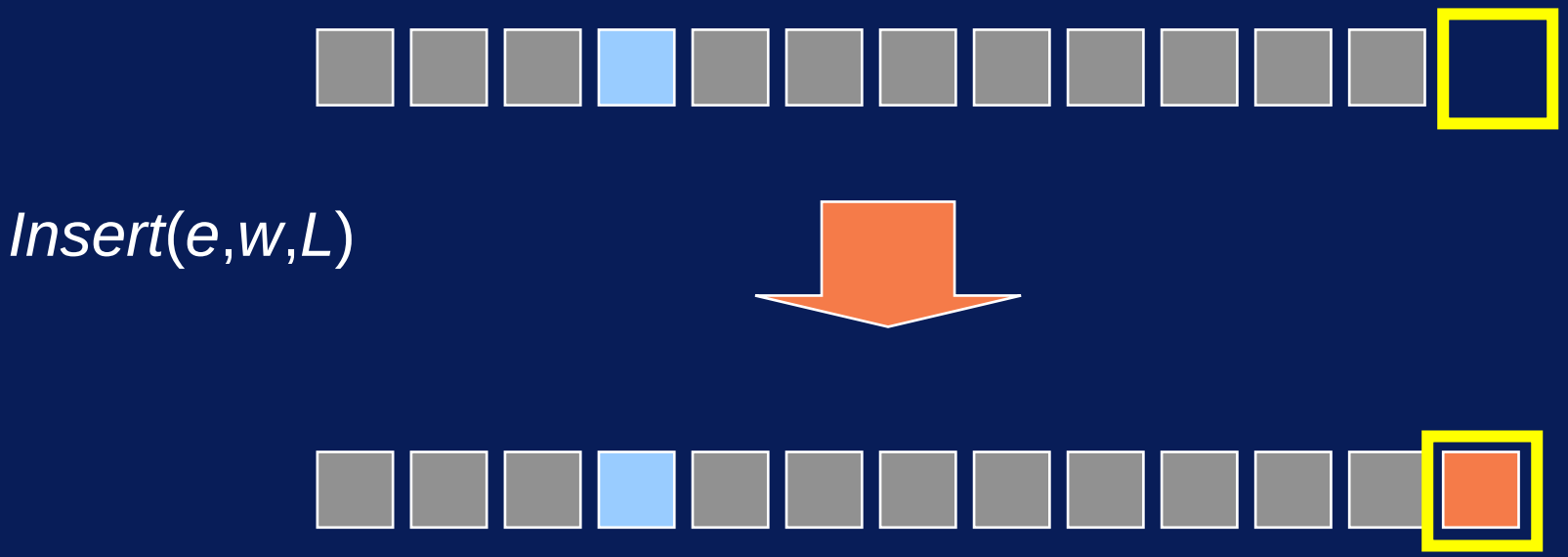
The window *w* is moved over the new element *e*

The function value is the list with the element inserted

LIST Operations



LIST Operations



LIST Operations

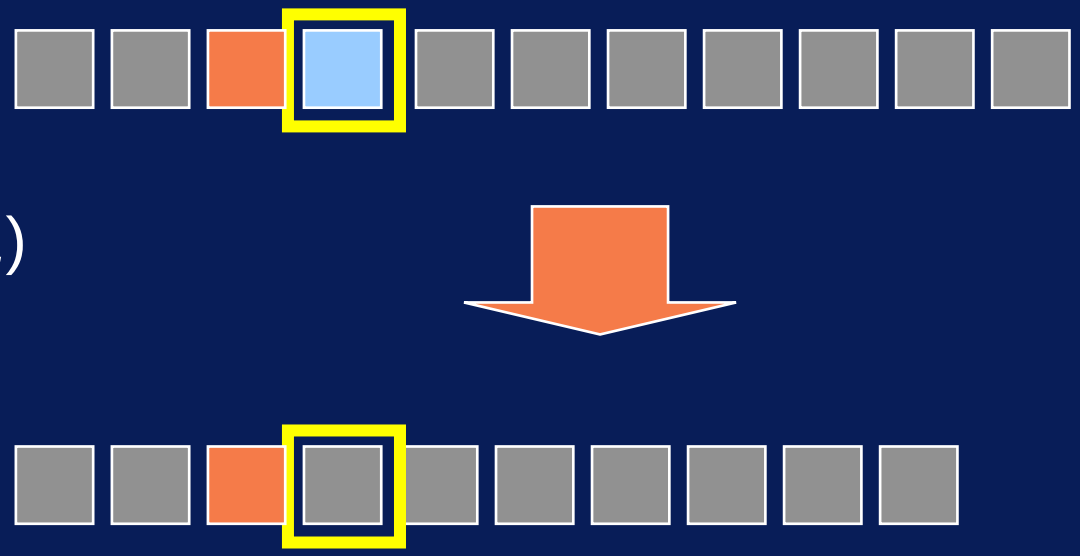
- *Delete*: $L \times W \rightarrow L \times W$:

  Delete(w, L)
  Delete the element at position *w* in the list *L*

  $a_1, a_2, \ldots a_n \rightarrow a_1, a_2, \ldots, a_{w-1}, a_{w+1}, \ldots, a_n$

  – If *w* = *End(L)* then the operation is undefined
  – The function value is the list with the element deleted

LIST Operations



LIST Operations

- *Examine*: $L \times W \rightarrow E$:

  The function value *Examine*(*w*, *L*) is the value of the element at position *w* in the list;

  if we are already at the end of the list (*i.e.* w = *End(L)),* then the value is undefined

LIST Operations

- *Declare(L)* *returns listtype*
- *End(L)* *returns windowtype*
- *Empty(L)* *returns windowtype*
- *IsEmpty(L)* *returns Boolean*
- *First(L)* *returns windowtype*
- *Next(w,L)* *returns windowtype*
- *Previous(w,L)* *returns windowtype*
- *Last(L)* *returns windowtype*

LIST Operations

- *Insert(e,w,L)* *returns listtype*
- *Delete(w,L)* *returns listtype*
- *Examine(w,L)* *returns elementtype*

LIST Operations

- *Example of List manipulation*

w = End(L)  empty list

LIST Operations

- *Example of List manipulation*

w = End(L)

Insert(e,w, L)

LIST Operations

- *Example of List manipulation*

w = End(L)

Insert(e,w, L)

Insert(e,w, L)

LIST Operations

- *Example of List manipulation*

w = End(L)

Insert(e,w, L)

Insert(e,w, L)

Insert(e,Last(L), L)

LIST Operations

- *Example of List manipulation*

  w = Next(Last(L),L)

LIST Operations

- *Example of List manipulation*

w = Next(Last(L),L)

Insert(e,w,L)

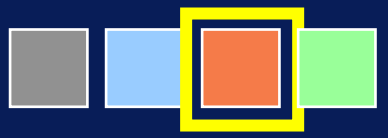
LIST Operations

- *Example of List manipulation*

w = Next(Last(L),L)

Insert(e,w,L)

w = Previous(w,L)



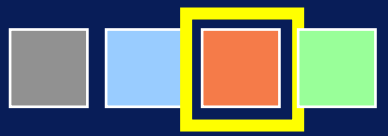
LIST Operations

- *Example of List manipulation*

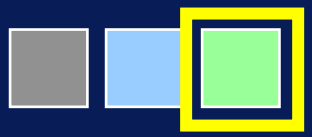
w = Next(Last(L),L)

Insert(e,w,L)

w = Previous(w,L)



Delete(w,L)



ADT Specification

- The key idea is that we have not specified how the lists are to be implemented, merely their values and the operations of which they can be operands
- This ‘old’ idea of data abstraction is one of the key features of object-oriented programming
- C++ is a particular implementation of this object-oriented methodology

ADT Implementation

- Of course, we still have to implement this ADT specification
- The choice of implementation will depend on the requirements of the application

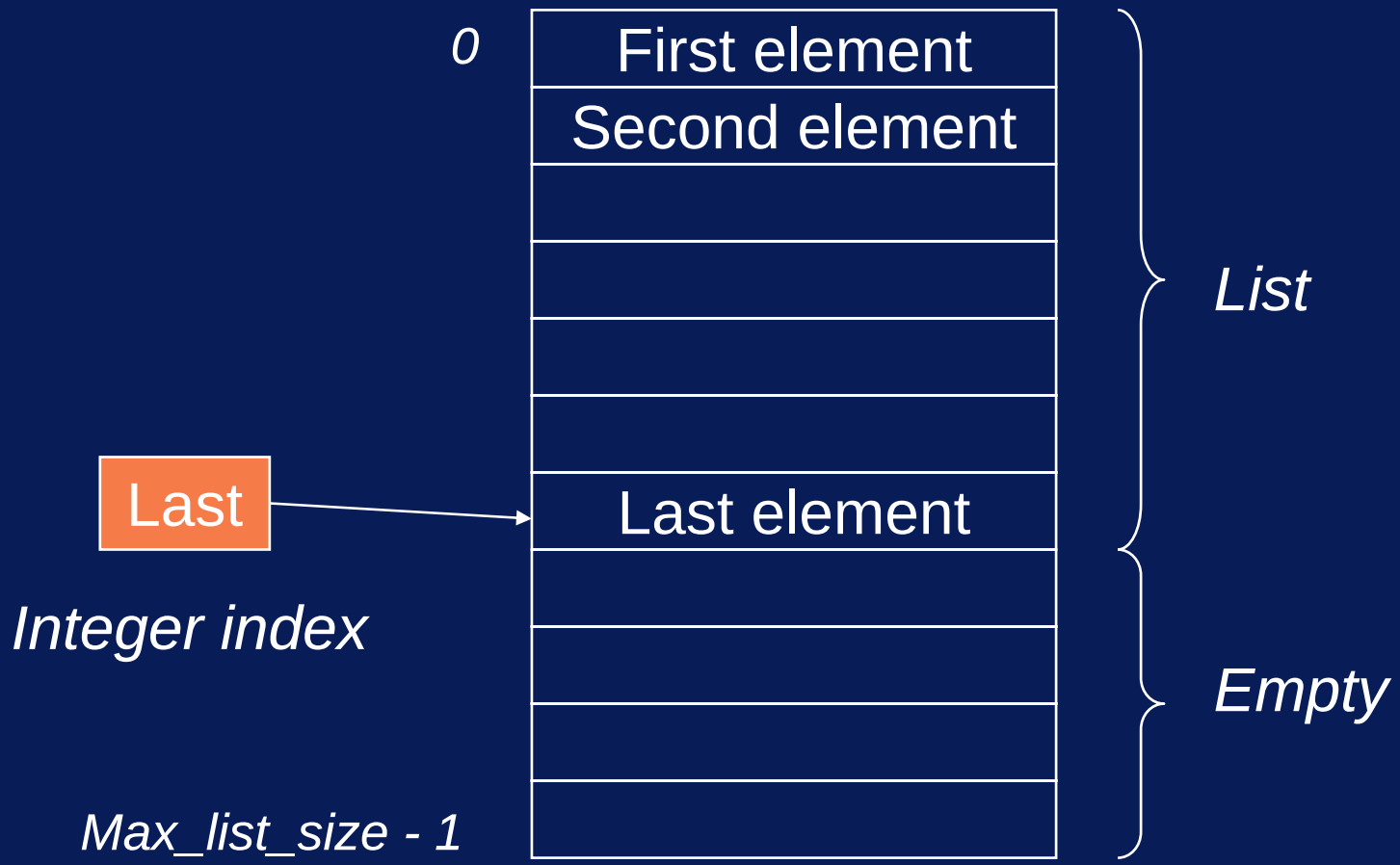
ADT Implementation

- We will look at two implementations
  - Array implementation
    - » uses a static data-structure
    - » reasonable if we know in advance the maximum number of elements in the list
  - Pointer implementation
    - » Also known as a linked-list implementation
    - » uses dynamic data-structure
    - » best if we don’t know in advance the number of elments in the list (or if it varies significantly)
    - » overhead in space: the pointer fields

LIST: Array Implementation

- We will do this in two steps:
  - *the implementation (or representation) of the four constituents datatypes of the ADT:*
    - » *list*
    - » *elementtype*
    - » *Boolean*
    - » *windowtype*
  - *the implementation of each of the ADT operations*

LIST: Array Implementation



LIST: Array Implementation

- type *elementtype*
- type LIST
- type Boolean
- type *windowtype*

LIST: Array Implementation

```
/* array implementation of LIST ADT */

#include <stdio.h>
#include <math.h>
#include <string.h>

#define MAX_LIST_SIZE 100
#define FALSE 0
#define TRUE 1

typedef struct {
           int number;
           char *string;
         } ELEMENT_TYPE;
```

LIST: Array Implementation

```
typedef struct {
           int last;
           ELEMENT_TYPE a[MAX_LIST_SIZE];
         } LIST_TYPE;

typedef int WINDOW_TYPE;

/** position following last element in a list ***/

WINDOW_TYPE end(LIST_TYPE *list) {
   return(list->last+1);
}
```

LIST: Array Implementation

```
/*** empty a list ***/

WINDOW_TYPE empty(LIST_TYPE *list) {
   list->last = -1;
   return(end(list));
}

/*** test to see if a list is empty ***/

int is_empty(LIST_TYPE *list) {
   if (list->last == -1)
      return(TRUE);
   else
      return(FALSE)
}
```

LIST: Array Implementation

```
/*** position at first element in a list ***/

WINDOW_TYPE first(LIST_TYPE *list) {
   if (is_empty(list) == FALSE) {
      return(0);
   else
      return(end(list));
}
```

LIST: Array Implementation

```
/*** position at next element in a list ***/

WINDOW_TYPE next(WINDOW_TYPE w, LIST_TYPE *list) {
   if (w == last(list)) {
      return(end(list));
   else if (w == end(list)) {
      error("can't find next after end of list");
   }
   else {
      return(w+1);
   }
}
```

LIST: Array Implementation

```
/*** position at previous element in a list ***/

WINDOW_TYPE previous(WINDOW_TYPE w, LIST_TYPE *list) {
   if (w != first(list)) {
      return(w-1);
   else {
      error("can't find previous before first element of
  list");
      return(w);
   }
}
```

LIST: Array Implementation

```
/*** position at last element in a list ***/

WINDOW_TYPE last(LIST_TYPE *list) {
return(list->last);
}
```

LIST: Array Implementation

```
/*** insert an element in a list ***/

LIST_TYPE *insert(ELEMENT_TYPE e, WINDOW_TYPE w,
                  LIST_TYPE *list) {
   int i;
   if (list->last >= MAX_LIST_SIZE-1) {
      error("Can't insert - list is full");
   }
   else if ((w > list->last + 1) || (w < 0)) {
      error("Position does not exist");
   }
   else {
      /* insert it … shift all after w to the right */
```

LIST: Array Implementation

```
        for (i=list->last; i>= w; i--) {
            list->a[i+1] = list->a[i];
        }

        list->a[w] = e;
        list->last = list->last + 1;

        return(list);
    }
}
```

LIST: Array Implementation

```
/*** delete an element from a list ***/

LIST_TYPE *delete(WINDOW_TYPE w, LIST_TYPE *list) {
   int i;
   if ((w > list->last) ¦¦ (w < 0)) {
      error(“Position does not exist”);
   }
   else {
      /* delete it … shift all after w to the left */
      list->last = list->last - 1;
      for (i=w; i <= list->last; i++) {
         list->a[i] = list->a[i+1];
      }
      return(list);
}}
```

LIST: Array Implementation

```
/*** retrieve an element from a list ***/

ELEMENT_TYPE retrieve(WINDOW_TYPE w, LIST_TYPE *list) {
   if ( (w < 0)) ¦¦  (w > list->last)) {

      /* list is empty */

      error(“Position does not exist”);
   }
   else {
      return(list->a[w]);
   }
}
```

LIST: Array Implementation

```
/*** print all elements in a list ***/

int print(LIST_TYPE *list) {
   WINDOW_TYPE w;
   ELEMENT_TYPE e;
printf(“Contents of list: \n”);
   w = first(list);
   while (w != end(list)) {
      e = retrieve(w, list);
      printf(“%d %s\n”, e.number, e.string);
      w = next(w, list);
   }
   printf(“---\n”);
   return(0);
}
```

LIST: Array Implementation

```
/*** error handler: print message passed as argument and
     take appropriate action                              ***/
int error(char *s); {
   printf("Error: %s\n", s);
   exit(0);
}

/*** assign values to an element ***/

int assign_element_values(ELEMENT_TYPE *e, int number,
   char s[]) {
   e->string = (char *) malloc(sizeof(char)* strlen(s+1));
   strcpy(e->string, s);
   e->number = number;
}
```

LIST: Array Implementation

```
/*** main driver routine ***/

   WINDOW_TYPE w;
   ELEMEN_TYPE e;
   LIST_TYPE list;
   int i;

   empty(&list);
   print(&list);

   assign_element_values(&e, 1, “String A”);
   w = first(&list);
   insert(e, w, &list);
   print(&list);
```

LIST: Array Implementation

```
assign_element_values(&e, 2, “String B”);
insert(e, w, &list);
print(&list);

assign_element_values(&e, 3, “String C”);
insert(e, last(&list), &list);
print(&list);

assign_element_values(&e, 4, “String D”);
w = next(last(&list), &list);
insert(e, w, &list);
print(&list);
```

LIST: Array Implementation

```
    w = previous(w, &list);
    delete(w, &list);
    print(&list);

}
```

LIST: Array Implementation

- Key points:
  - *we have implemented all list manipulation operations with dedicated access functions*
  - *we never directly access the data-structure when using it but we always use the access functions*
  - *Why?*

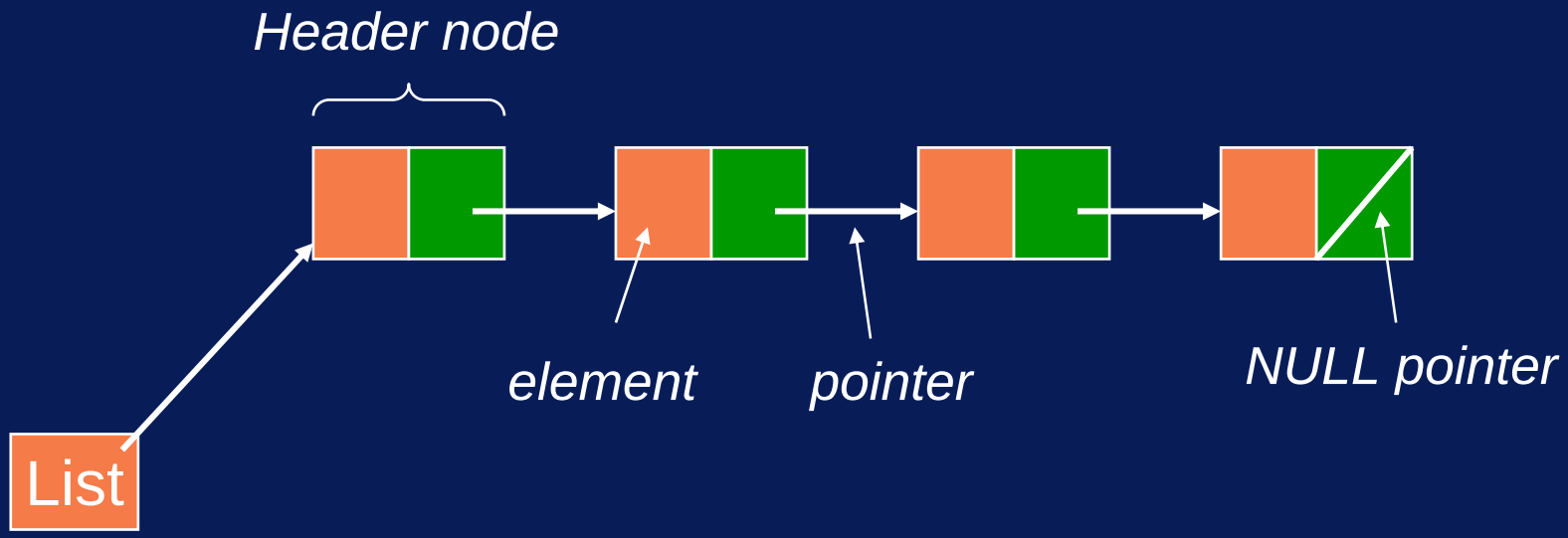
LIST: Array Implementation

- Key points:
  - *greater security: localized control and more resilient software maintenance*
  - *data hiding: the implementation of the data-structure is hidden from the user and so we can change the implementation and the user will never know*

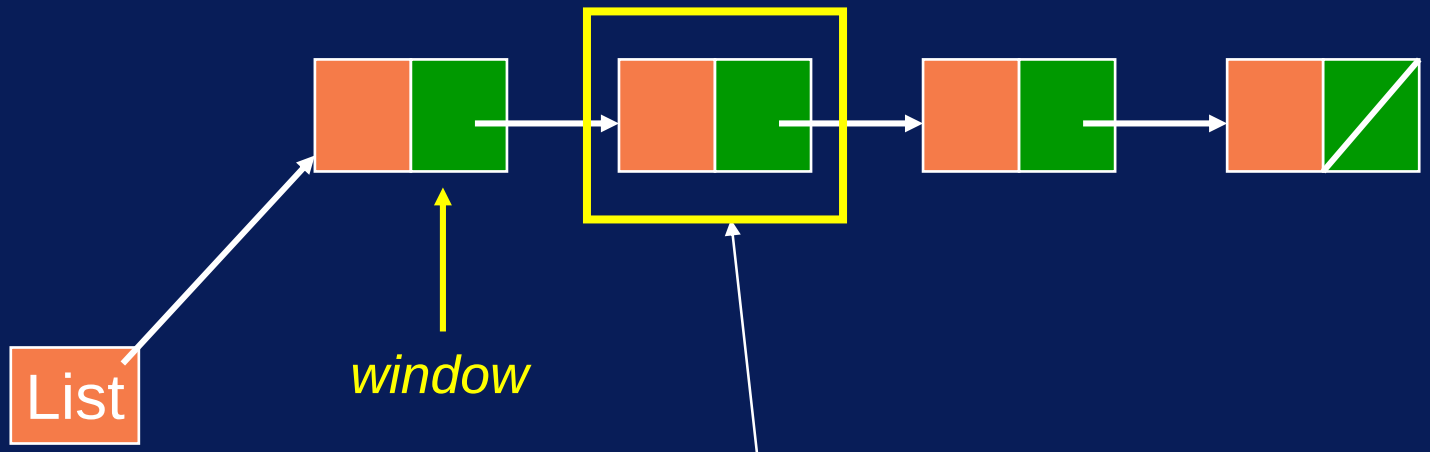
LIST: Array Implementation

- Possible problems with the implementation:
  - *have to shift elements when inserting and deleting (i.e. insert and delete are O(n))*
  - *have to specify the maximum size of the list at compile time*

LIST: Linked-List Implementation

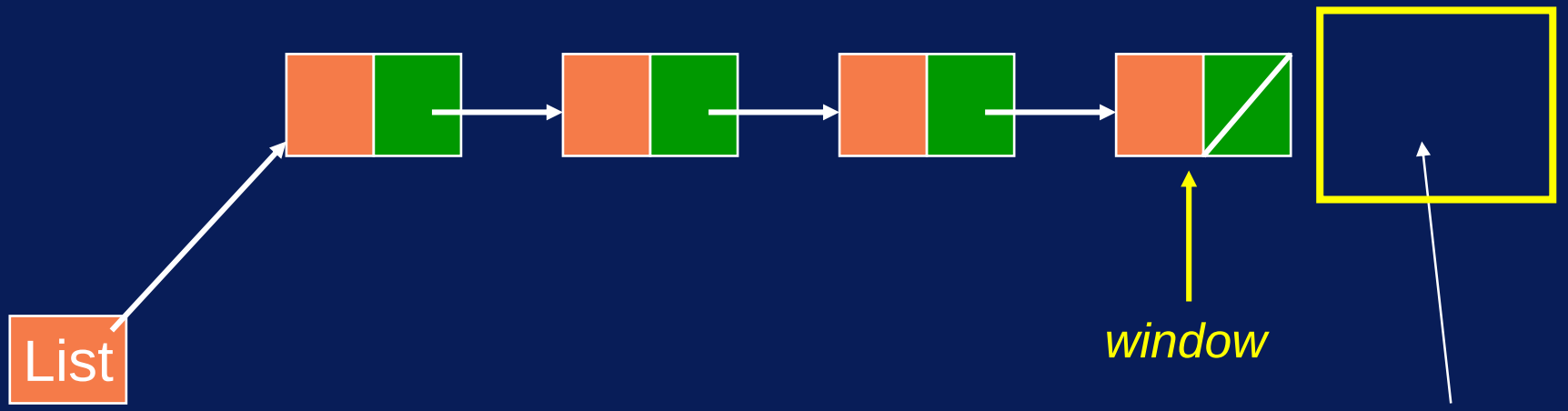


LIST: Linked-List Implementation



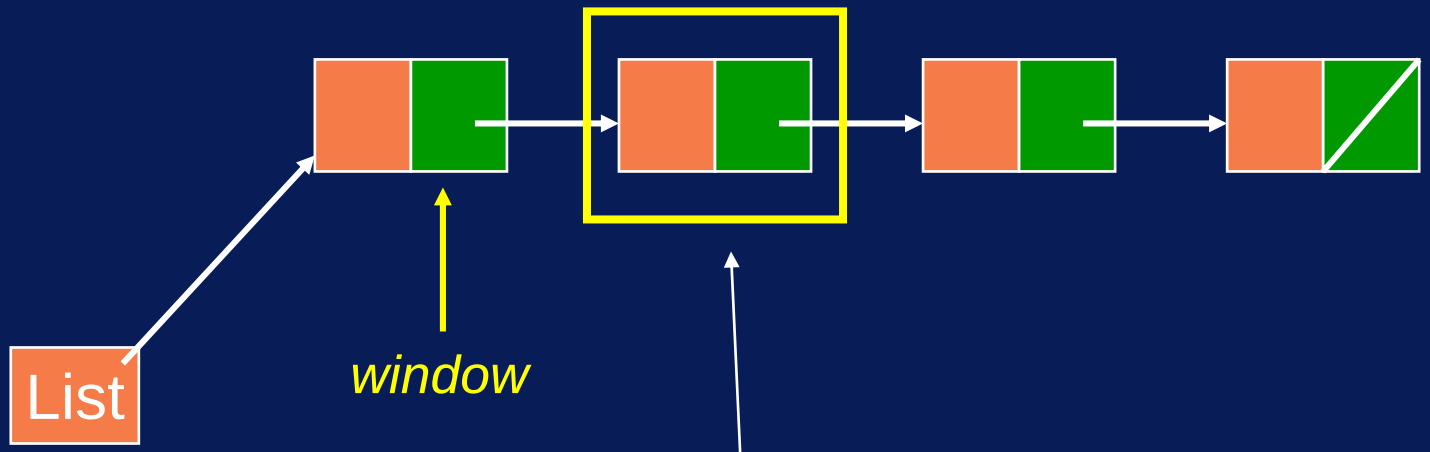
To place the window at this position we provide a link to the previous node (this is why we need a header node)

LIST: Linked-List Implementation



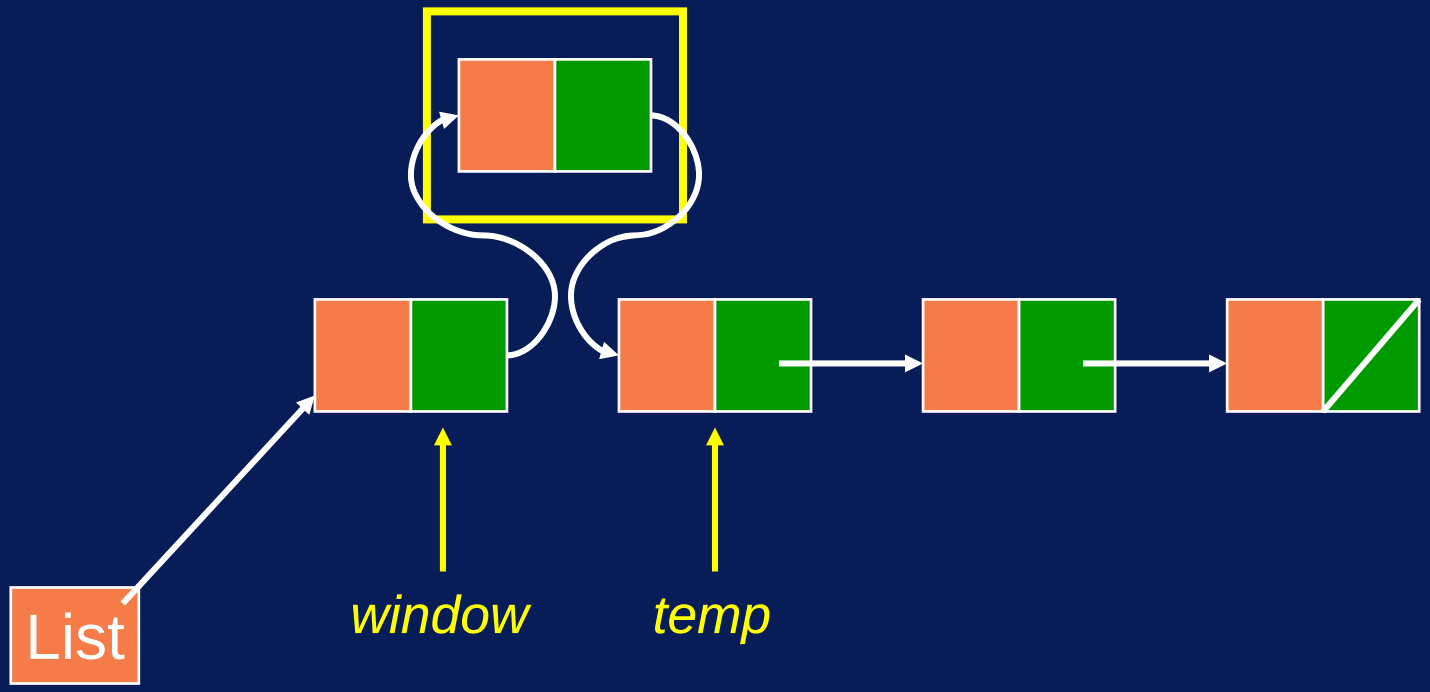
To place the window at end of the list we provide a link to the last node

LIST: Linked-List Implementation



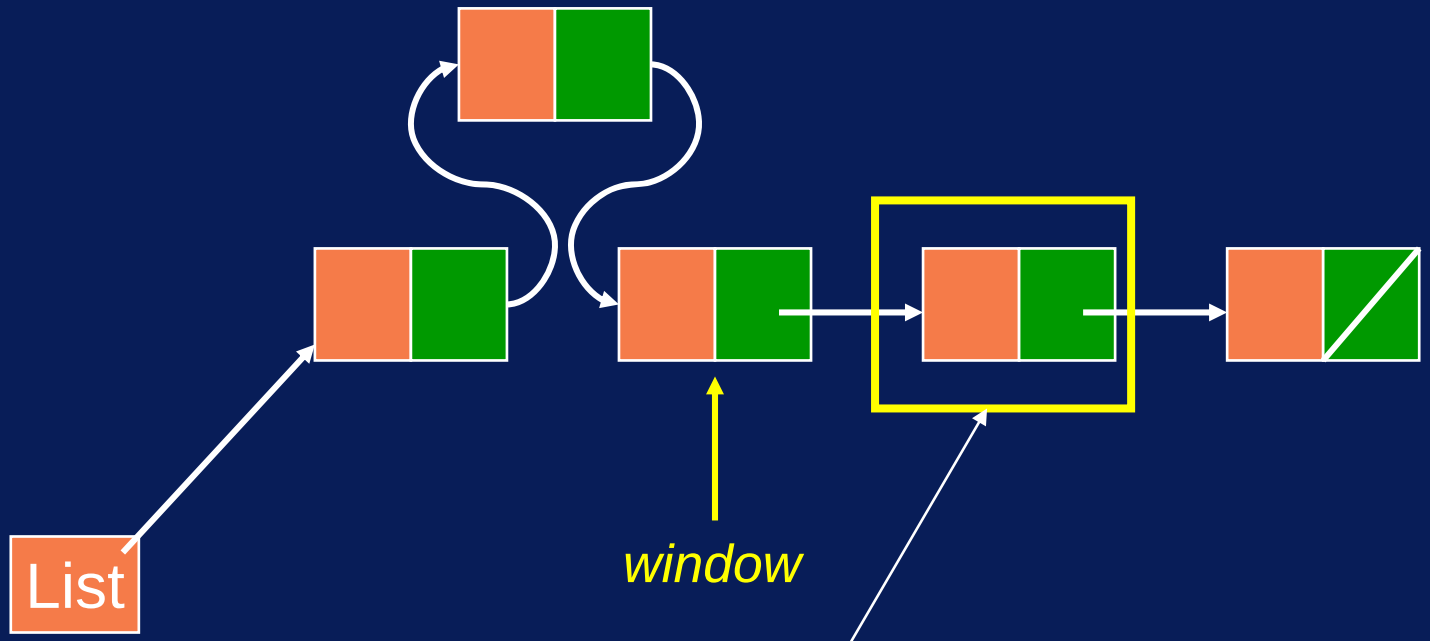
To insert a node at this window position
we create the node and re-arrange the links

LIST: Linked-List Implementation



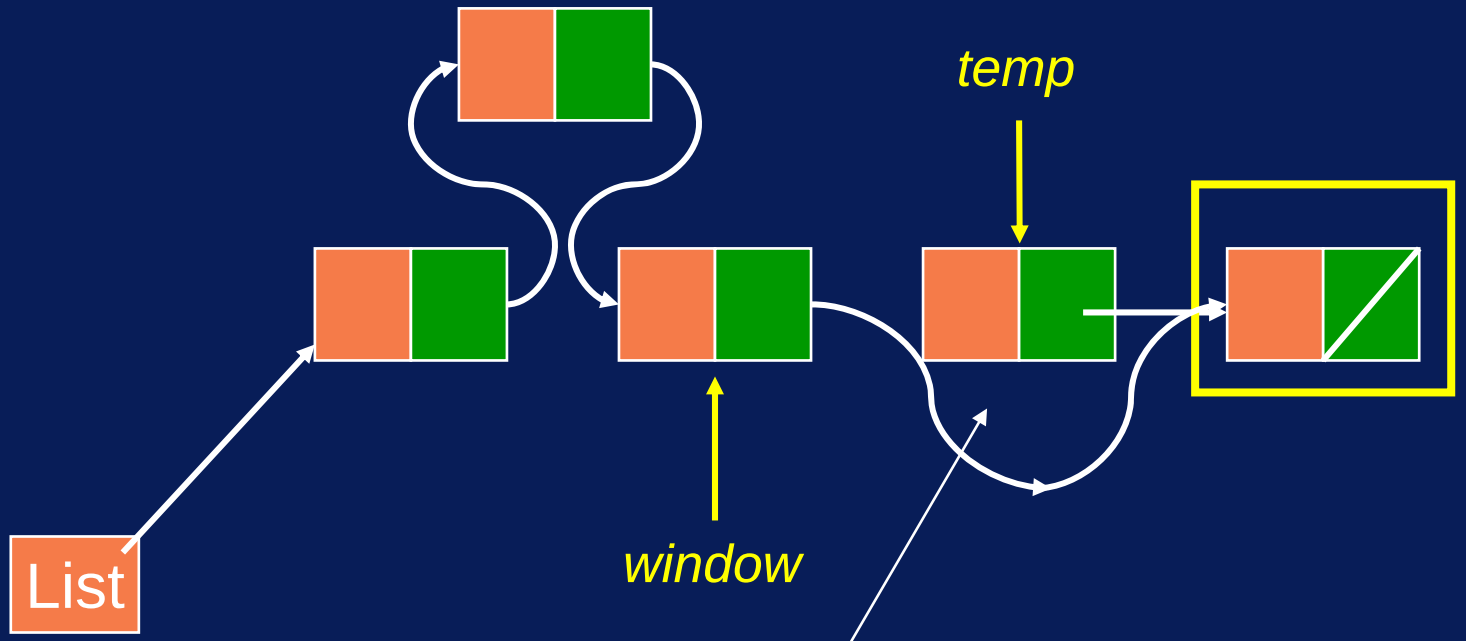
To insert a node at this window position
we create the node and re-arrange the links

LIST: Linked-List Implementation



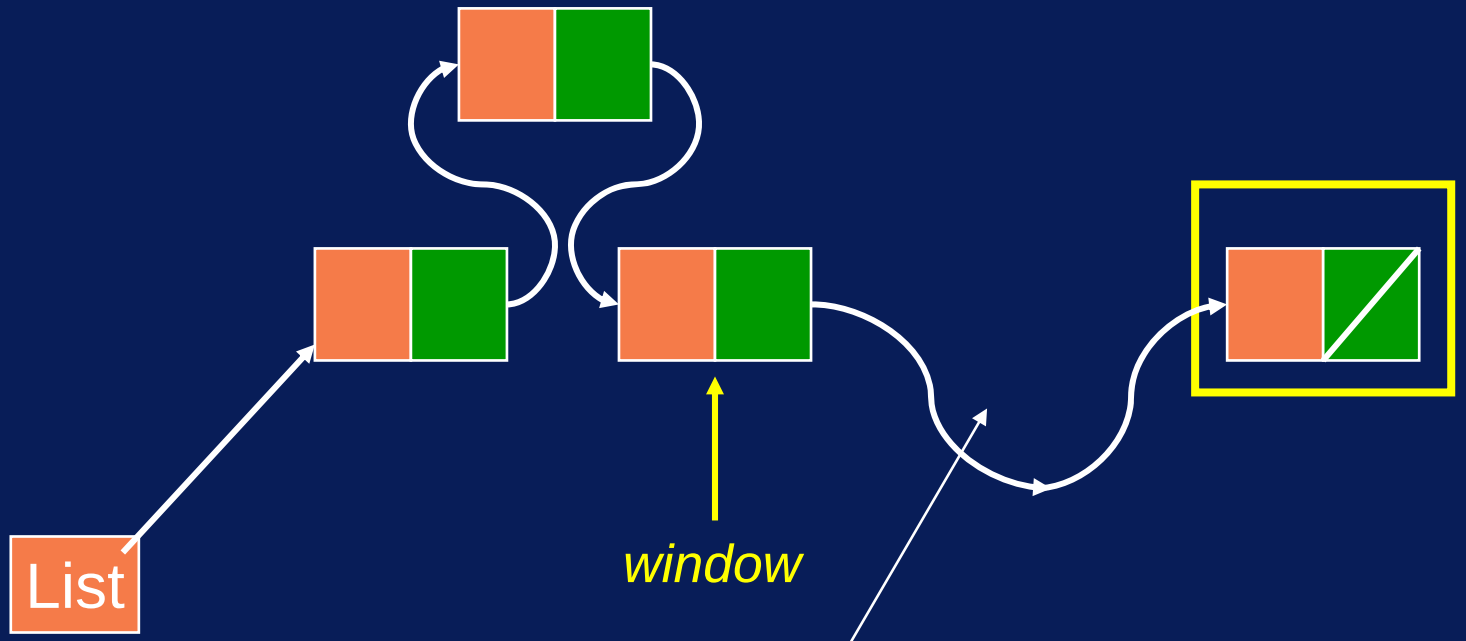
To delete a node at this window position we re-arrange the links and free the node

LIST: Linked-List Implementation



To delete a node at this window position we re-arrange the links and free the node

LIST: Linked-List Implementation



To delete a node at this window position we re-arrange the links and free the node

LIST: Linked-List Implementation

- type *elementtype*
- type *LIST*
- type *Boolean*
- type *windowtype*

LIST: Linked-List Implementation

```
/* linked-list implementation of LIST ADT */

#include <stdio.h>
#include <math.h>
#include <string.h>

#define FALSE 0
#define TRUE 1

typedef struct {
          int number;
          char *string;
        } ELEMENT_TYPE;
```

LIST: Linked-List Implementation

```
typedef struct node *NODE_TYPE;

typedef struct node{
          ELEMENT_TYPE element;
          NODE_TYPE next;
        } NODE;

typedef NODE_TYPE LIST_TYPE;
typedef NODE_TYPE WINDOW_TYPE;
```

LIST: Linked-List Implementation

```
/** position following last element in a list ***/

WINDOW_TYPE end(LIST_TYPE *list) {
   WINDOW_TYPE q;
   q = *list;
   if (q == NULL) {
      error("non-existent list");
   }
   else {
      while (q->next != NULL) {
         q = q->next;
      }
   }
   return(q);
}
```

LIST: Linked-List Implementation

```
/*** empty a list ***/

WINDOW_TYPE empty(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list != NULL) {
      /* list exists: delete all nodes including header */
      q = *list;
      while (q->next != NULL) {
         p = q;
         q = q->next;
         free(p);
      }
      free(q)
   }
   /* now, create a new empty one with a header node */
```

LIST: Linked-List Implementation

```
    /* now, create a new empty one with a header node */

    if ((q = (NODE_TYPE) malloc(sizeof(NODE))) == NULL)
       error("function empty: unable to allocate memory");
    else {
       q->next = NULL;
       *list = q;
    }
    return(end(list));
}
```

LIST: Linked-List Implementation

```
/*** test to see if a list is empty ***/

int is_empty(LIST_TYPE *list) {
   WINDOW_TYPE q;
   q = *list;
   if (q == NULL) {
      error("non-existent list");
   }
   else {
      if (q->next == NULL) {
         return(TRUE);
      else
         return(FALSE);
   }
}
```

LIST: Linked-List Implementation

```
/*** position at first element in a list ***/

WINDOW_TYPE first(LIST_TYPE *list) {
   if (is_empty(list) == FALSE) {
      return(*list);
   else
      return(end(list));
}
```

LIST: Linked-List Implementation

```
/*** position at next element in a list ***/

WINDOW_TYPE next(WINDOW_TYPE w, LIST_TYPE *list) {
   if (w == last(list)) {
      return(end(list));
   }
   else if (w == end(list)) {
      error("can't find next after end of list");
   }
   else {
      return(w->next);
   }
}
```

LIST: Linked-List Implementation

```
/*** position at previous element in a list ***/

WINDOW_TYPE previous(WINDOW_TYPE w, LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (w != first(list)) {
      p = first(list);
      while (p->next != w) {
         p = p->next;
         if (p == NULL) break; /* trap this to ensure   */
      }                        /* we don't dereference  */
      if (p != NULL)           /* a null pointer in the */
         return(p);            /* while condition       */
```

LIST: Linked-List Implementation

```
        else {
            error("can't find previous to a non-existent
    node");
        }
     }
     else {
        error("can't find previous before first element of
    list");
        return(w);
     }
}
```

LIST: Linked-List Implementation

```
/*** position at last element in a list ***/

WINDOW_TYPE last(LIST_TYPE *list) {
   WINDOW_TYPE p, q;
   if (*list == NULL) {
      error("non-existent list");
   }
   else {
      /* list exists: find last node */
```

LIST: Linked-List Implementation

```
        /* list exists: find last node */

        if (is_empty(list)) {
           p = end(list);
        }
        else {
           p = *list;
           q = p=>next;
           while (q->next != NULL) {
              p = q;
              q = q->next;
           }
        }
        return(p);
}  }
```

LIST: Linked-List Implementation

```
/*** insert an element in a list ***/

LIST_TYPE *insert(ELEMENT_TYPE e, WINDOW_TYPE w,
                  LIST_TYPE *list) {
   WINDOW_TYPE temp;
   if (*list == NULL) {
      error("cannot insert in a non-existent list");
   }
```

LIST: Linked-List Implementation

```
    else {
        /* insert it after w */
        temp = w->next;
        if ((w->next = (NODE_TYPE) malloc(sizeof(NODE))) =
    NULL)
            error("function insert: unable to allocate
    memory");
        else {
            w->next->element = e;
            w->next->next = temp;
        }
        return(list);
    }
}
```

LIST: Linked-List Implementation

```
/*** delete an element from a list ***/

LIST_TYPE *delete(WINDOW_TYPE w, LIST_TYPE *list) {
   WINDOW_TYPE p;
   if (*list == NULL) {
      error("cannot delete from a non-existent list");
   }
   else {
      p = w->next; /* node to be deleted */
      w->next = w->next->next;  /* rearrange the links */
      free(p);     /* delete the node */
      return(list);
   }
}
```

LIST: Linked-List Implementation

```
/*** retrieve an element from a list ***/

ELEMENT_TYPE retrieve(WINDOW_TYPE w, LIST_TYPE *list) {
   WINDOW_TYPE p;

   if (*list == NULL) {
      error("cannot retrieve from a non-existent list");
   }
   else {
      return(w->next->element);
   }
}
```

LIST: Linked-List Implementation

```
/*** print all elements in a list ***/

int print(LIST_TYPE *list) {
   WINDOW_TYPE w;
   ELEMENT_TYPE e;

   printf("Contents of list: \n");
   w = first(list);
   while (w != end(list)) {
      printf("%d %s\n", e.number, e.string);
      w = next(w, list);
   }
   printf("---\n");
   return(0);
}
```

LIST: Linked-List Implementation

```
/*** error handler: print message passed as argument and
     take appropriate action                              ***/
int error(char *s); {
   printf("Error: %s\n", s);
   exit(0);
}

/*** assign values to an element ***/

int assign_element_values(ELEMENT_TYPE *e, int number,
  char s[]) {
   e->string = (char *) malloc(sizeof(char) * strlen(s));
   strcpy(e->string, s);
   e->number = number;
}
```

LIST: Linked-List Implementation

```
/*** main driver routine ***/

   WINDOW_TYPE w;
   ELEMEN_TYPE e;
   LIST_TYPE list;
   int i;

   empty(&list);
   print(&list);

   assign_element_values(&e, 1, “String A”);
   w = first(&list);
   insert(e, w, &list);
   print(&list);
```

LIST: Linked-List Implementation

```
assign_element_values(&e, 2, “String B”);
insert(e, w, &list);
print(&list);

assign_element_values(&e, 3, “String C”);
insert(e, last(&list), &list);
print(&list);

assign_element_values(&e, 4, “String D”);
w = next(last(&list), &list);
insert(e, w, &list);
print(&list);
```

LIST: Linked-List Implementation

```
    w = previous(w, &list);
    delete(w, &list);
    print(&list);

}
```

LIST: Linked-List Implementation

- Key points:
  - *All we changed was the implementation of the data-structure and the access routines*
  - *But by keeping the interface to the access routines the same as before, these changes are transparent to the user*
  - *And we didn't have to make any changes in the main function which was actually manipulating the list*

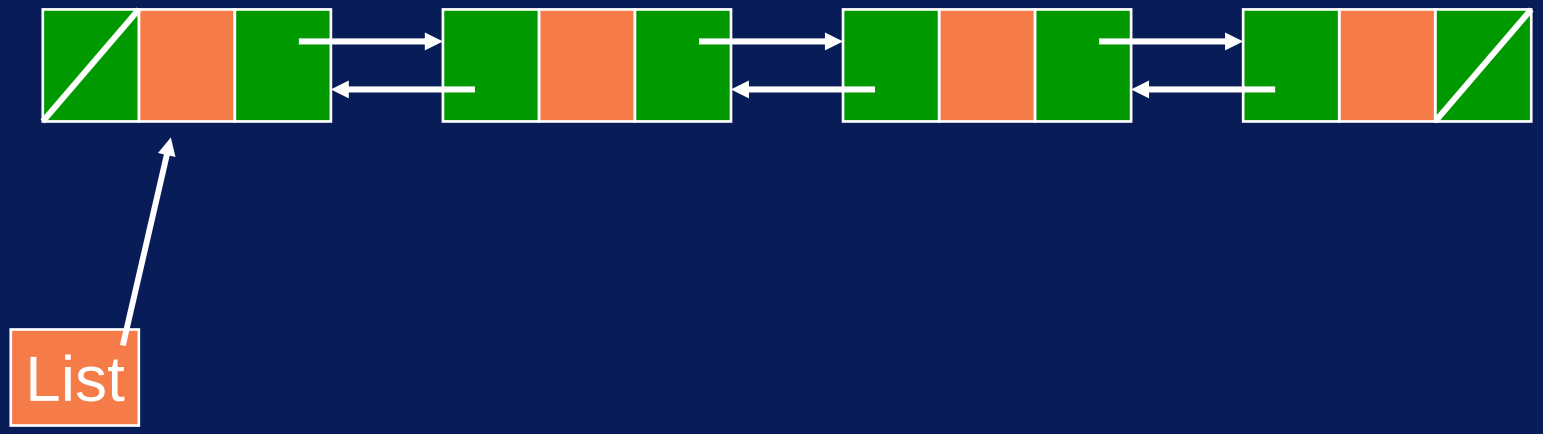
LIST: Linked-List Implementation

- Key points:
  - *In a real software system where perhaps hundreds (or thousands) of people are using these list primitives, this transparency is critical*
  - *We couldn't have achieved it if we manipulated the data-structure directly*

LIST: Linked-List Implementation

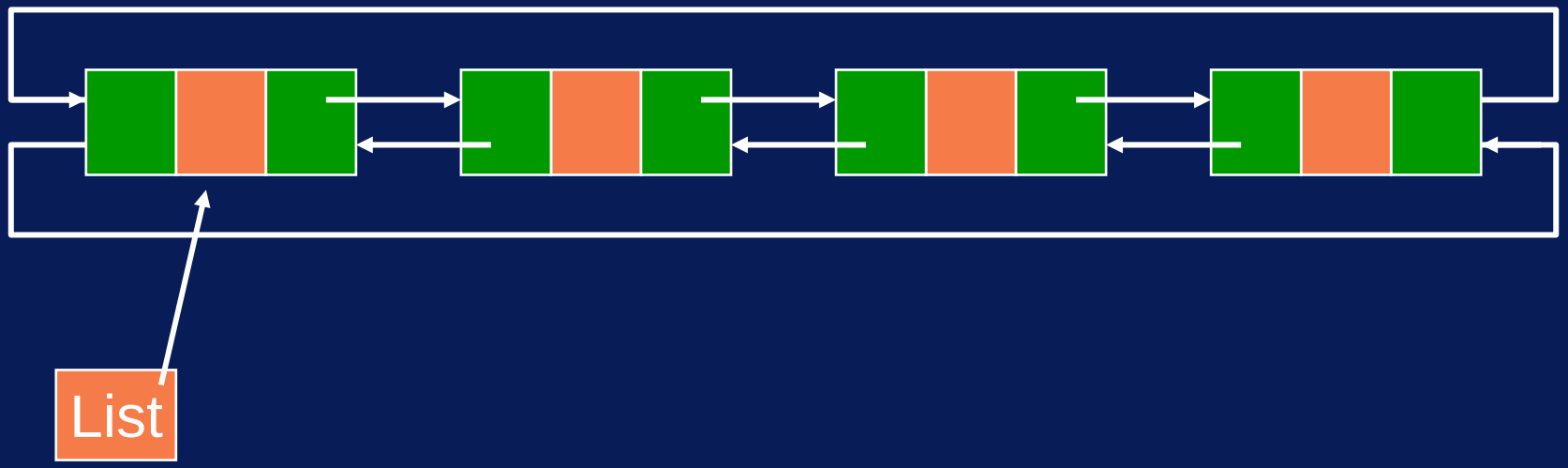
- Possible problems with the implementation:
  - *we have to run the length of the list in order to find the end (i.e. end(L) is O(n))*
  - *there is a (small) overhead in using the pointers*
- *On the other hand, the list can now grow as large as necessary, without having to predefine the maximum size*

LIST: Linked-List Implementation



We can also have a doubly-linked list;
this removes the need to have a header node
and make finding the previous node more efficient

LIST: Linked-List Implementation



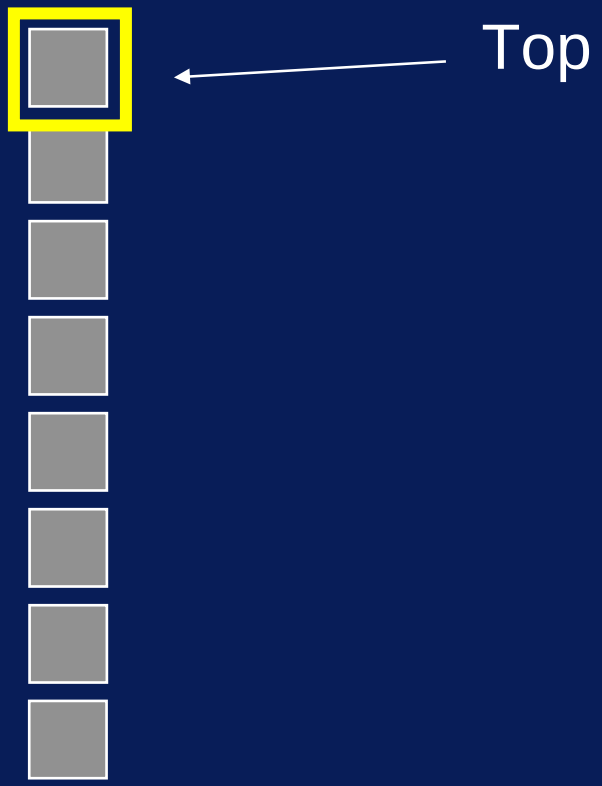
Lists can also be circular

Stacks

Stacks

- A stack is a special type of list
  - all insertions and deletions take place at one end, called the top
  - thus, the last one added is always the first one available for deletion
  - also referred to as
    - » pushdown stack
    - » pushdown list
    - » LIFO list (Last In First Out)

Stacks



Stack Operations

- *Declare*: $\rightarrow$ S :

  The function value of *Declare(S)* is an empty stack

Stack Operations

- *Empty*: $\rightarrow$ S :

  The function *Empty* causes the stack to be emptied and it returns position *End(S)*

Stack Operations

- *IsEmpty*: S $\rightarrow$ B :

  The function value *IsEmpty*(*S*) is *true* if *S* is empty; otherwise it is *false*

Stack Operations

- *Top*: S $\rightarrow$ E :

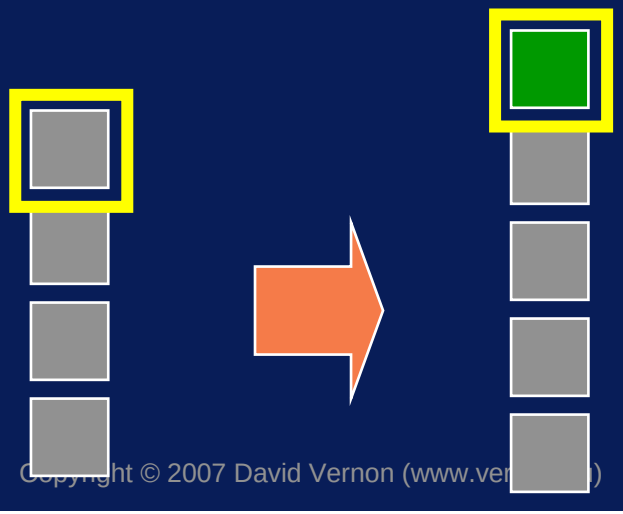
  The function value *Top*(*S*) is the first element in the list;

  if the list is empty, the value is undefined

Stack Operations

- *Push*: $E \times S \rightarrow L$:

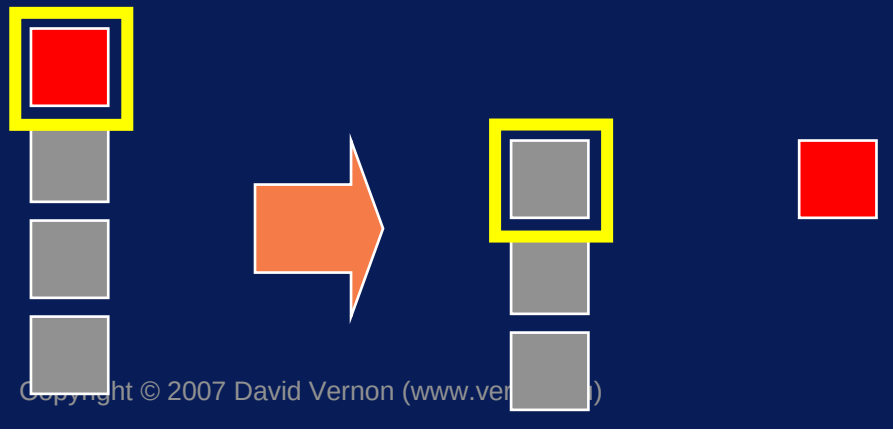
  Push(e, S)
  Insert an element *e* at the top of the stack



Stack Operations

- *Pop*: S $\rightarrow$ E :

  Pop(S)
  Remove the top element from the stack: i.e. return the top element and delete it from the stack



Stack Operations

- *All these operations can be directly implemented using the LIST ADT operations on a List S*
- *Although it may be more efficient to use a dedicated implementation*
- *It depends what you want: code efficiency or software re-use (i.e. utilization efficiency)*

Stack Operations

- Declare(S)
- Empty(S)
- Top(S)
  - » Retrieve(First(S), S)
- Push(e, S)
  - » Insert(e, First(S), S)
- Pop(S)
  - » Retrieve(First(S), S)
  - » Delete(First(S), S)

Queues

Queues

- A queue is another special type of list
  - insertions are made at one end, called the tail of the queue
  - deletions take place at the other end, called the head
  - thus, the last one added is always the last one available for deletion
  - also referred to as
    - » FiFO list (First In First Out)

Queues

Queue Operations

- *Declare*: $\rightarrow$ Q :

  The function value of *Declare(Q)* is an empty queue

Queue Operations

- *Empty*: $\rightarrow$ Q :

  The function *Empty* causes the queue to be emptied and it returns position *End(Q)*

Queue Operations

- *IsEmpty*: Q $\rightarrow$ B :

  The function value *IsEmpty*(*Q*) is *true* if *Q* is empty; otherwise it is *false*

Queue Operations

- *Head*: Q $\rightarrow$ E :

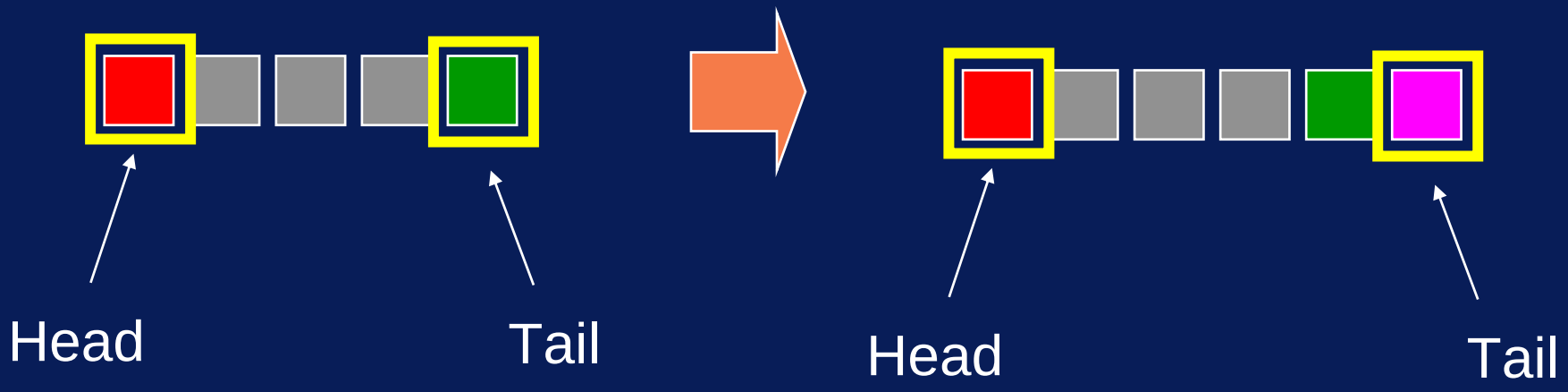
  The function value *Head*(*Q*) is the first element in the list;

  if the queue is empty, the value is undefined

Queue Operations

- *Enqueue*: $E \times Q \rightarrow Q$:

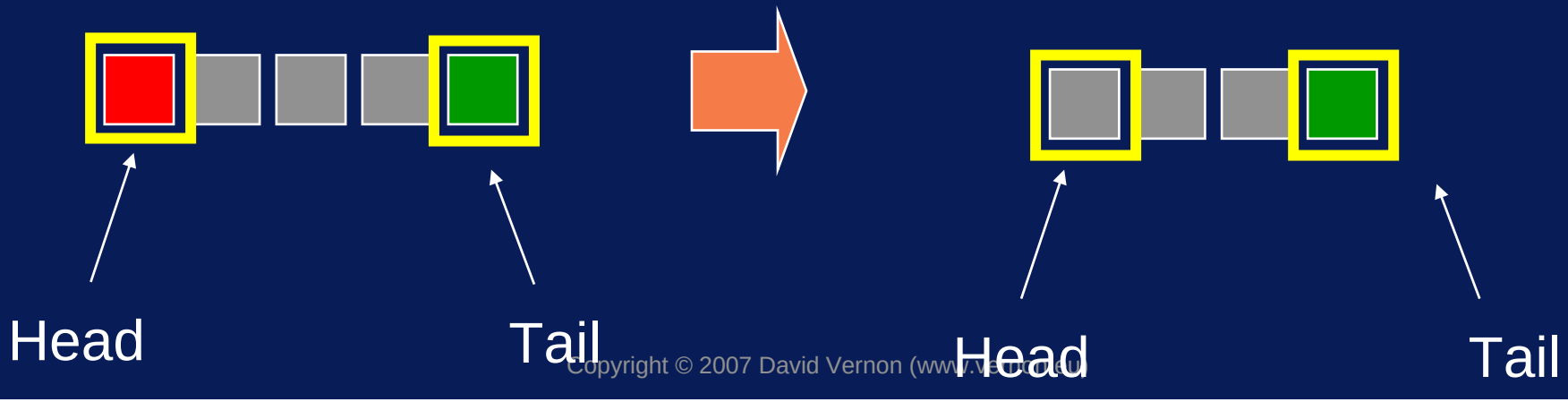
  Enqueue(e, Q)
  Add an element *e* the the tail of the queue



Queue Operations

- *Dequeue*: Q $\rightarrow$ E :

  Dequeue(Q)
  Remove the element from the head of the queue: i.e. return the first element and delete it from the queue



Queue Operations

- *All these operations can be directly implemented using the LIST ADT operations on a queue Q*
- *Again, it may be more efficient to use a dedicated implementation*
- *And, again, it depends what you want: code efficiency or software re-use (i.e. utilization efficiency)*

Queue Operations

- Declare(Q)
- Empty(Q)
- Head(Q)
  - » Retrieve(First(Q), Q)
- Enqueue(e, Q)
  - » Insert(e, End(Q), Q)
- Dequeque(Q)
  - » Retrieve(First(Q), Q)
  - » Delete(First(Q), Q)

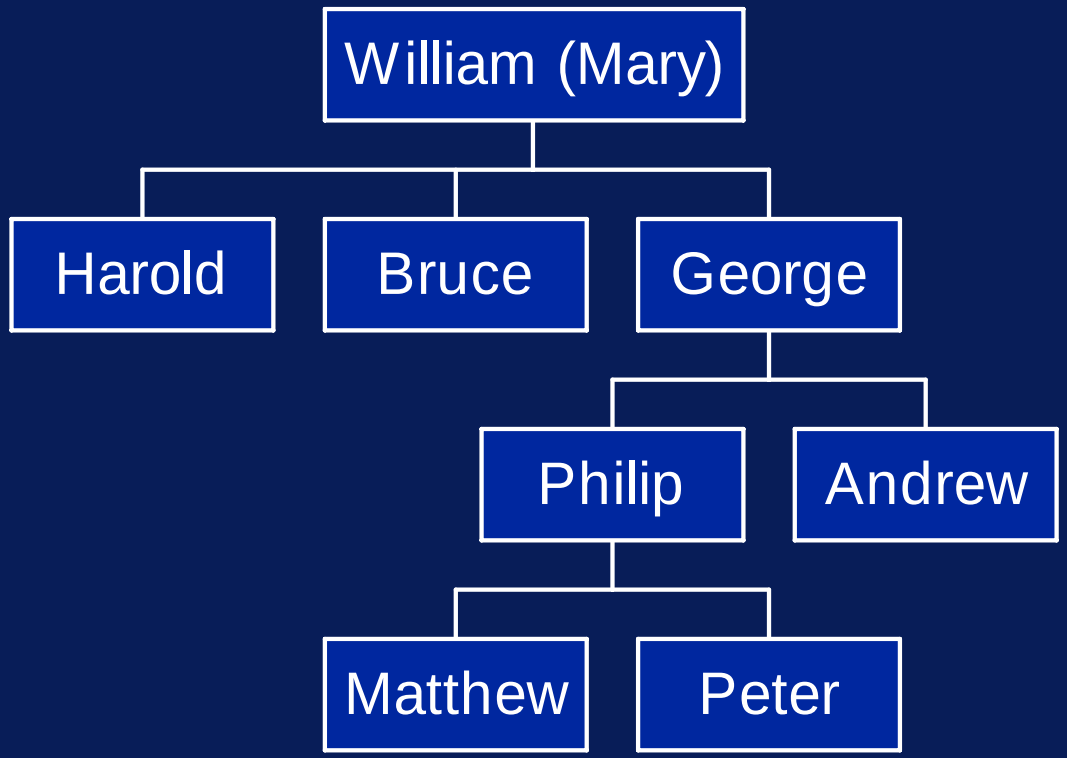
Trees

Trees

- Trees are everywhere
- Hierarchical method of structuring data
- Uses of trees:
  - genealogical tree
  - organizational tree
  - expression tree
  - binary search tree
  - decision tree

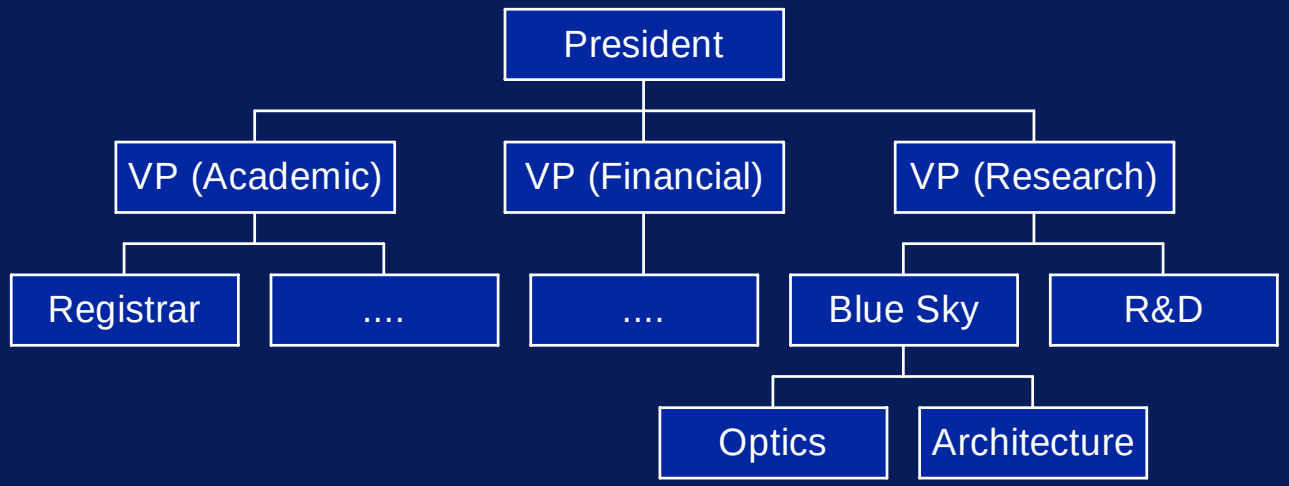
Uses of Trees

Genealogical Tree



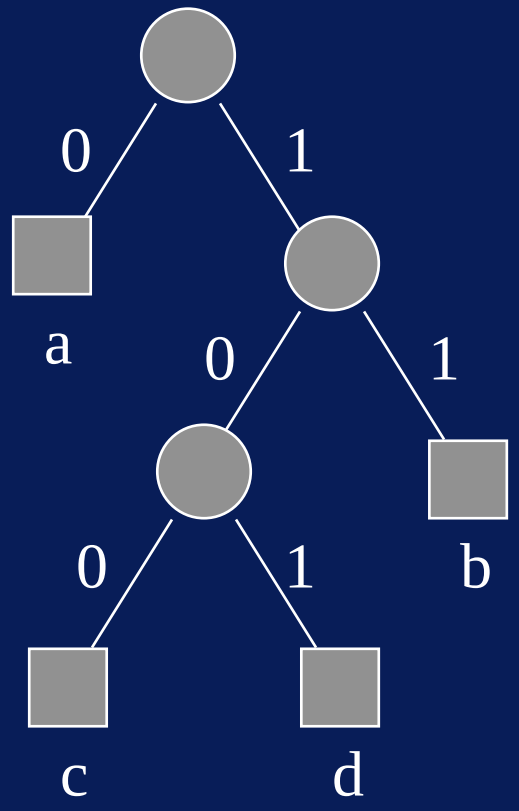
Uses of Trees

Organization Tree



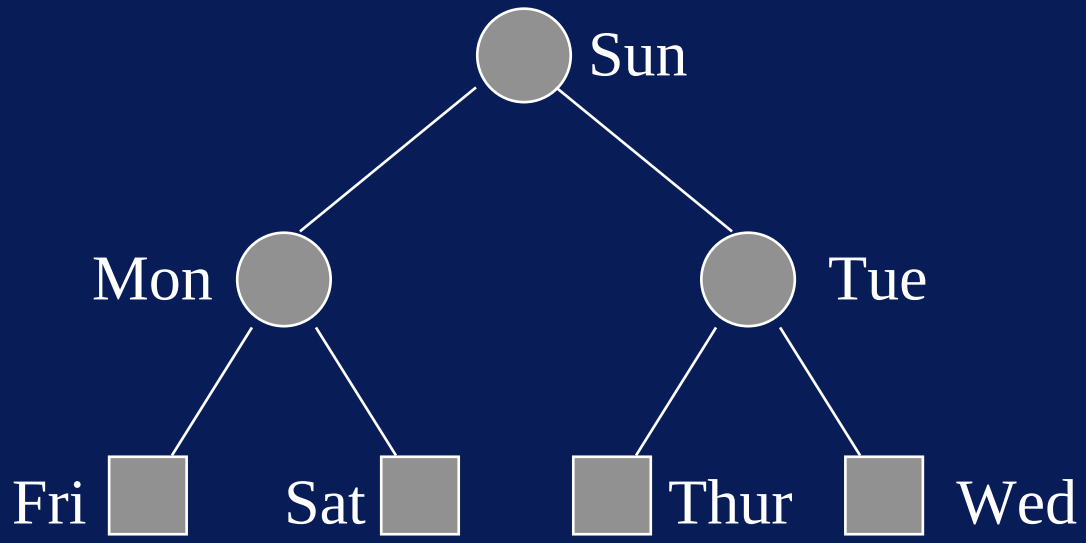
Uses of Trees

Code Tree



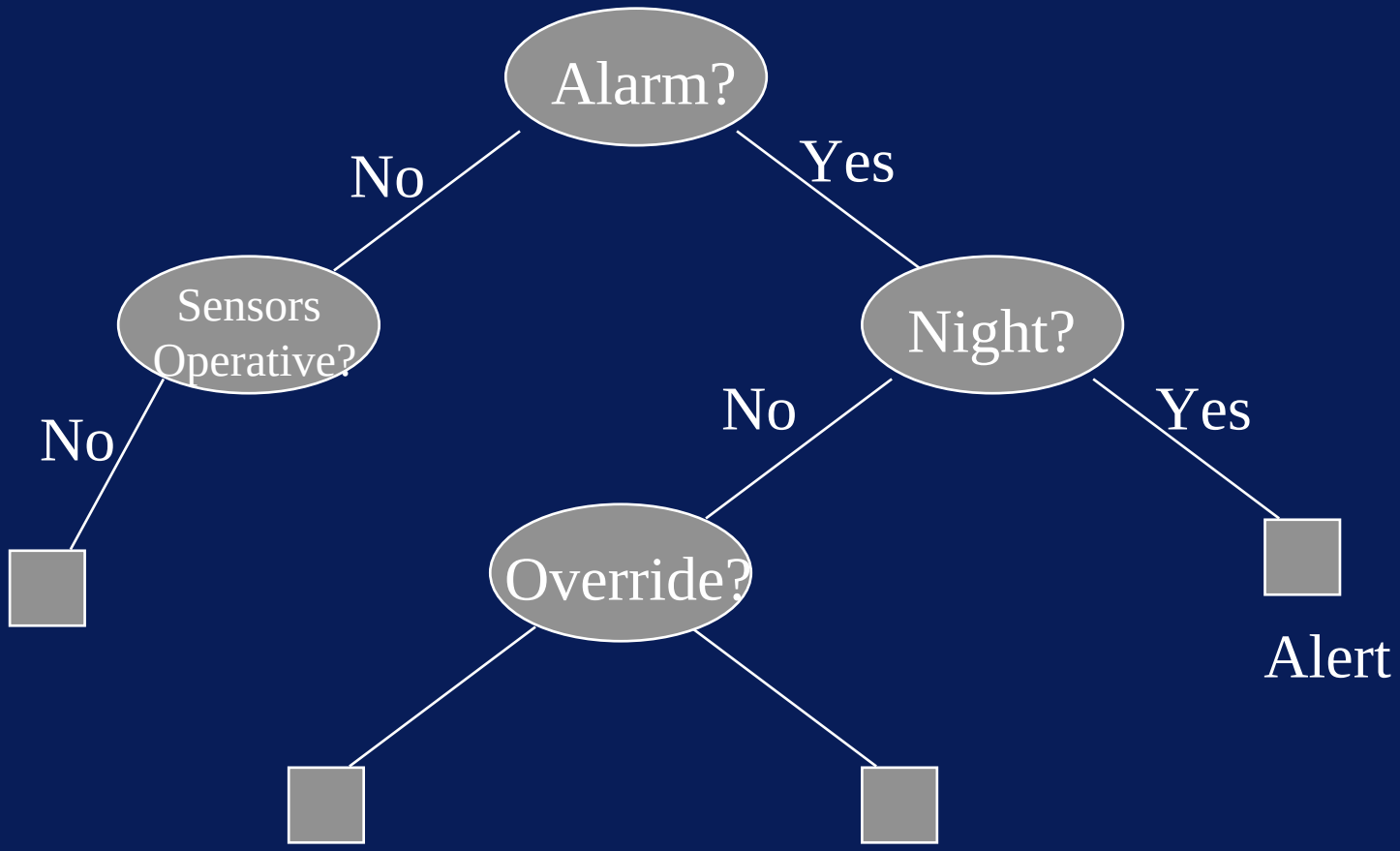
Uses of Trees

Binary Seach Tree



Uses of Trees

Decision Tree



Trees

- Fundamentals
- Traversals
- Display
- Representation
- Abstract Data Type (ADT) approach
- Emphasis on binary tree
- Also mention multi-way trees, forests, orchards

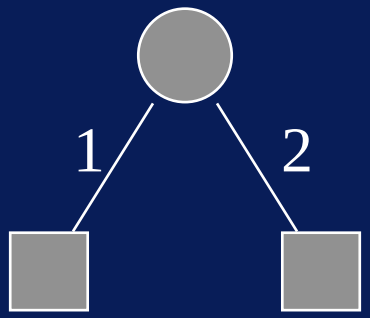
Tree Definitions

- A binary tree T of n nodes, $n \geq 0$,
  - either is empty, if $n = 0$,
  - or consists of a root node u and two binary trees $u(1)$ and $u(2)$ of n_1 and n_2 nodes, respectively, such that
    $n = n + n_1 + n_2$.
- We say that $u(1)$ is the first or left subtree of T, and $u(2)$ is the second or right subtree of T.

Binary Tree

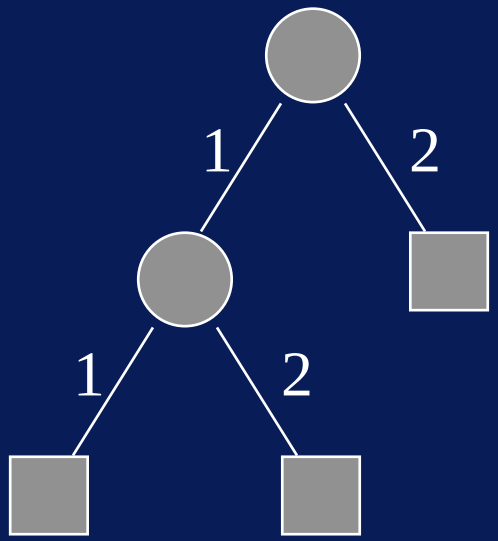
Binary Tree of zero nodes

Binary Tree



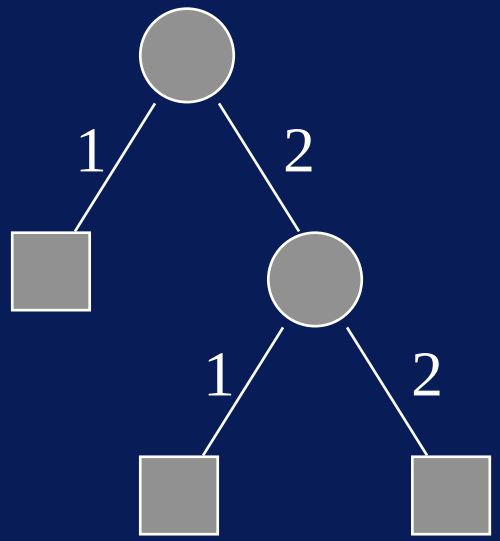
Binary Tree of one node

Binary Tree



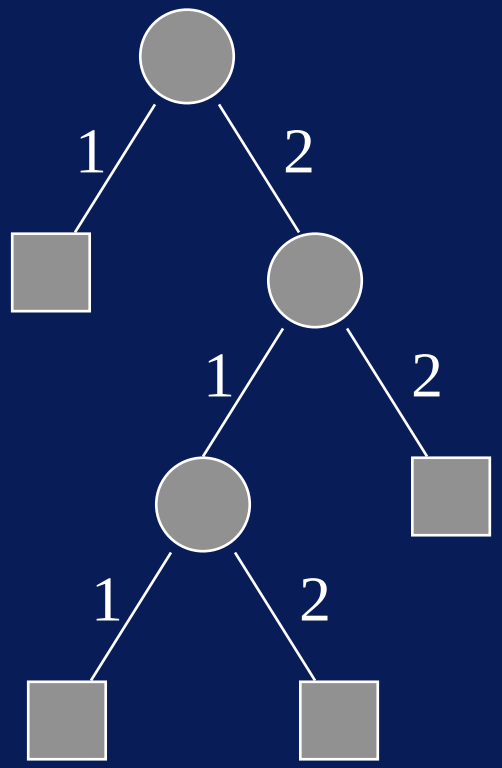
Binary Tree of two nodes

Binary Tree



Binary Tree of two nodes

Binary Tree



Binary Tree of three nodes

Binary Tree

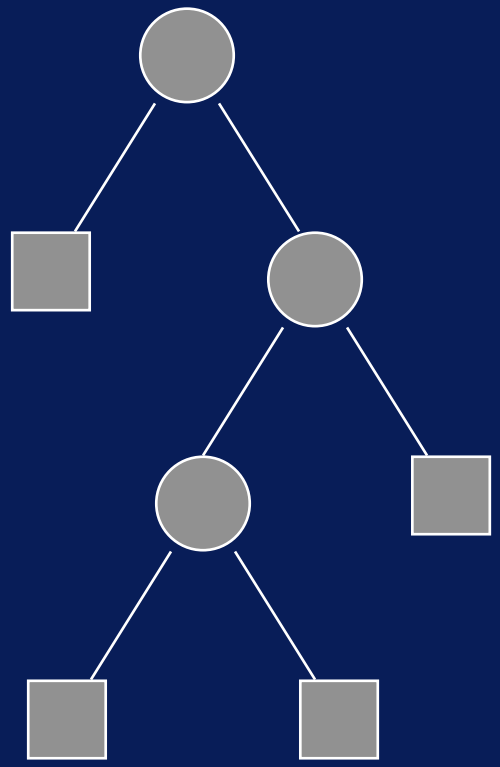
External nodes - have no subtrees

Internal nodes - always have two subtrees

Binary Tree Terminology

- Let *T* be a binary tree with root *u*
- Let *v* be any node in *T*
- If *v* is the root of either *u*(1) or *u*(2), then we say *u* is the parent of *v*, and that *v* is the child of *u*
- If *w* is also a child of *u*, and *w* is distinct from *v*, we say that *v* and *w* are siblings.

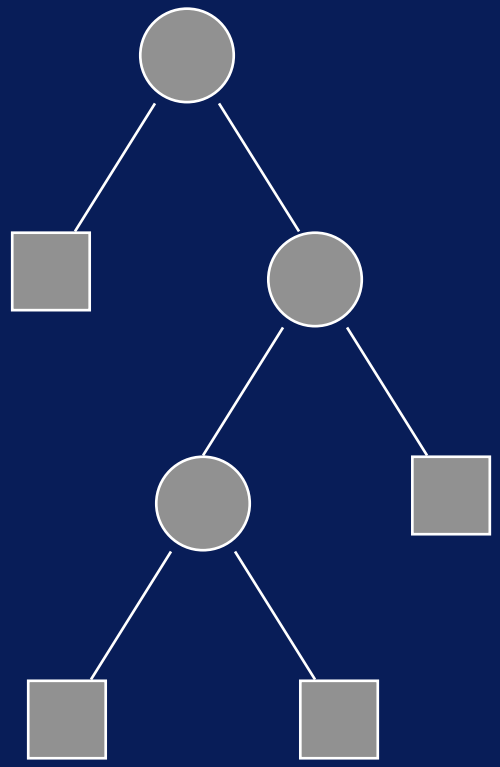
Binary Tree



Binary Tree Terminology

- If *v* is the root of *u*(*i*),
- then *v* is the *i*th child of *u*; *u*(1) is the left child and *u*(2) is the right child.
- Also have grandparents and grandchildren

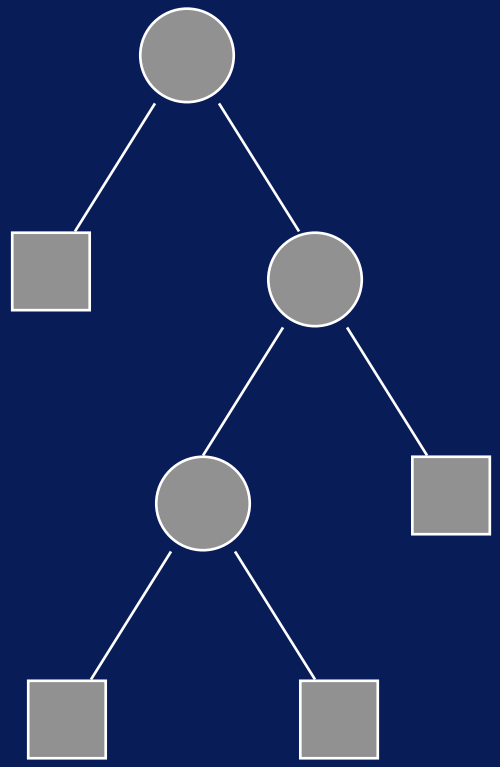
Binary Tree



Binary Tree Terminology

- Given a binary tree T of n nodes, $n \geq 0$,
- then *v* is a descendent of *u* if either
  - *v* is equal to *u*
    or
  - *v* is a child of some node *w* and *w* is a descendant of *u*.
- We write $v \; desc_T \; u$.
- v is a proper descendent of *u* if *v* is a descendant of *u* and $v \neq u$.

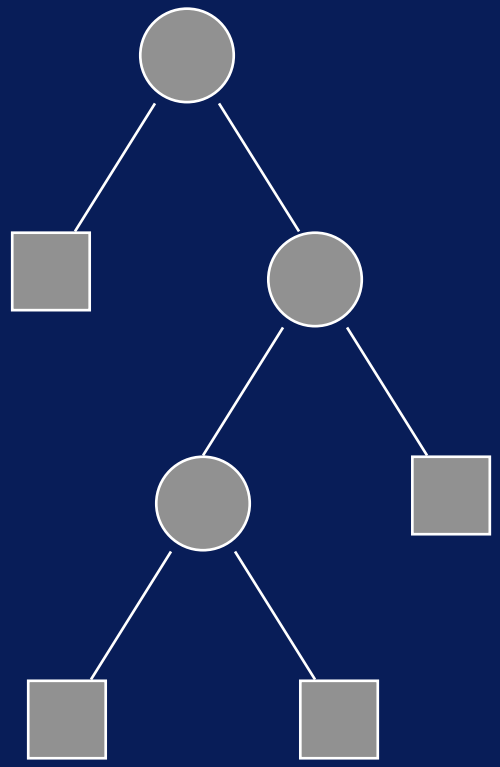
Binary Tree



Binary Tree Terminology

- Given a binary tree T of n nodes, $n \geq 0$,
- then *v* is a left descendent of *u* if either
  - *v* is equal to *u*
    or
  - *v* is a left child of some node *w* and *w* is a left descendant of *u*.
- We write $v \; ldesc_T \; u$.
- Similarly we have $v \; rdesc_T \; u$.

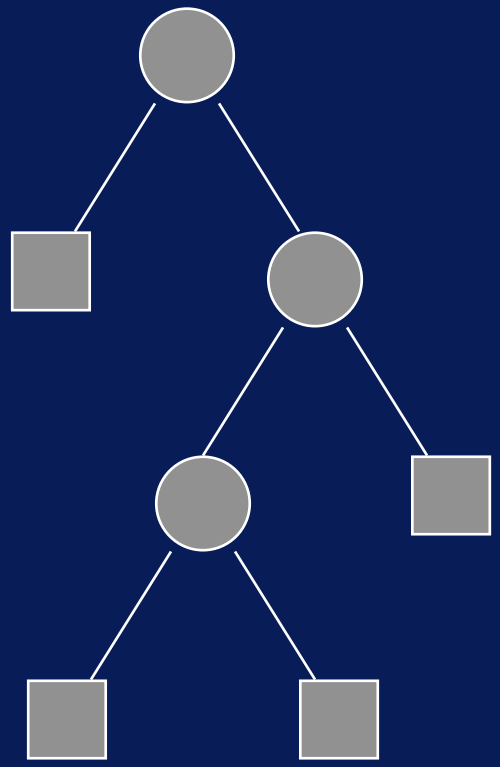
Binary Tree



Binary Tree Terminology

- $ldesc_T$ relates nodes across a binary tree rather than up and down a binary tree.
- Given two nodes *u* and *v* in a binary tree *T*, we say that *v* is to the left of *u* if there is a new node *w* in *T* such that *v* is a left descendant of *w* and *u* is a right descendant of *w*.
- We denote this relation by $left_T$ and write $v\ left_T\ u$.

Binary Tree



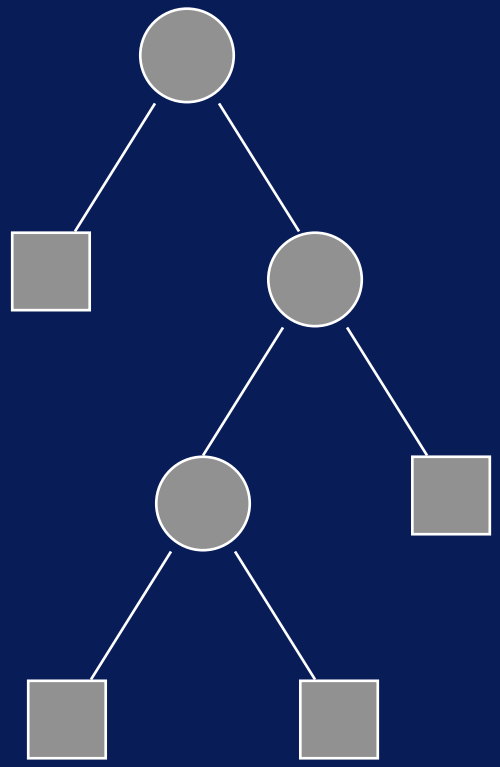
Binary Tree Terminology

- The external nodes of a tree define its frontier
- We can count the number of nodes in a binary tree in three ways:
  - Number of internal nodes
  - Number of external nodes
  - Number of internal and external nodes
- The number of internal nodes is the size of the tree

Binary Tree Terminology

- Let T be a binary tree of size n , $n \geq 0$,
- Then, the number of external nodes of T is
  $n + 1$

Binary Tree



Binary Tree Terminology

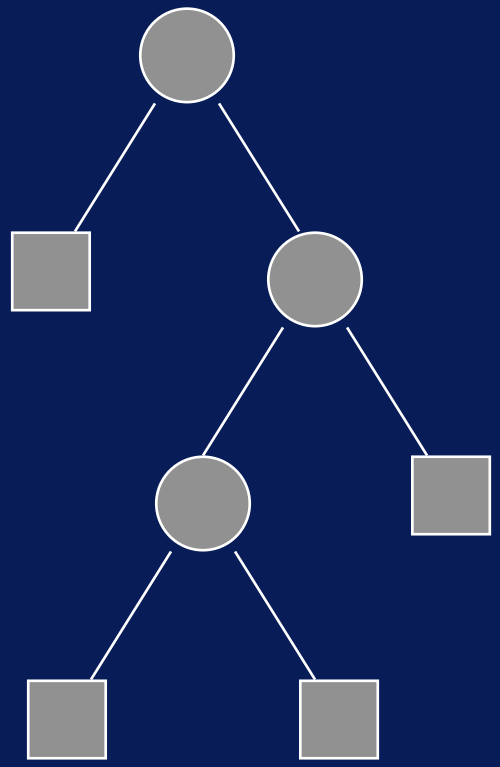
- The height of T is defined recursively as

  0 if T is empty and

  1 + $max(height(T_1), height(T_2))$ otherwise, where T_1 and T_2 are the subtrees of the root.

- The height of a tree is the length of a longest chain of descendents

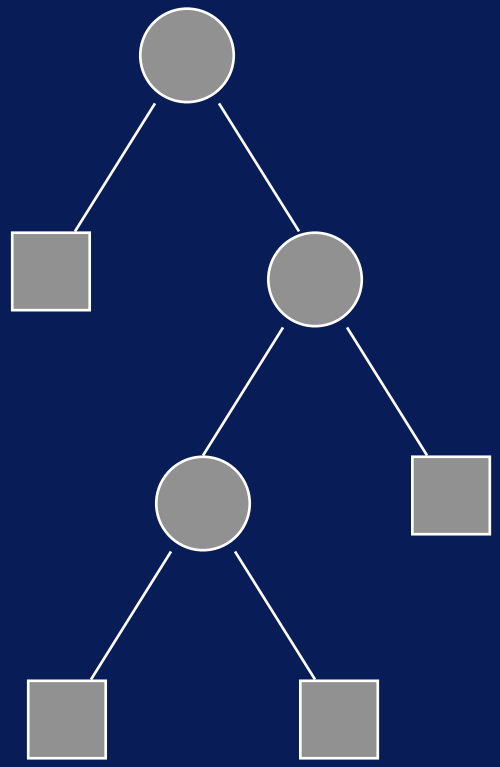
Binary Tree



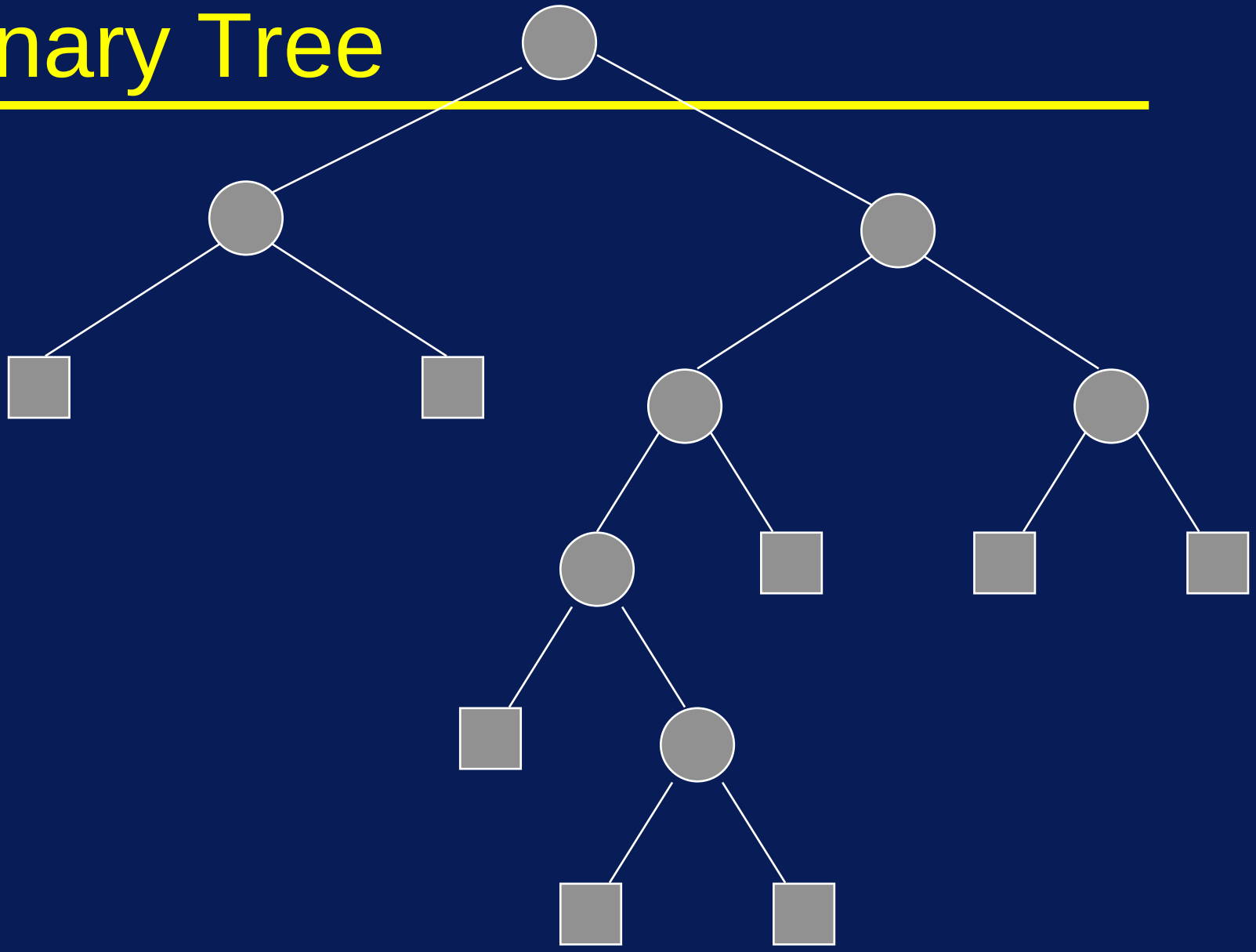
Binary Tree Terminology

- Height Numbering
  - Number all external nodes 0
  - Number each internal node to be one more than the maximum of the numbers of its children
  - Then the number of the root is the height of *T*
- The height of a node *u* in *T* is the height of the subtree rooted at *u*

Binary Tree



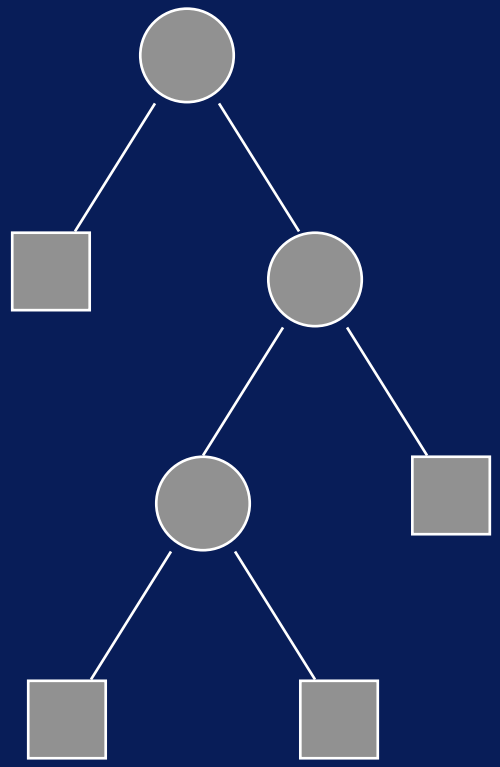
Binary Tree



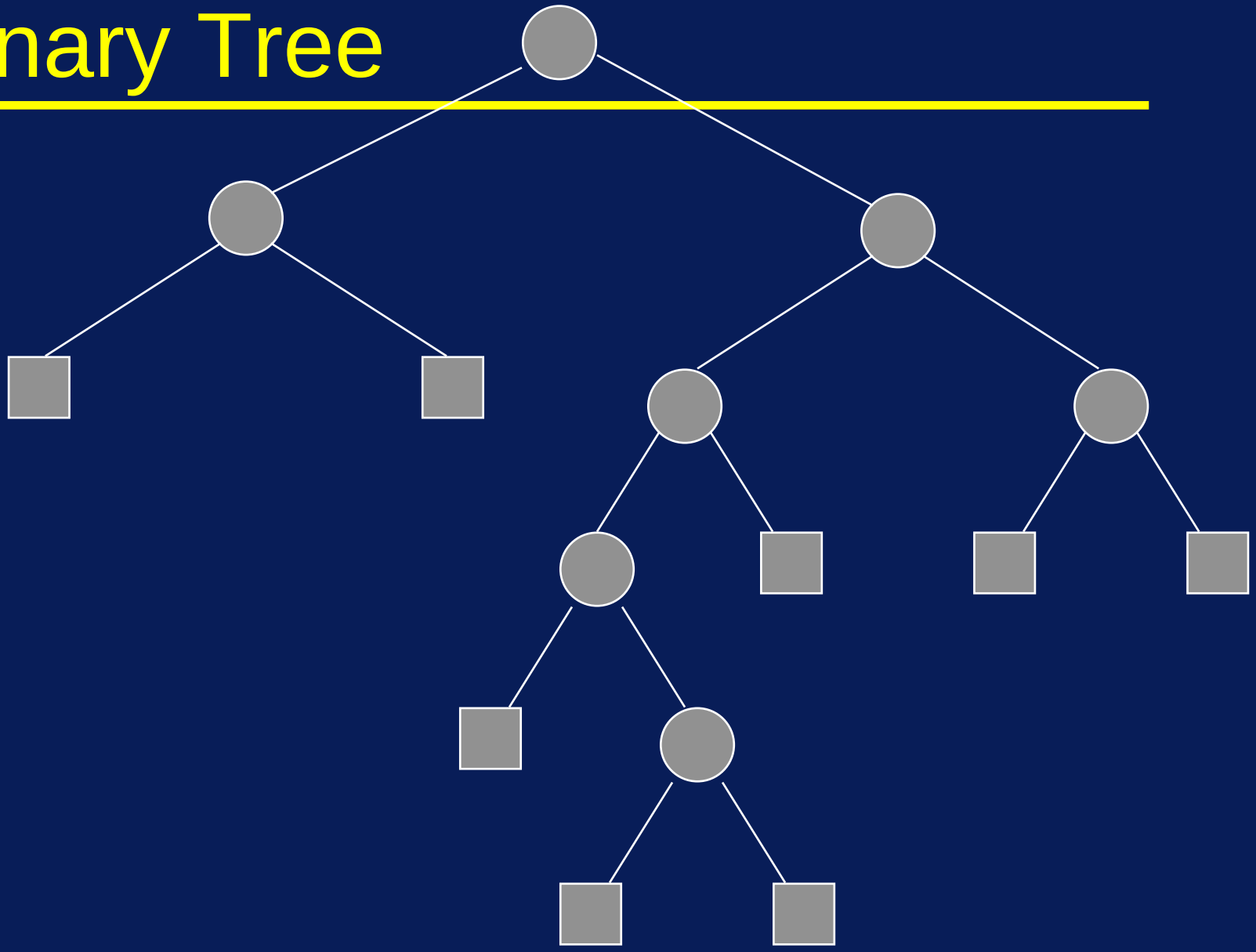
Binary Tree Terminology

- Levels of nodes
  - The level of a node in a binary tree is computed as follows
  - Number the root node 0
  - Number every other node to be 1 more than its parent
  - Then the number of a node *v* is that node's level
  - The level of *v* is the number of branches on the path from to root to *v*

Binary Tree



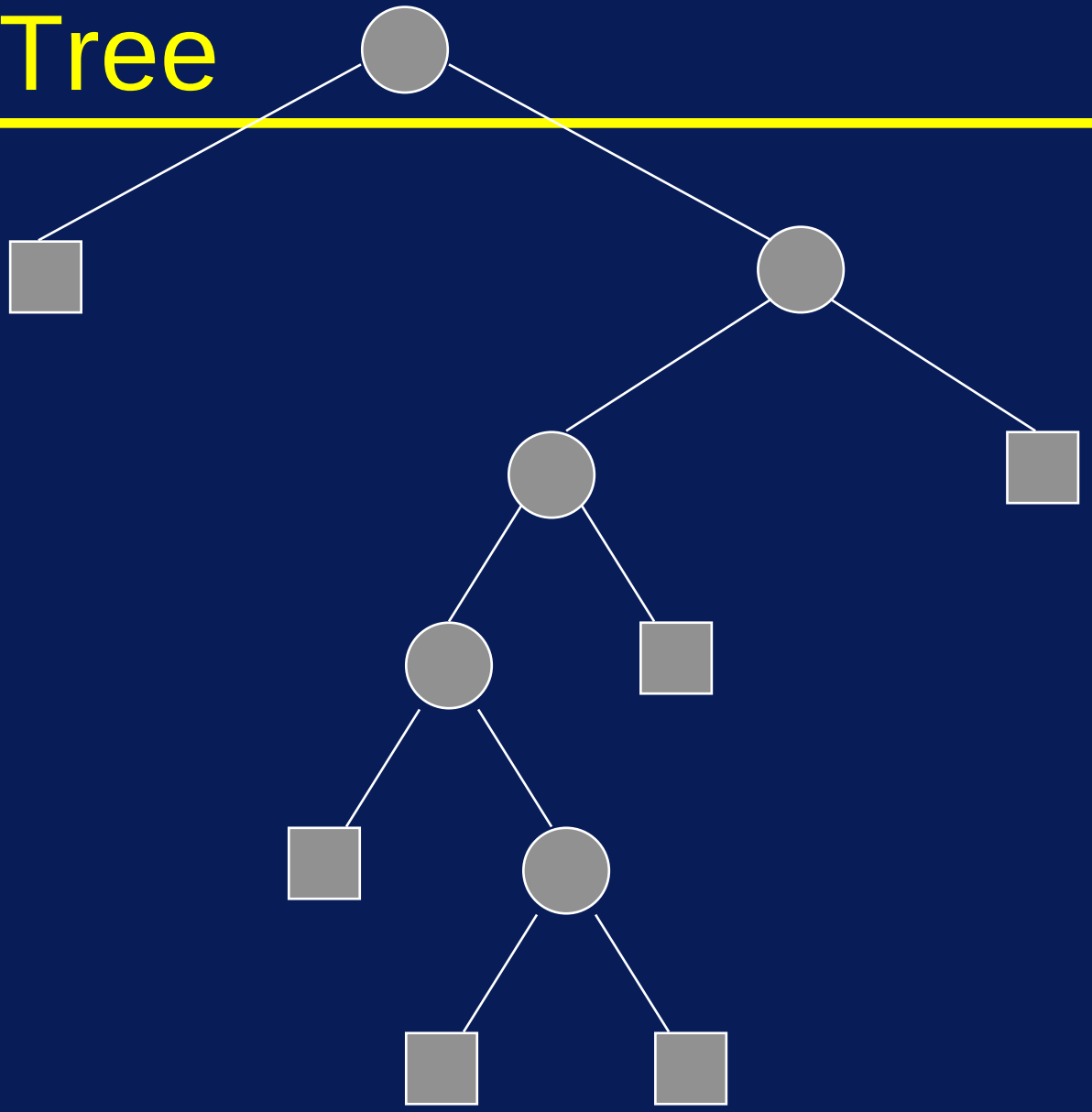
Binary Tree



Binary Tree Terminology

- Skinny Trees
  - every internal node has at most one internal child

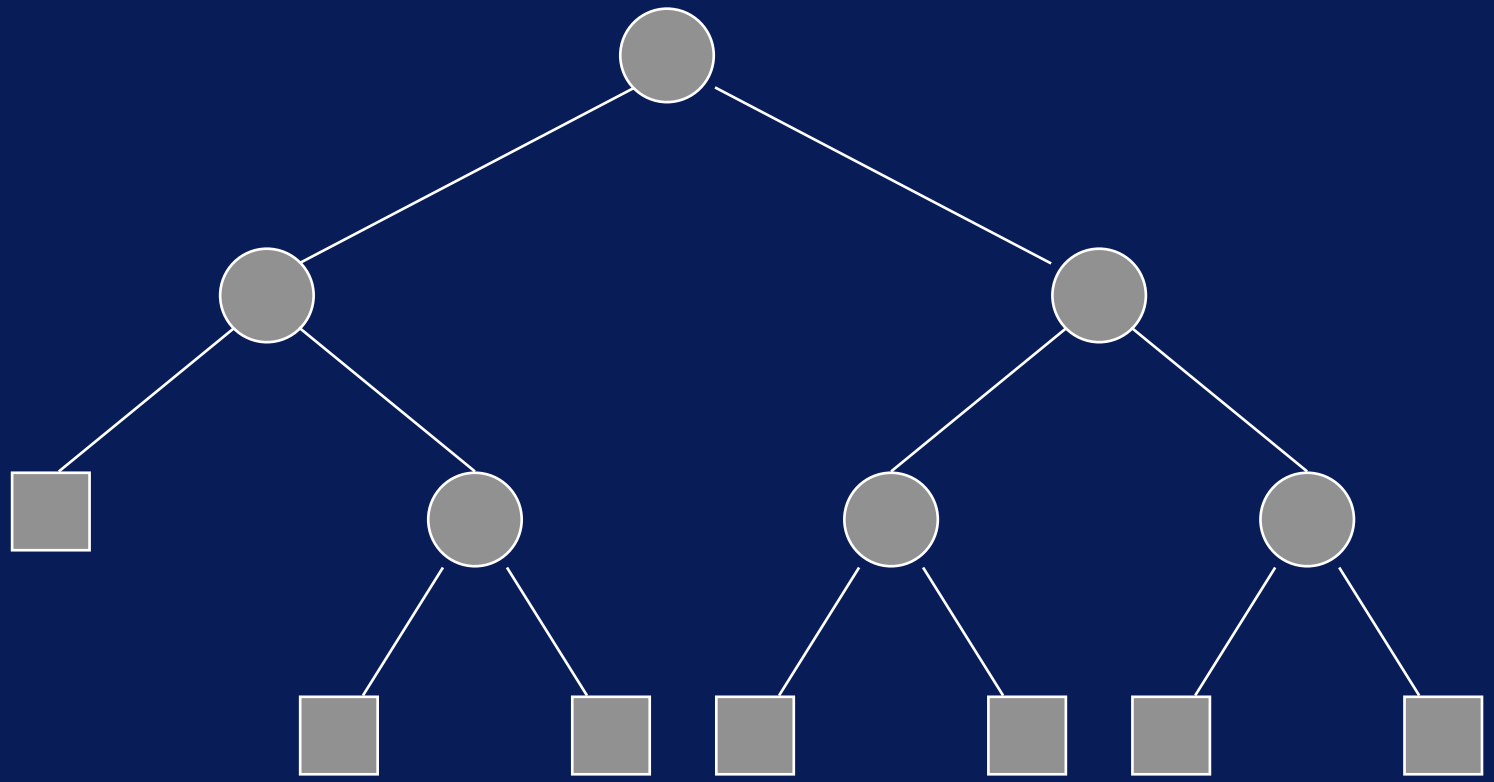
Skinny Tree



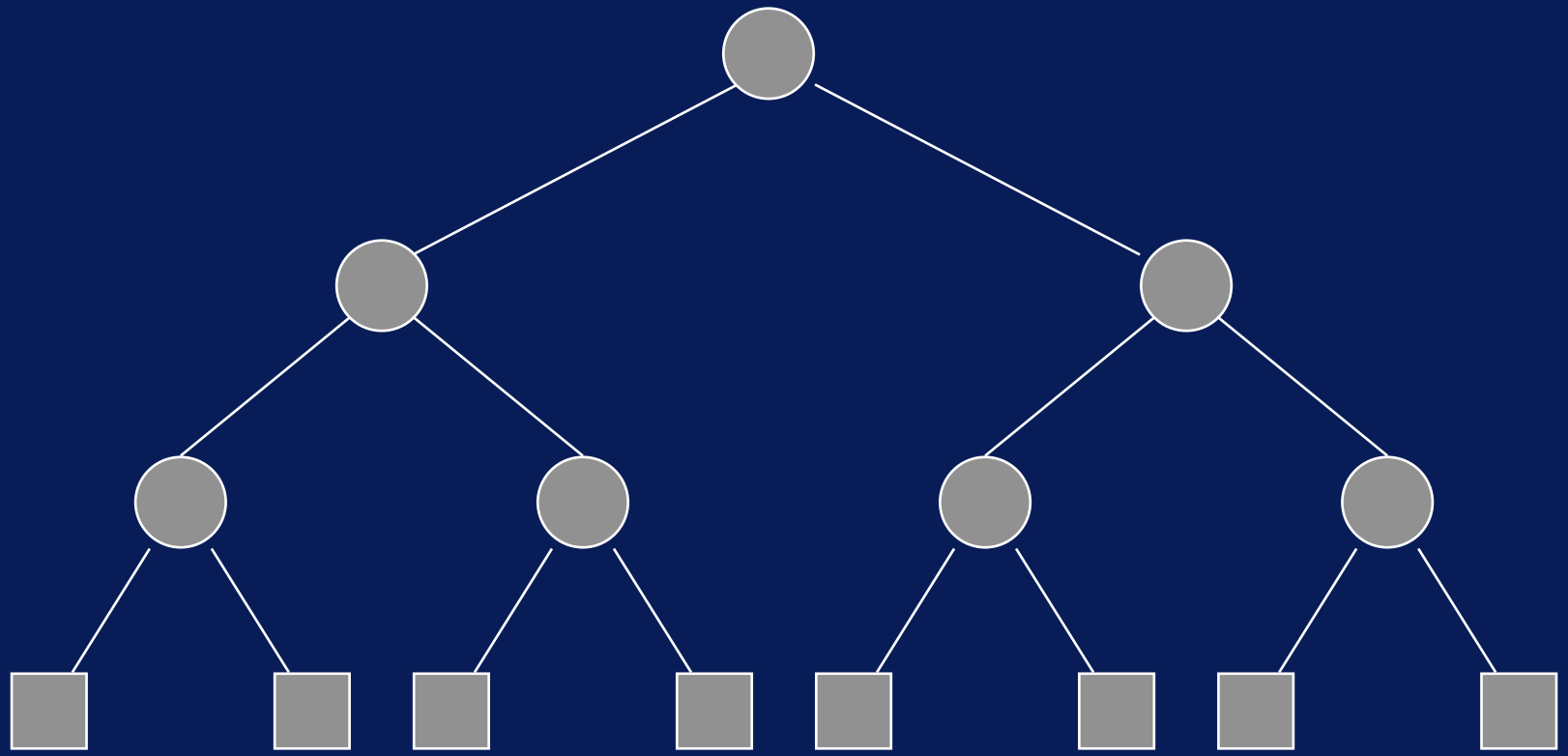
Binary Tree Terminology

- Complete Binary Trees (Fat Trees)
  - the external nodes appear on at most two adjacent levels
  - Perfect Trees: complete trees having all their external nodes on one level
  - Left-complete Trees: the internal nodes on the lowest level is in the leftmost possible position.
  - Skinny trees are the highest possible trees
  - Complete trees are the lowest possible trees

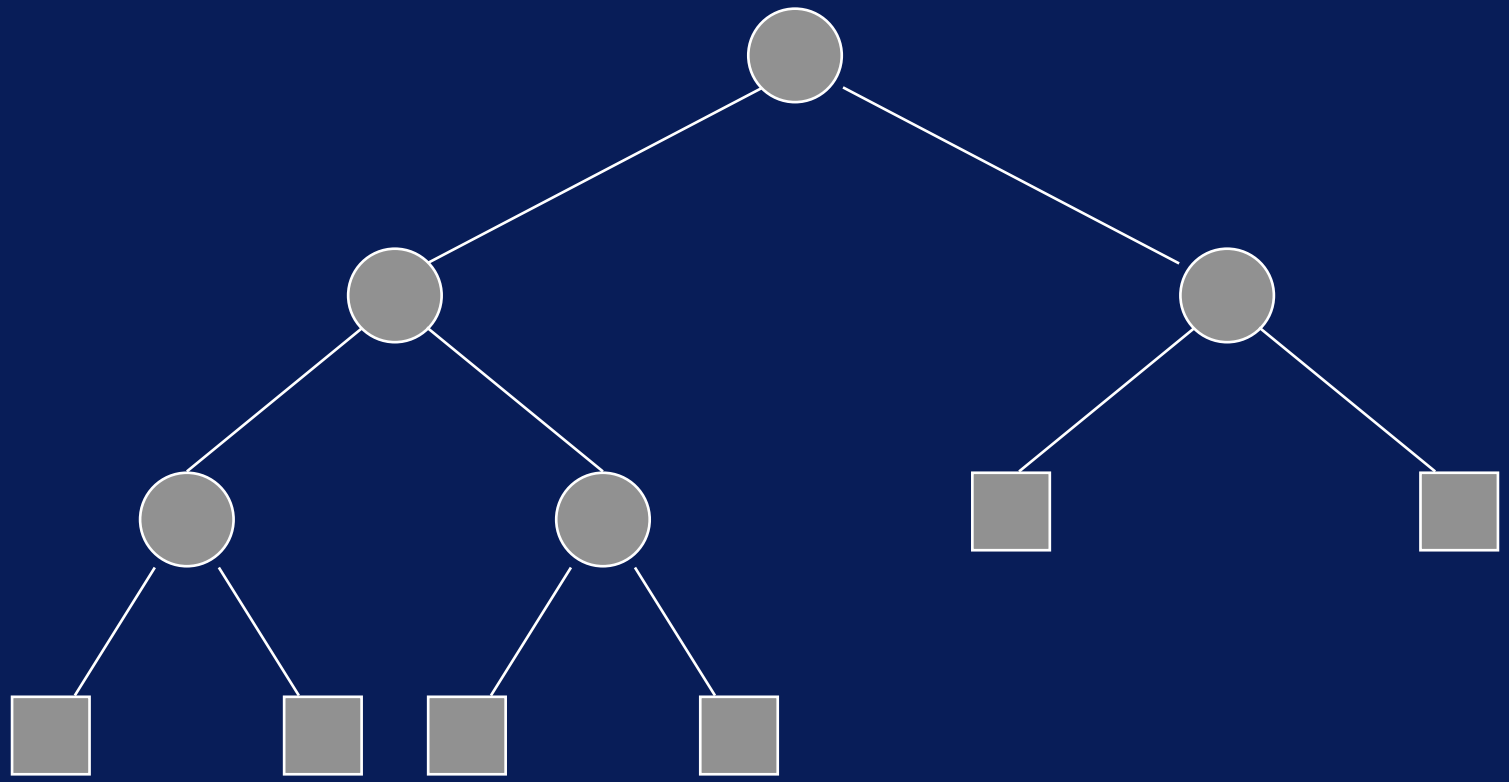
Complete Tree



Perfect Tree



Left-Complete Tree



Binary Tree Terminology

- A binary tree of height $h \geq 0$ has size at least h
- A binary tree of height at most $h \geq 0$ has size at most $2^h - 1$
- A binary tree of size $n \geq 0$ has height at most n
- A binary tree of size $n \geq 0$ has height at least $\lceil \log (n + 1) \rceil$

Multiway Trees

- Multiway trees are defined in a similar way to binary trees, except that the degree (the maximum number of children) is no longer restricted to the value 2

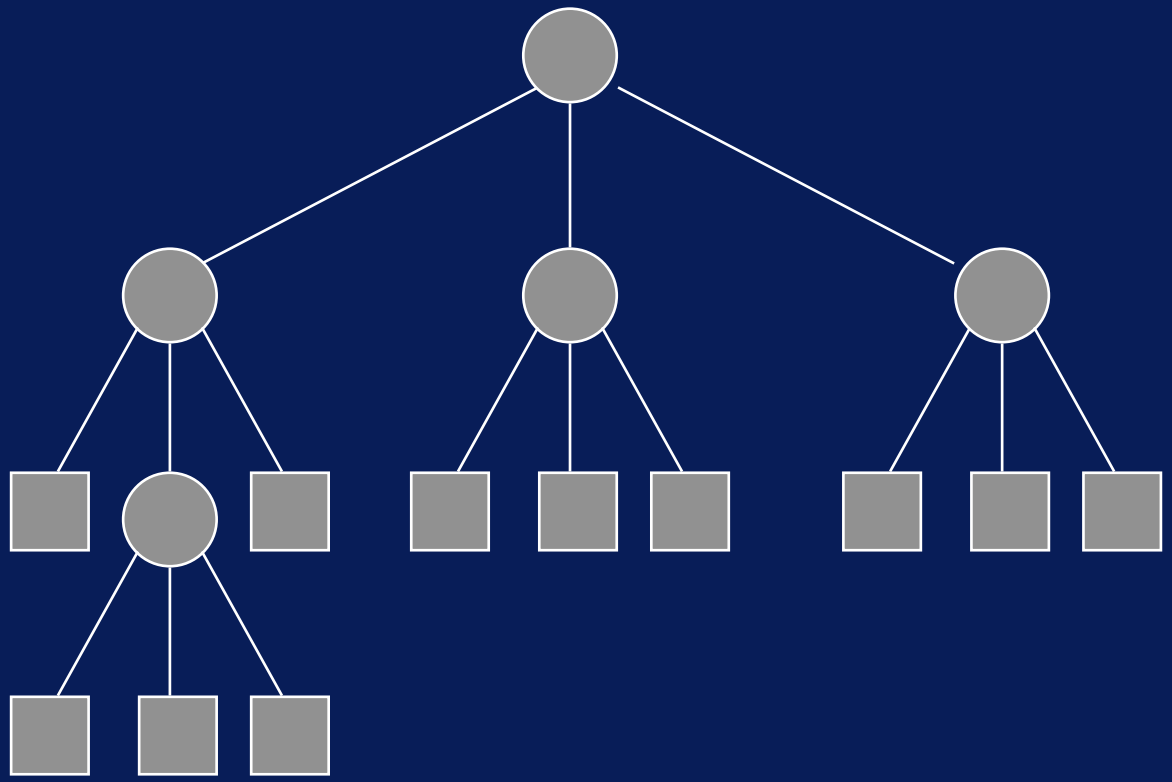
Multiway Trees

- A multiway tree T of n internal nodes, $n \geq 0$,
  - either is empty, if $n = 0$,
  - or consists of
    - » a root node u,
    - » an integer $d_u \geq 1$, the degree of u,
    - » and multiway trees $u(1)$ of n_1 nodes, ..., $u(d_u)$ of n_{d_u} nodes such that $n = 1 + n_1 + ... + n_{d_u}$

Multiway Trees

- A multiway tree T is a d-ary tree, for some $d > 0$, if $d_u = d$, for all internal nodes u in T

d-ary Tree



Multiway Trees

- A multiway tree T is a (a, b)-tree, if $1 \leq a \leq d_u \leq b$, for all u in T
- Every binary tree is a (2, 2)-tree, and vice versa

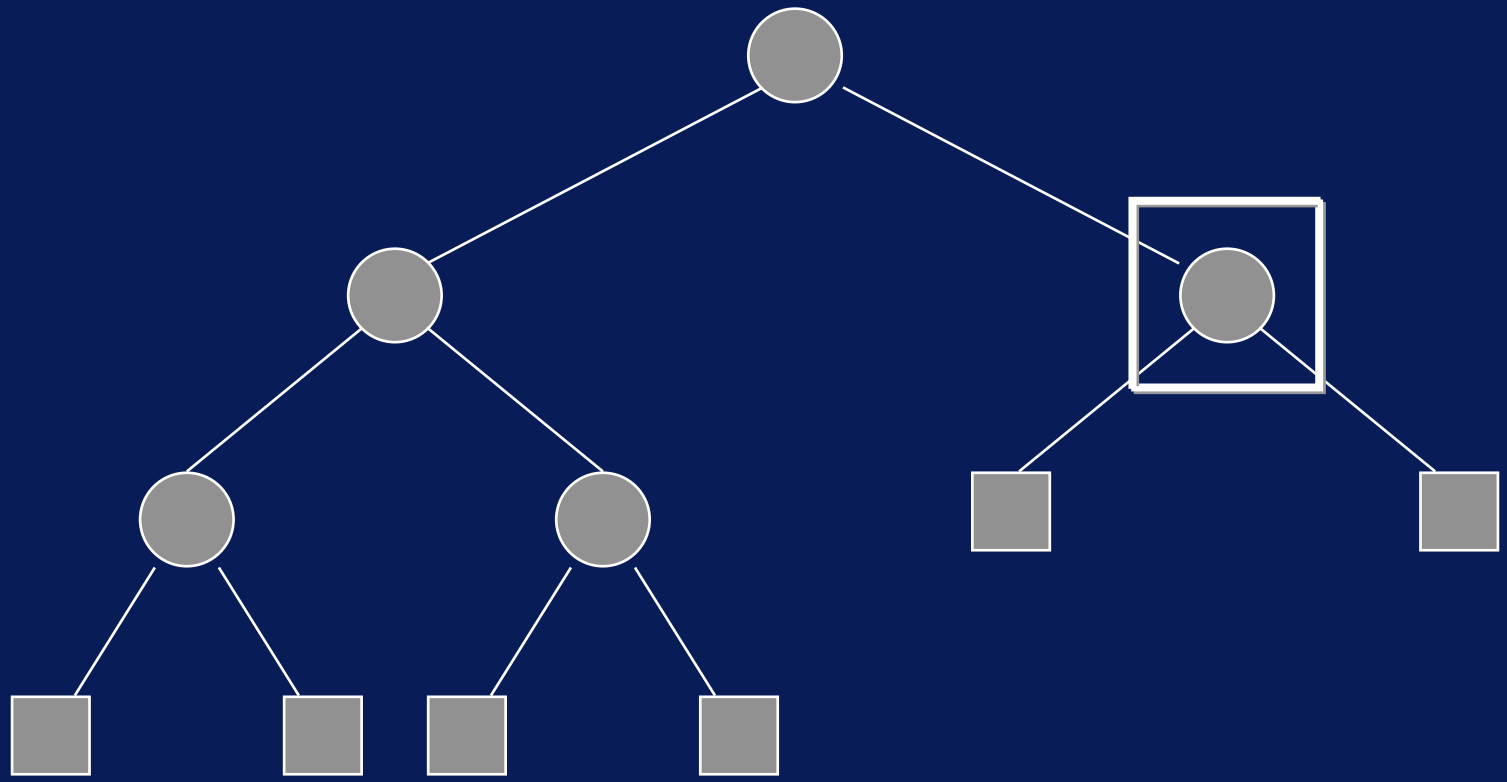
BINARY_TREE & TREE Specification

- So far, no values associated with the nodes of a tree
- Now want to introduce an ADT called BINARY_TREE, which
  - has value of type *intelementtype* associated with the internal nodes
  - has value of type *extelementtype* associated with the external nodes
- These value don't have any effect on BINARY_TREE operations

BINARY_TREE & TREE Specification

- BINARY_TREE has explicit windows and window-manipulation operations
- A window allows us to ‘see’ the value in a node (and to gain access to it)
- Windows can be positioned over any internal or external node
- Windows can be moved from parent to child
- Windows can be moved from child to parent

Window



BINARY_TREE & TREE Specification

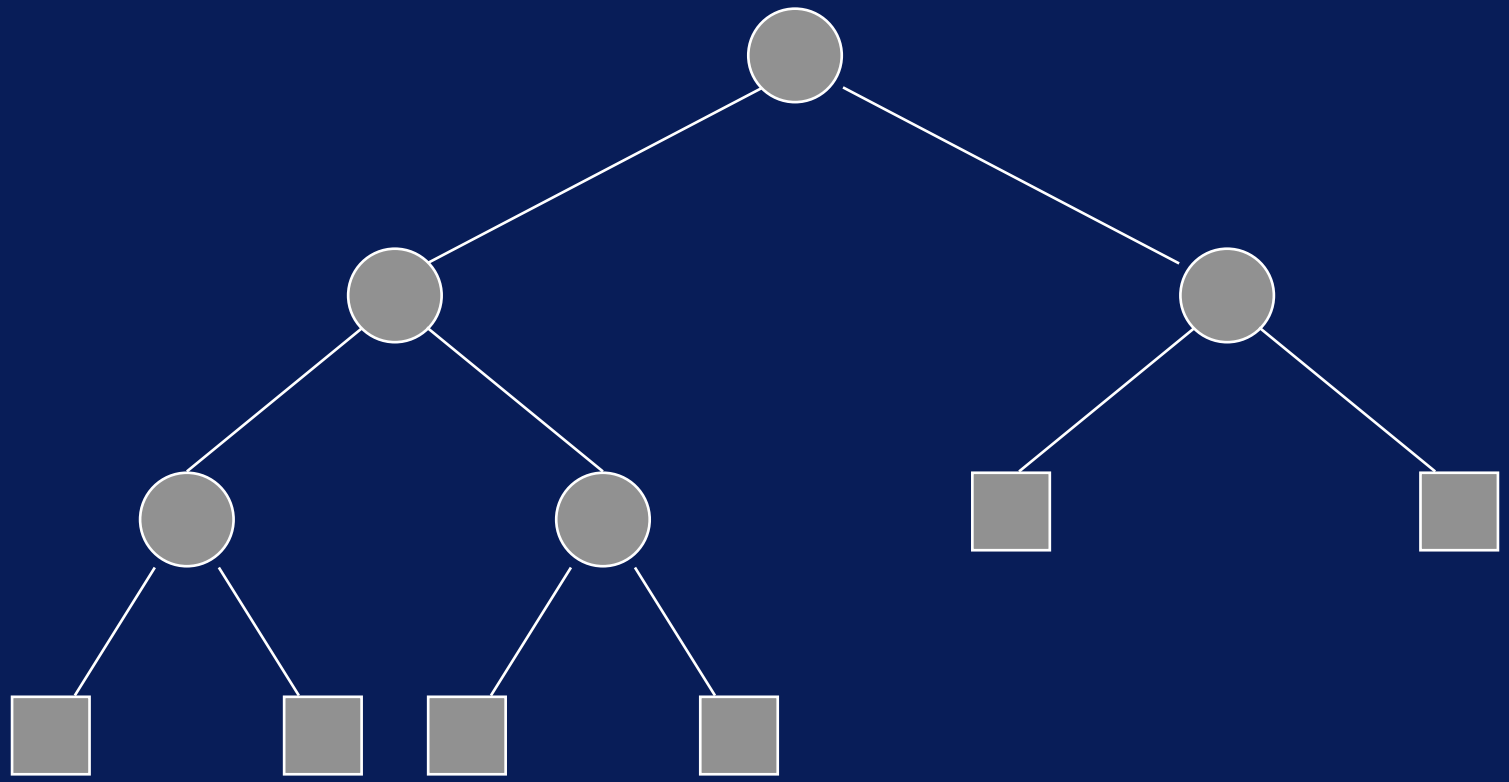
- Let BT denote denote the set of values of BINARY_TREE of *elementtype*
- Let E denote the set of values of type *elementtype*
- Let W denote the set of values of type *windowtype*
- Let B denote the set of Boolean values *true* and *false*

BINARY_TREE Operations

- *Empty*: BT $\rightarrow$ BT :
  The function *Empty(T)* is an empty binary tree; if necessary, the tree is deleted

- *IsEmpty*: BT $\rightarrow$ B :
  The function value *IsEmpty*(*T*) is *true* if *T* is empty; otherwise it is false

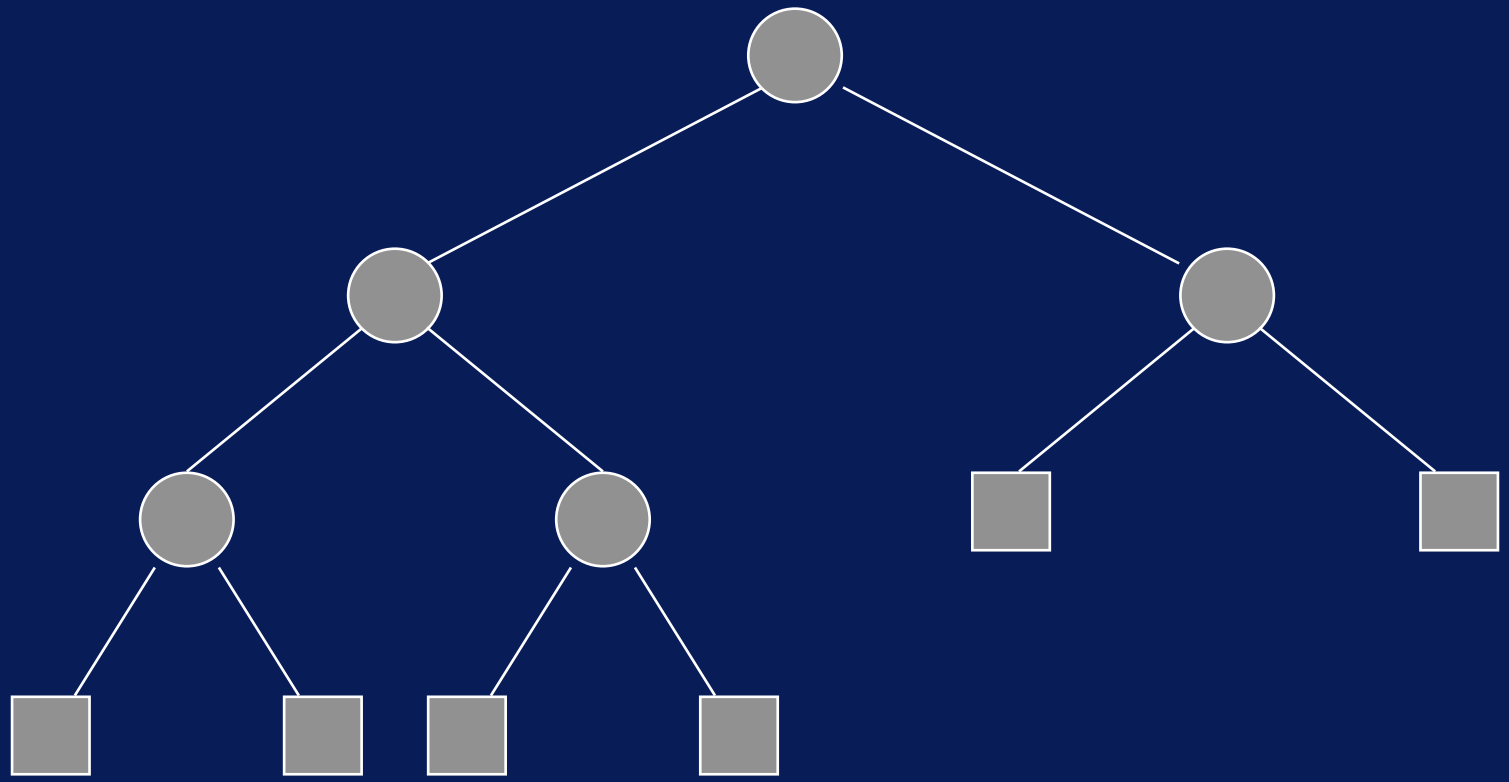
Example



BINARY_TREE Operations

- *Root*: BT $\rightarrow$ W :
  The function value *Root*(*T*) is the window position of the single external node if *T* is empty; otherwise it is the window position of the root of *T*

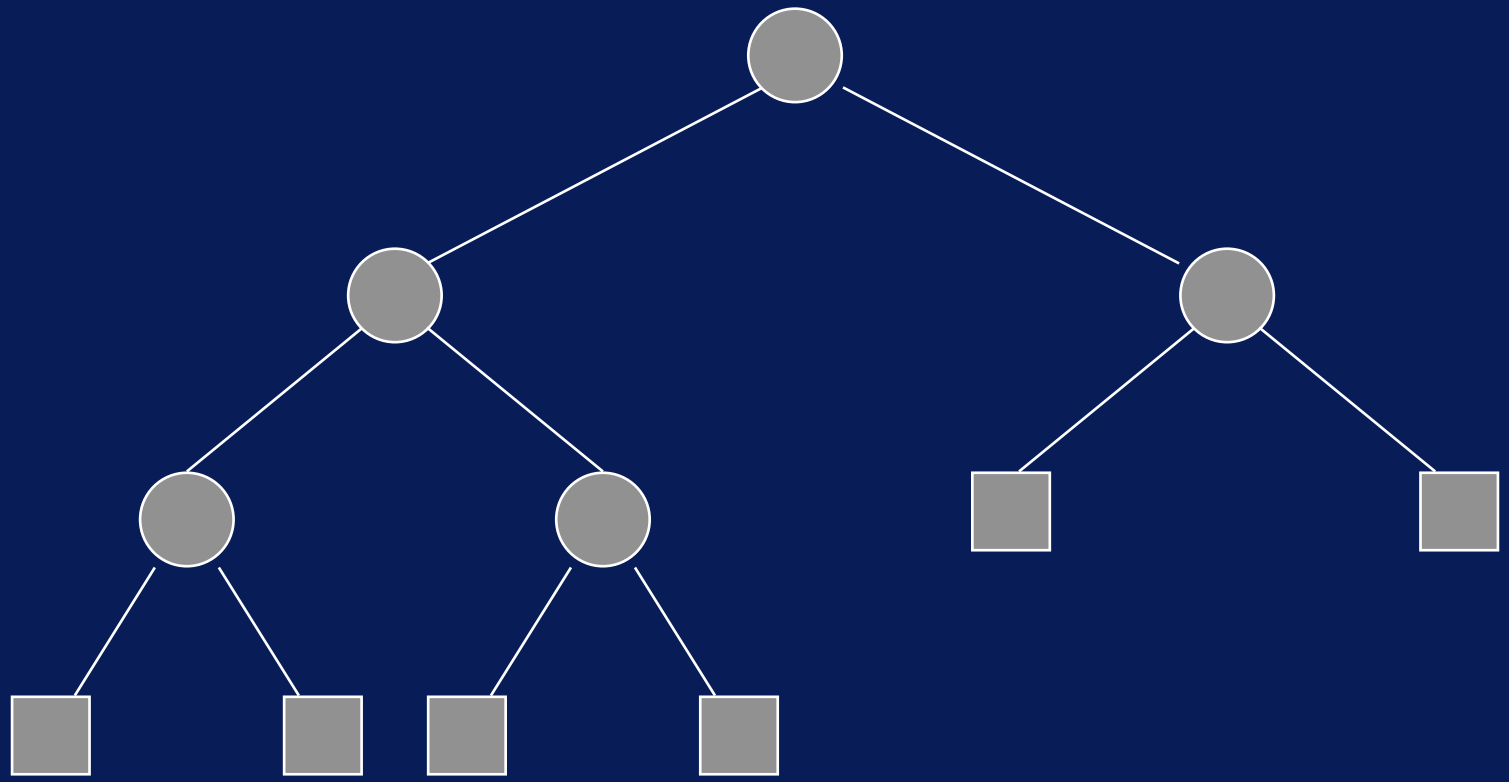
Example



BINARY_TREE Operations

- *IsRoot*: W $\times$ BT $\rightarrow$ B :
  The function value *IsRoot*(*w*, *T*) is *true* if the window *w* is over the root; otherwise it is *false*

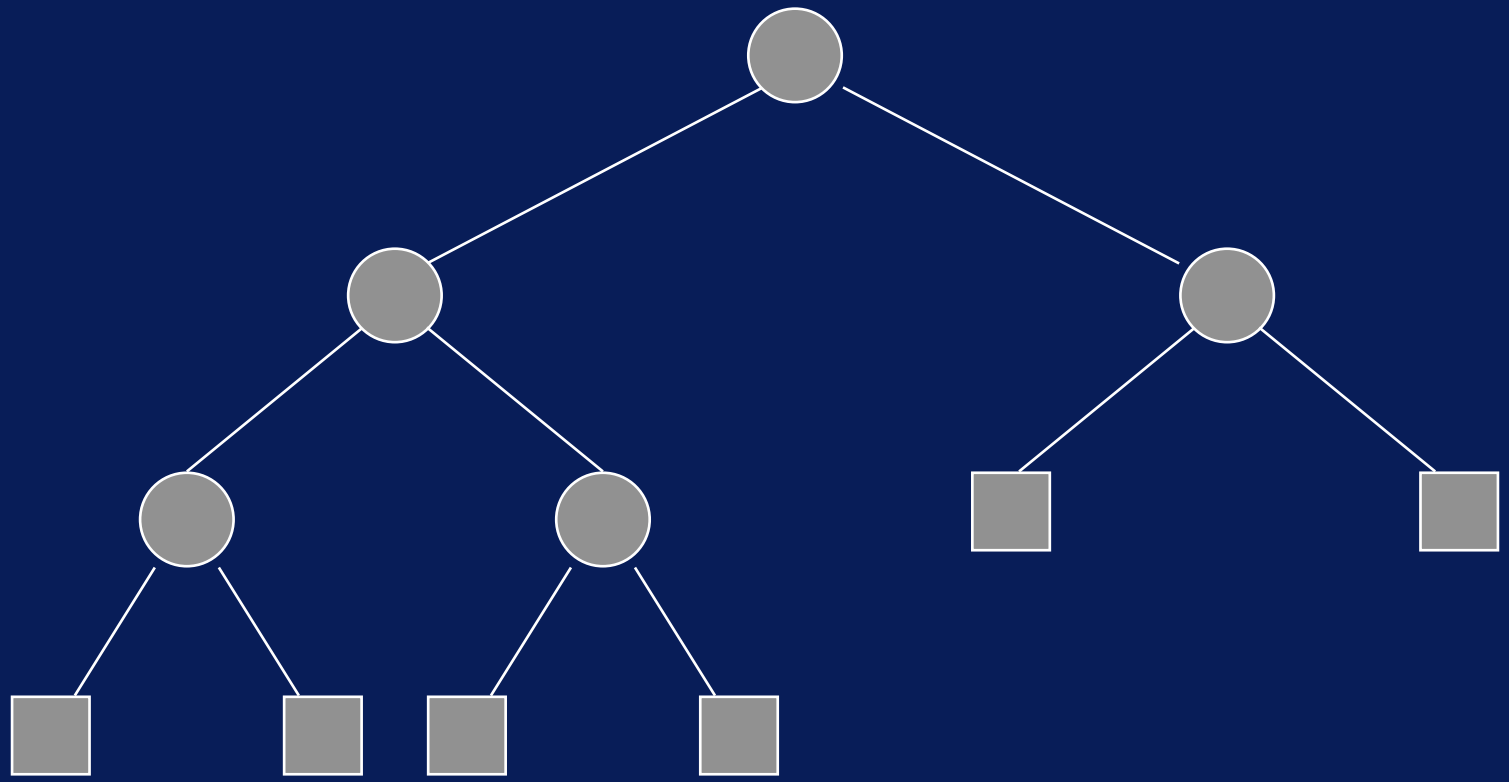
Example



BINARY_TREE Operations

- *IsExternal*: W $\times$ BT $\rightarrow$ B :
  The function value *IsExternal*(*w*, *T*) is *true* if the window *w* is over an external node of *T*; otherwise it is *false*

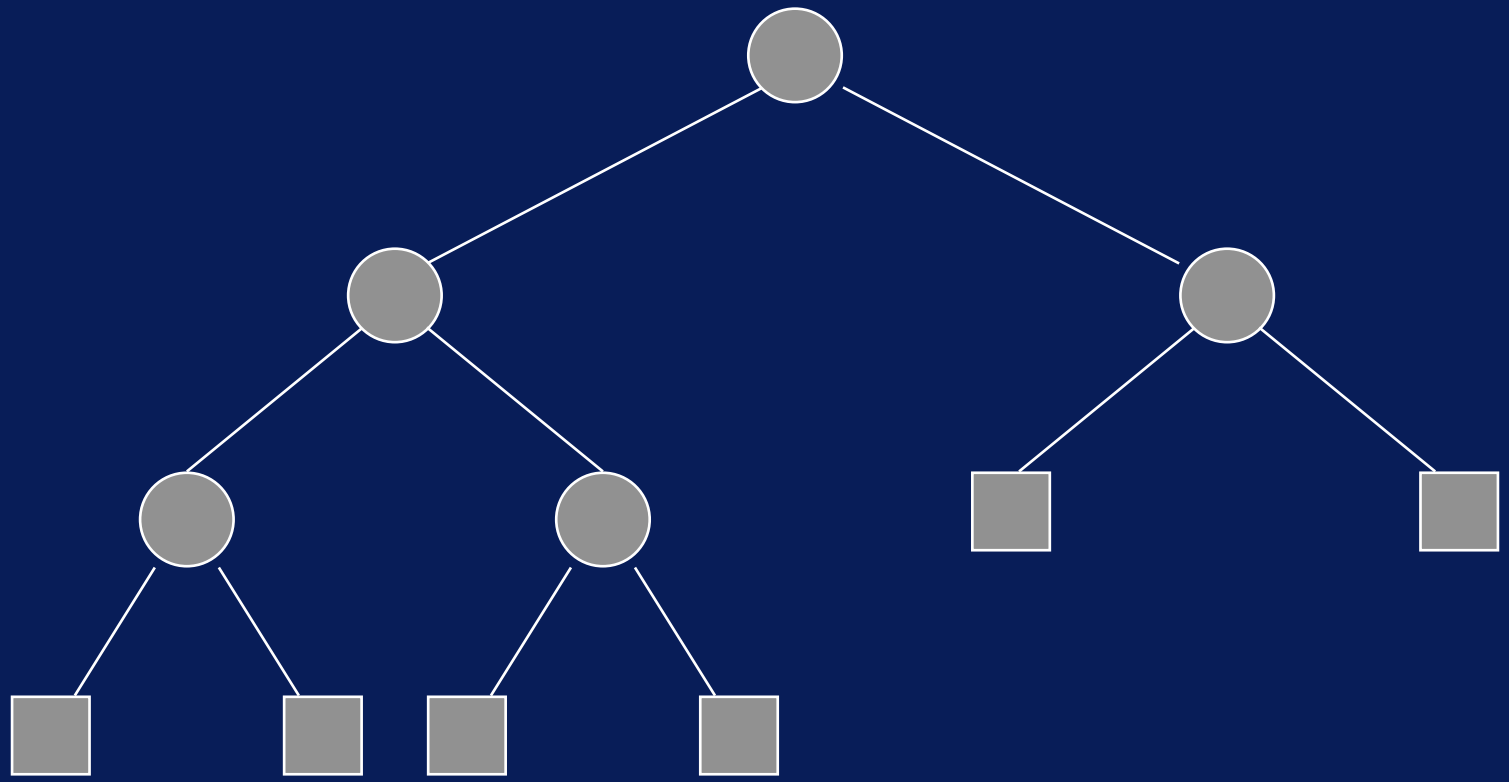
Example



BINARY_TREE Operations

- *Child*: N $\times$ W $\times$ BT $\rightarrow$ W :
  The function value *Child*(*i*, *w*, *T*) is undefined if the node in the window *W* is external or the node in *w* is internal and *i* is neither 1 nor 2; otherwise it is the *i*th child of the node in *w*

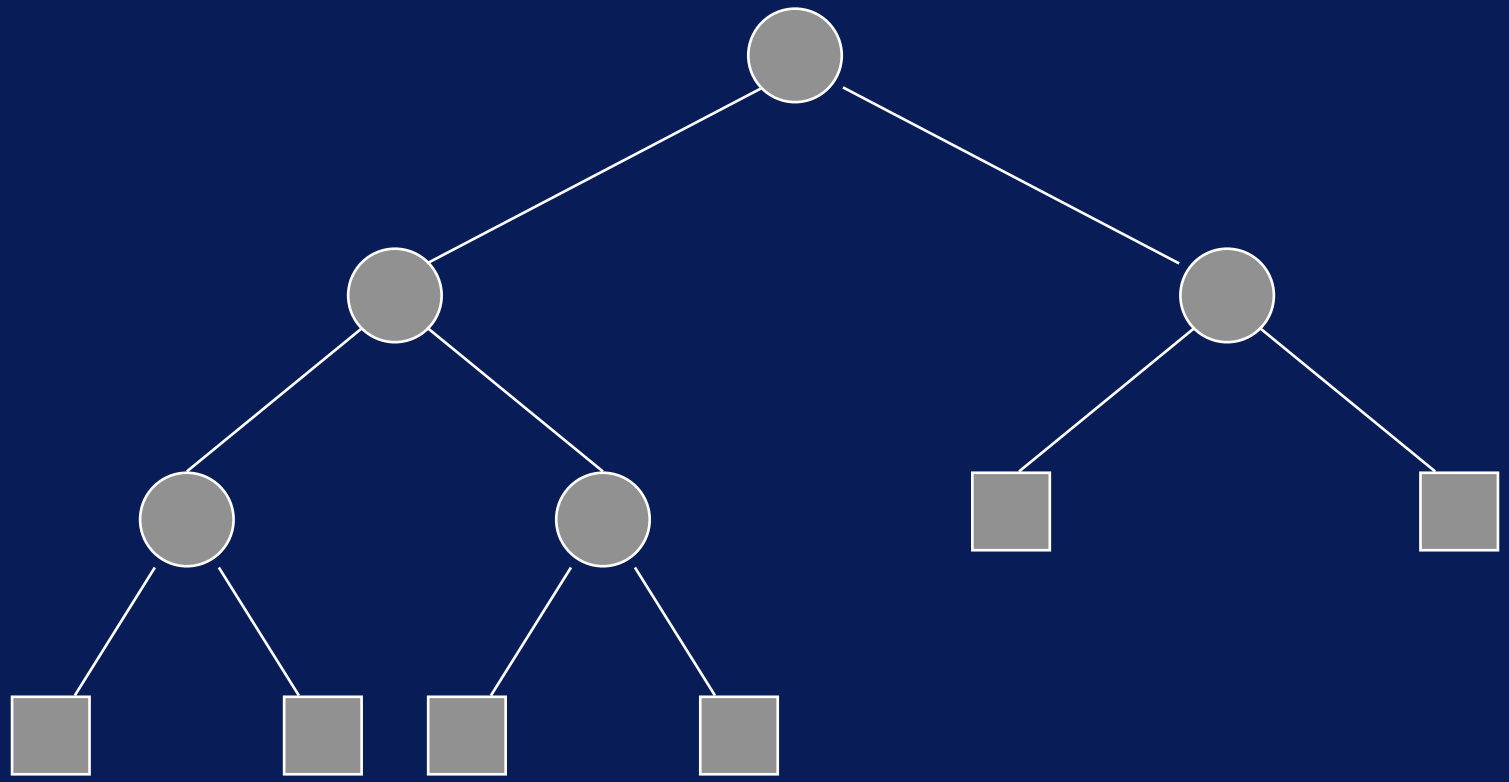
Example



BINARY_TREE Operations

- *Parent*: W $\times$ BT $\rightarrow$ W :
  The function value *Parent*(*w*, *T*) is undefined if *T* is empty or *w* is over the root of *T*; otherwise it is the window position of the parent of the node in the window *w*

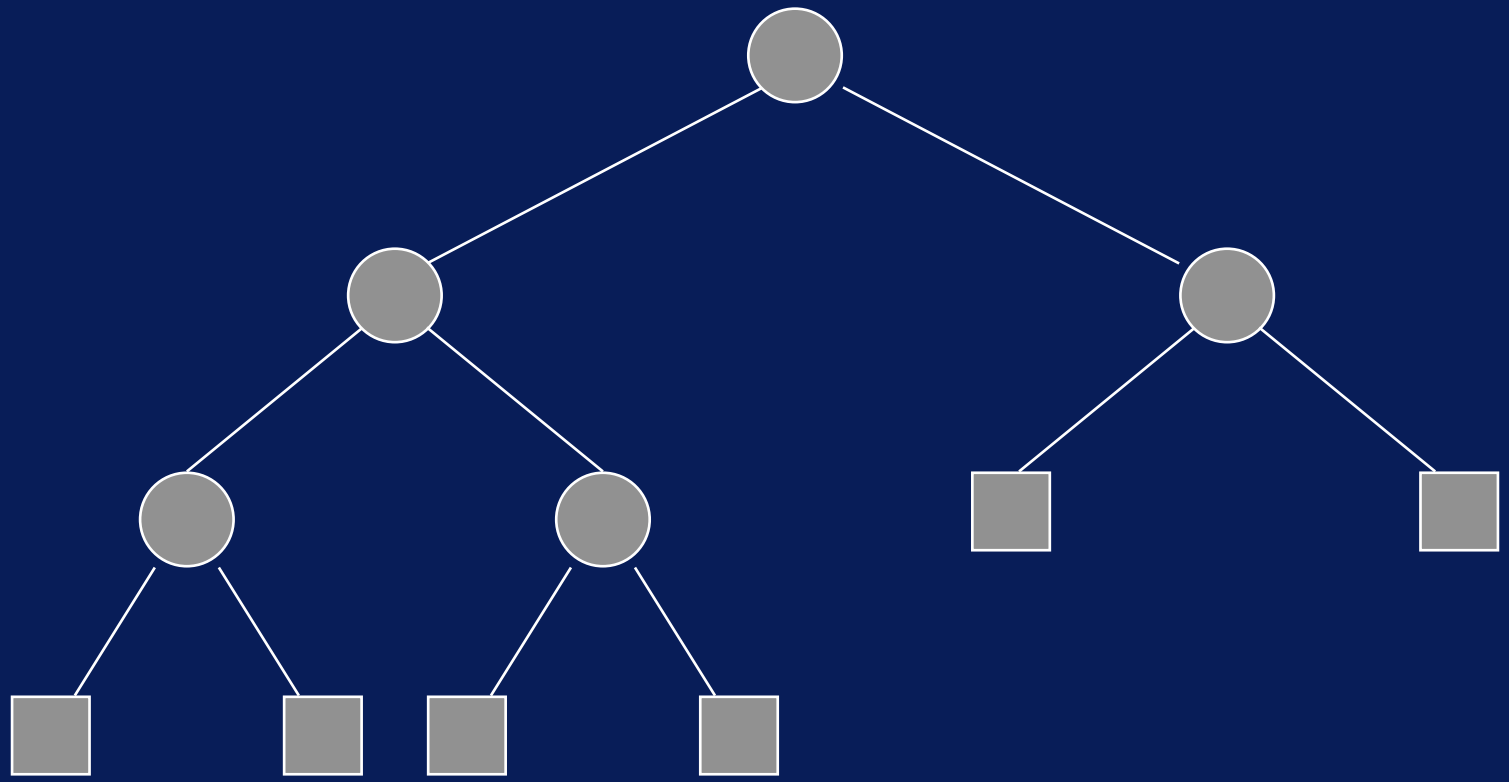
Example



BINARY_TREE Operations

- *Examine*: W $\times$ BT $\rightarrow$ *I* :
  The function value *Examine*(*w*, *T*) is undefined if *w* is over an external node; otherwise it is element at the internal node in the window *w*

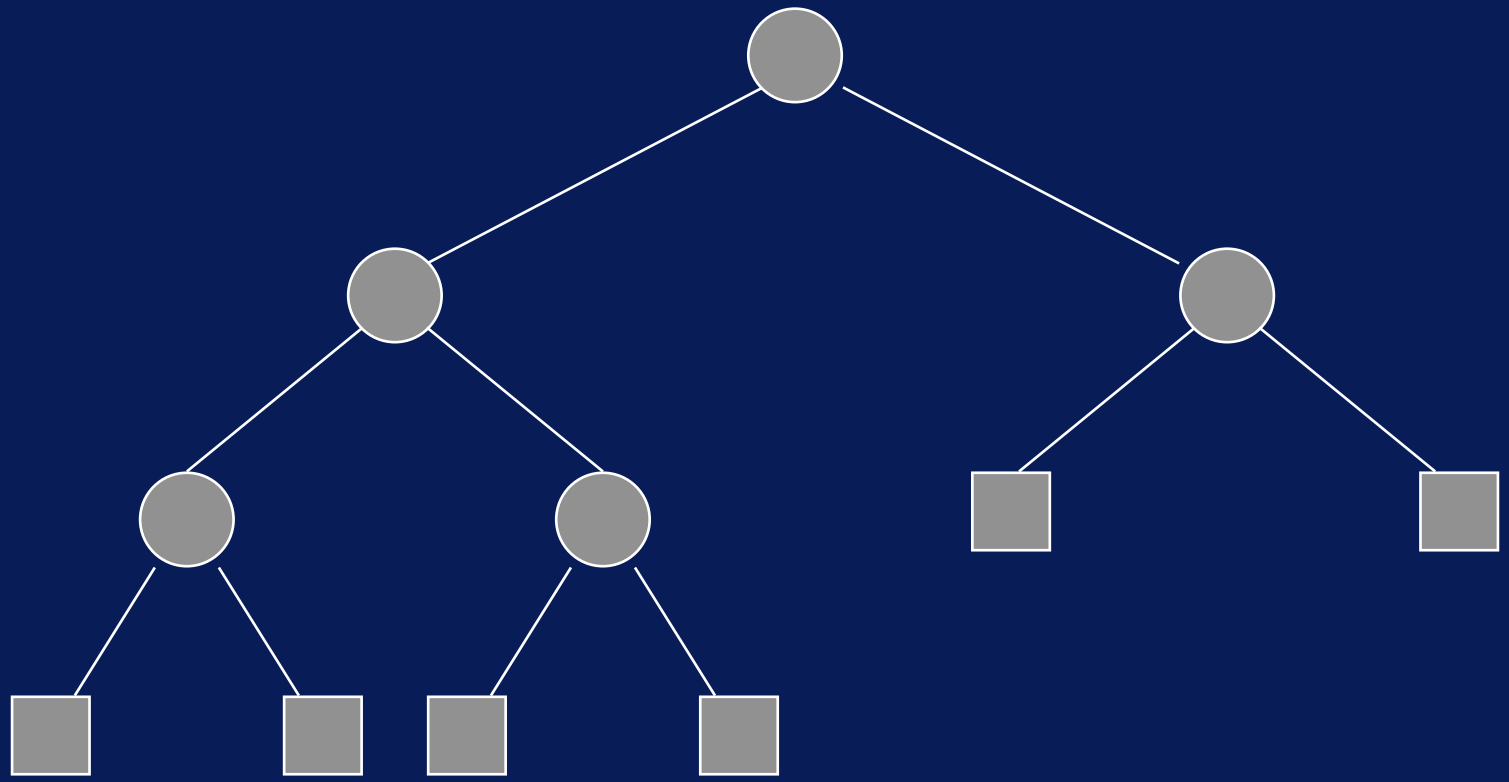
Example



BINARY_TREE Operations

- *Replace*: $E \times \mathrm{W} \times \mathrm{BT} \rightarrow \mathrm{BT}$:
  The function value *Replace*(*e*, *w*, *T*) is undefined if *w* is over an external node; otherwise it is *T*, with the element at the internal node in *w* replaced by *e*

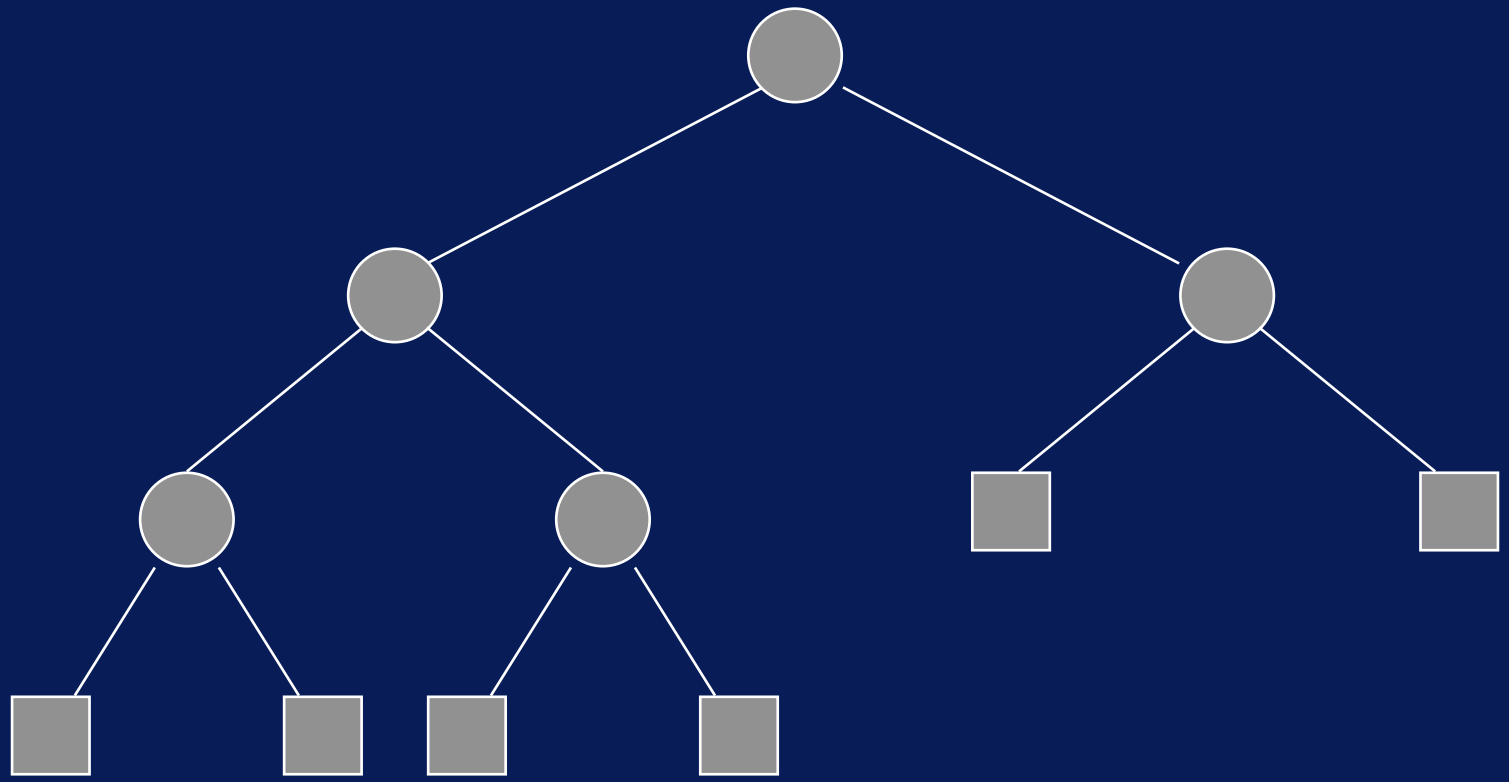
Example



BINARY_TREE Operations

- *Insert*: $E \times \mathrm{W} \times \mathrm{BT} \rightarrow \mathrm{W} \times \mathrm{BT}$:
  The function value *Insert*(*e*, *w*, *T*) is undefined if *w* is over an internal node; otherwise it is *T*, with the external node in *w* replaced by a new internal node with two external children.
  - Furthermore, the new internal node is given the value *e* and the window is moved over the new internal node.

Example



BINARY_TREE Operations

- *Delete*: W $\times$ BT $\rightarrow$ W $\times$ BT :
  - The function value *Delete*(*w*, *T*) is undefined if *w* is over an external node;
  - If w is over a leaf node (both its children are external nodes), then the function value is T with the internal node to be deleted replaced by its left external node

BINARY_TREE Operations

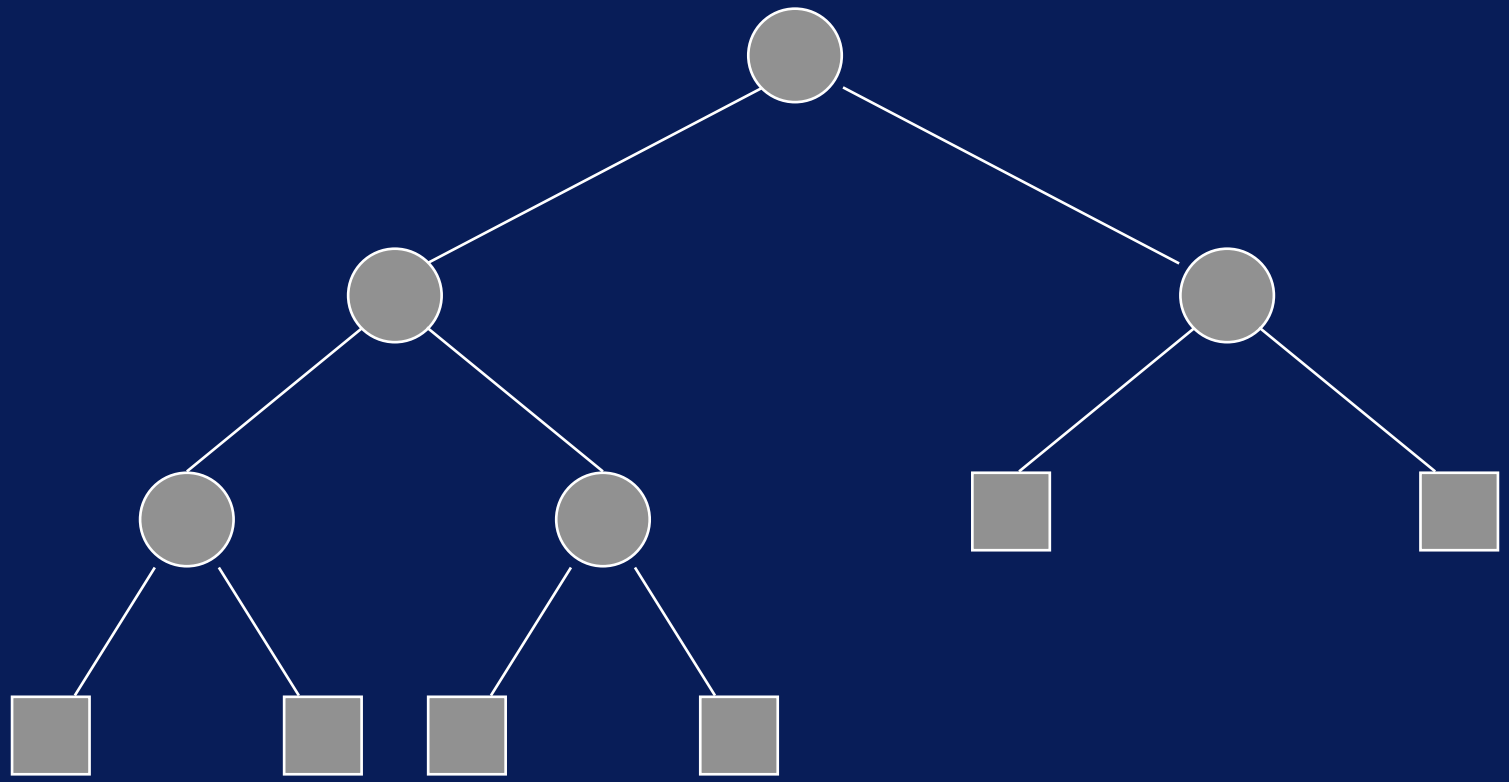
- *Delete*: W $\times$ BT $\rightarrow$ W $\times$ BT :

  If w is over an internal node with just one internal node child, then the function value is T with the internal node to be deleted replaced by its child

BINARY_TREE Operations

- *Delete*: W $\times$ BT $\rightarrow$ W $\times$ BT :
  - if w is over an internal node with two internal node children, then the function value is T with the internal node to be deleted replaced by the leftmost internal node descendent in its right sub-tree
  - In all cases, the window is moved over the replacement node.

Example

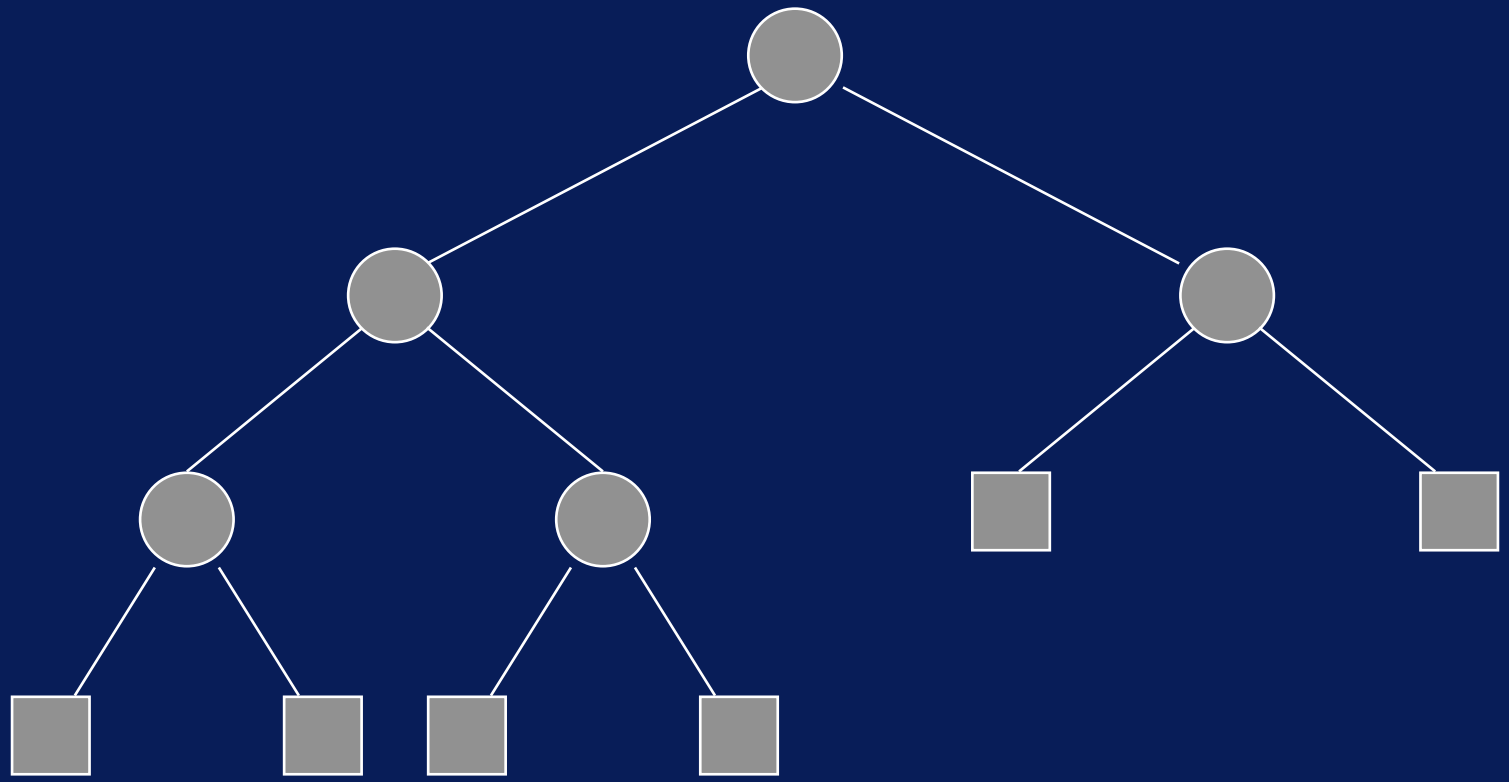


BINARY_TREE Operations

- *Left*: W $\times$ BT $\rightarrow$ W :

  The function value *Left*(*w*, *T*) is undefined if *w* is over an external node; otherwise it is the window position of the left (or first) child of the node *w*

Example

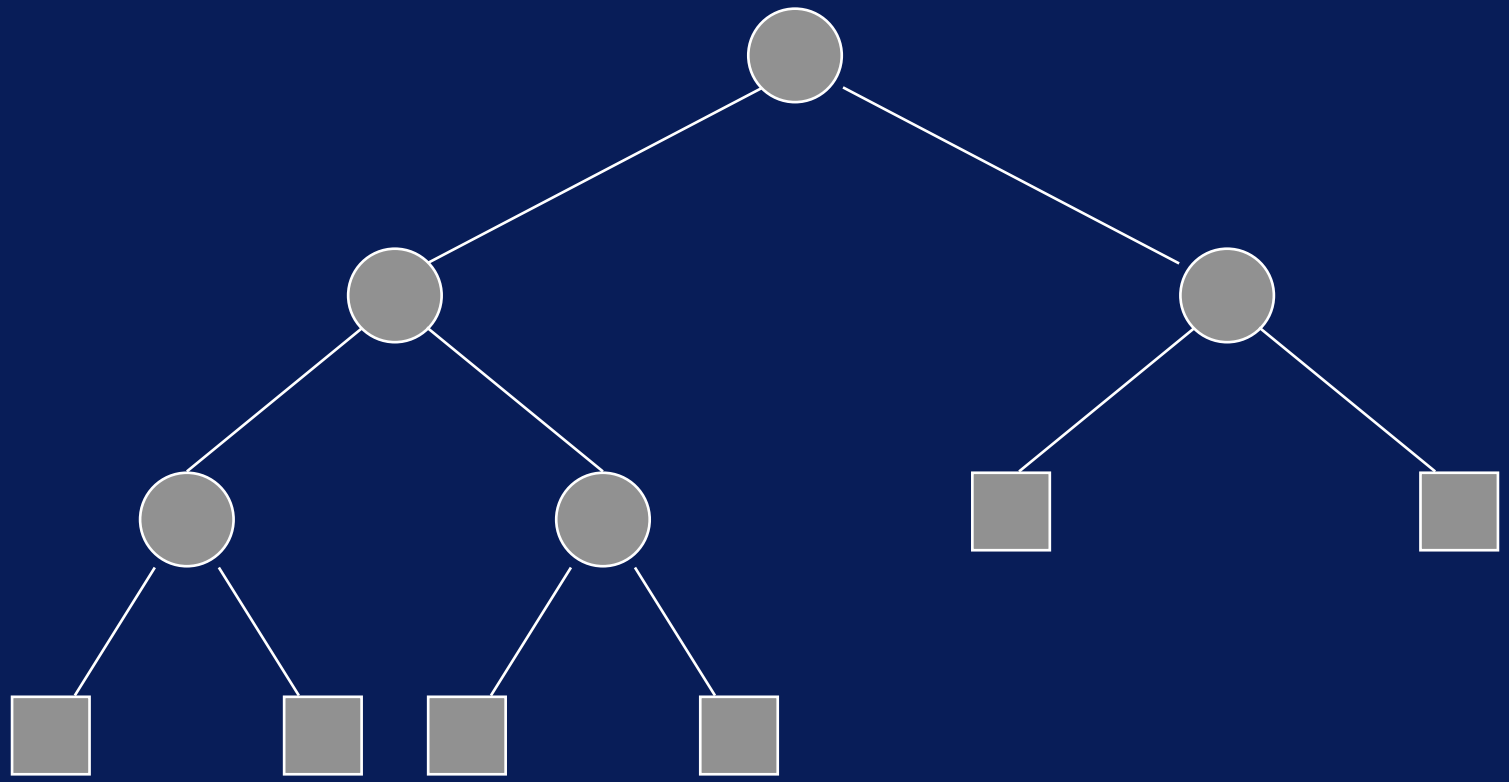


BINARY_TREE Operations

- *Right*: W $\times$ BT $\rightarrow$ W :

  The function value *Right*(*w*, *T*) is undefined if *w* is over an external node; otherwise it is the window position of the right (or second) child of the node *w*

Example

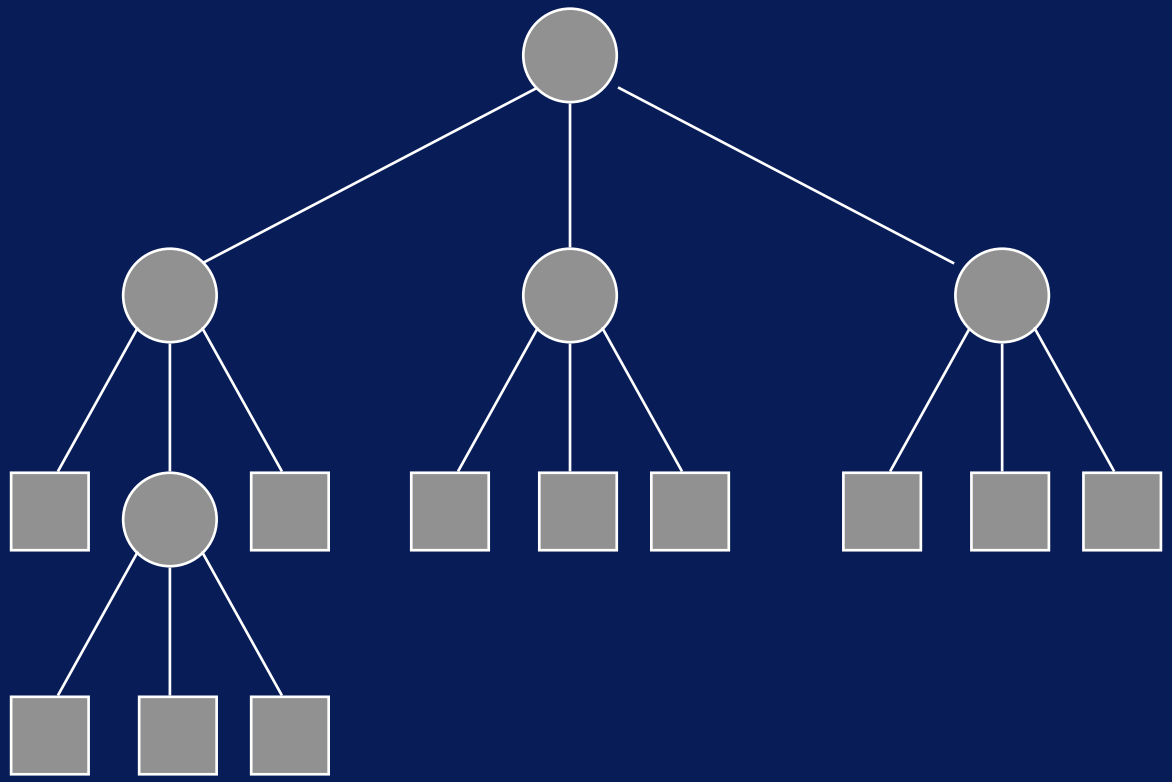


TREE Operations

- *Degree*: W $\times$ T $\rightarrow$ *I* :

  The function value *Degree*(*w*, *T*) is the degree of the node in the window *w*

d-ary Tree

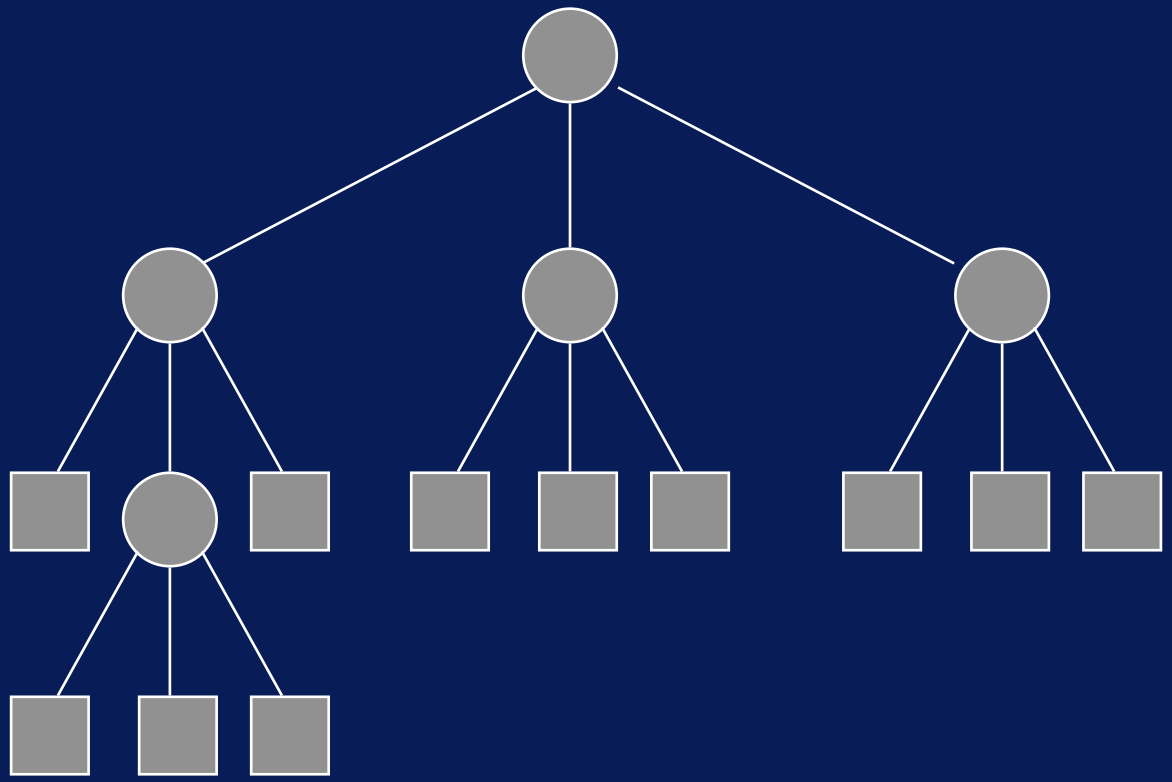


TREE Operations

- *Child*: $\mathbb{N} \times \mathbb{W} \times \mathbb{T} \rightarrow \mathbb{W}$:

  The function value *Child*(*i*, *w*, *T*) is undefined if the node in the window *w* is external, or if the node in *w* is internal and *i* is outside the range 1..*d*, where *d* is the degree of the node; otherwise it is the *i*th child of the node in *w*

d-ary Tree



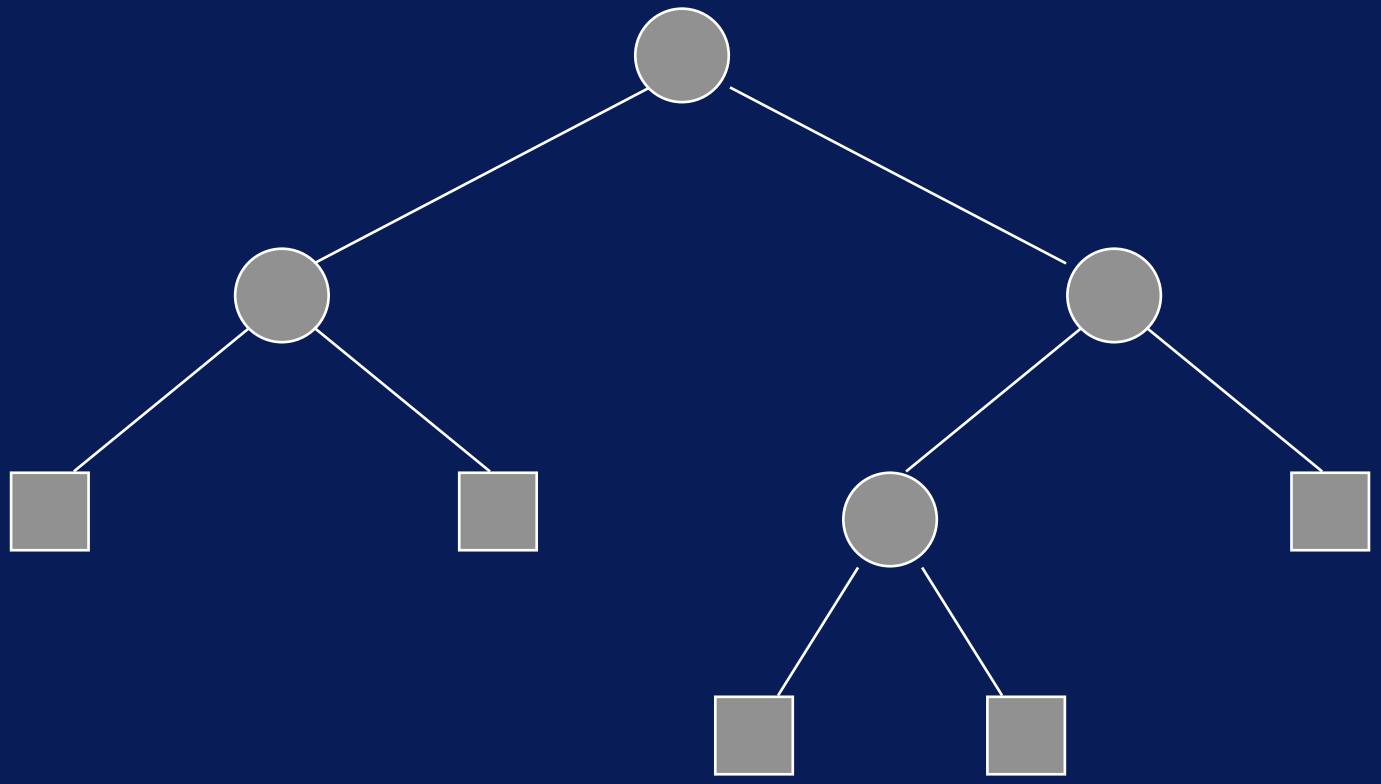
BINARY_TREE Representation

```
/* pointer implementation of BINART_TREE ADT */

#include <stdio.h>
#include <math.h>
#include <string.h>

#define FALSE 0
#define TRUE 1

typedef struct {
           int number;
           char *string;
         } ELEMENT_TYPE;
```

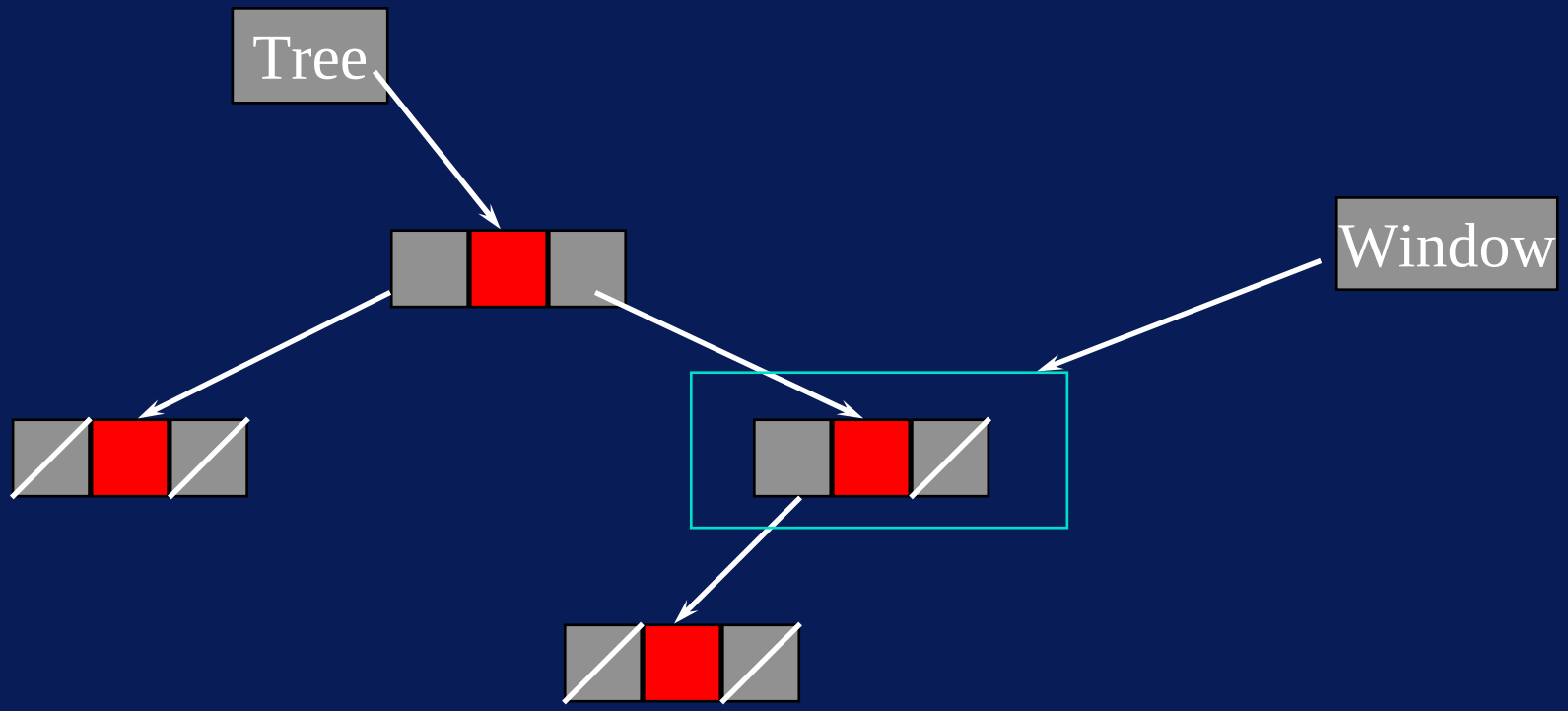
BINARY_TREE Representation

```
typedef struct node *NODE_TYPE;

typedef struct node{
          ELEMENT_TYPE element;
          NODE_TYPE left, right;
        } NODE;

typedef NODE_TYPE BINARY_TREE_TYPE;
typedef NODE_TYPE WINDOW_TYPE;
```

BINARY_TREE Representation



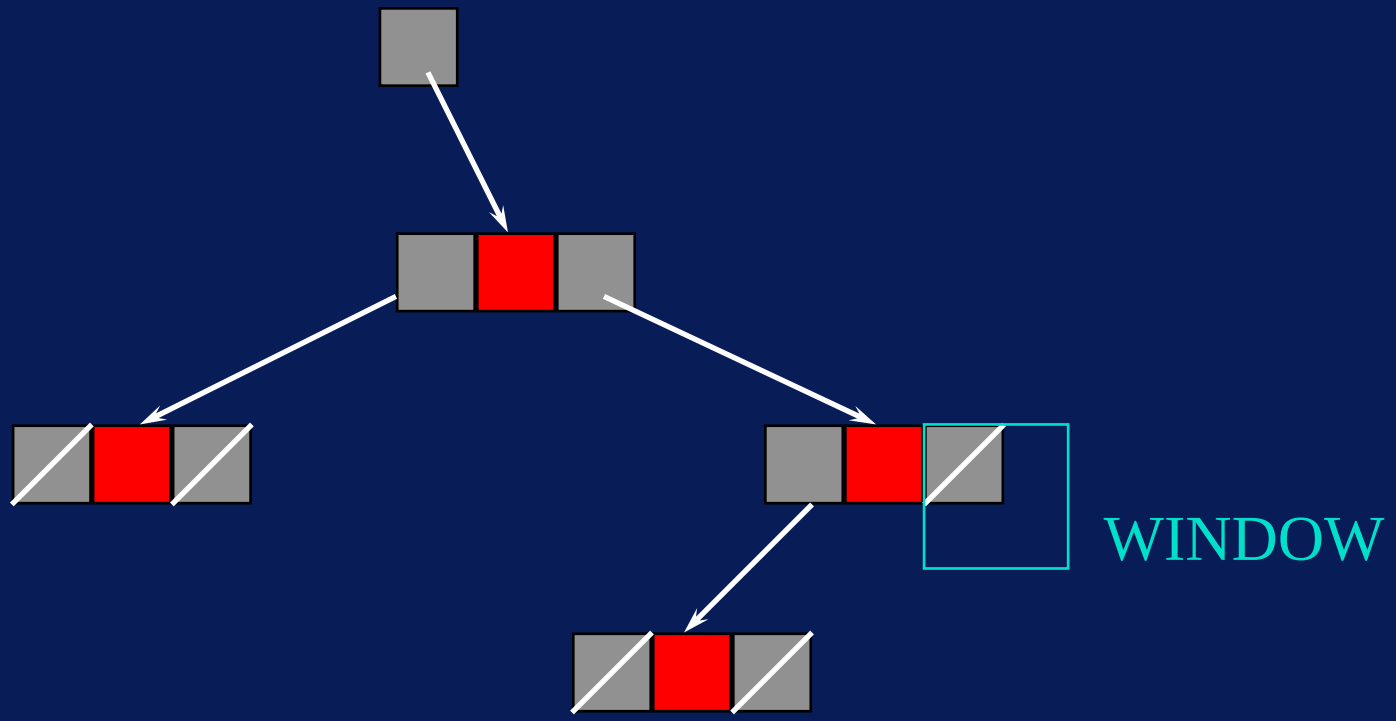
BINARY_TREE Representation



BINARY_TREE Representations

- This implementation assumes that we are going to represent external nodes as NULL links
- For many ADT operations, we need to know if the window is over an internal or an external node
  - we are over an external node if the window is NULL

BINARY_TREE Representation



BINARY_TREE Representations

- Whenever we insert an internal node (remember we can only do this if the window is over an external node) we simply make its two children NULL

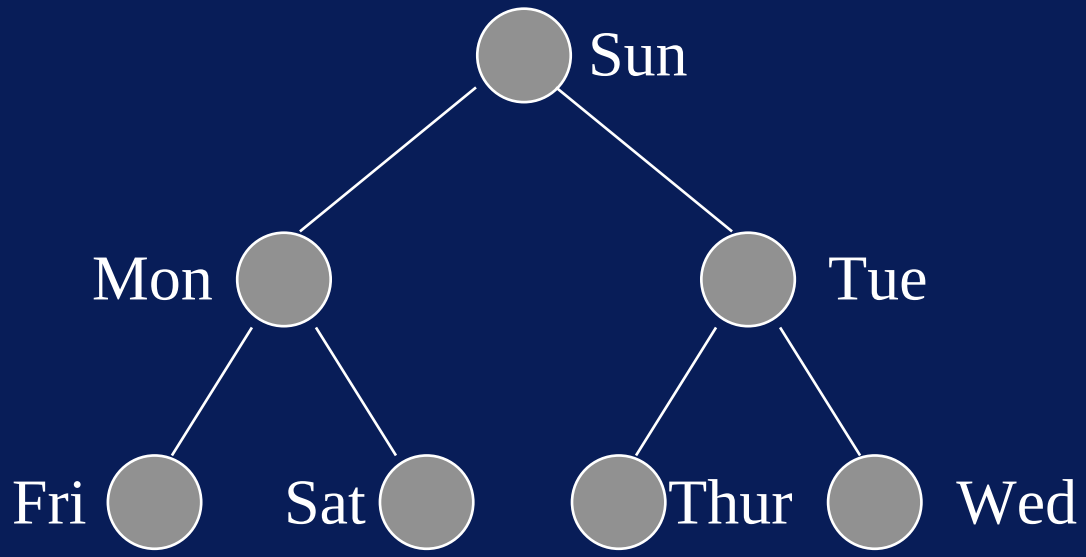
Binary Search Trees

- A Binary Search Tree (BST) is a special type of binary tree
  - it represents information is an ordered format
  - A binary tree is binary search tree if for every node *w*, all keys in the left subtree of i have values less than the key of *w* and all keys in the right subtree have values greater than key of *w*.

Binary Search Trees

- Definition: A binary search tree *T* is a binary tree; either it is empty or each node in the tree contains an identifier and:
  - all keys in the left subtree of *T* are less (numerically or alphabetically) than the identifier in the root node *T*;
  - all identifers in the right subtree of *T* are greater than the identifier in the root node *T*;
  - The left and right subtrees of *T* are also binary search trees.

Binary Search Trees



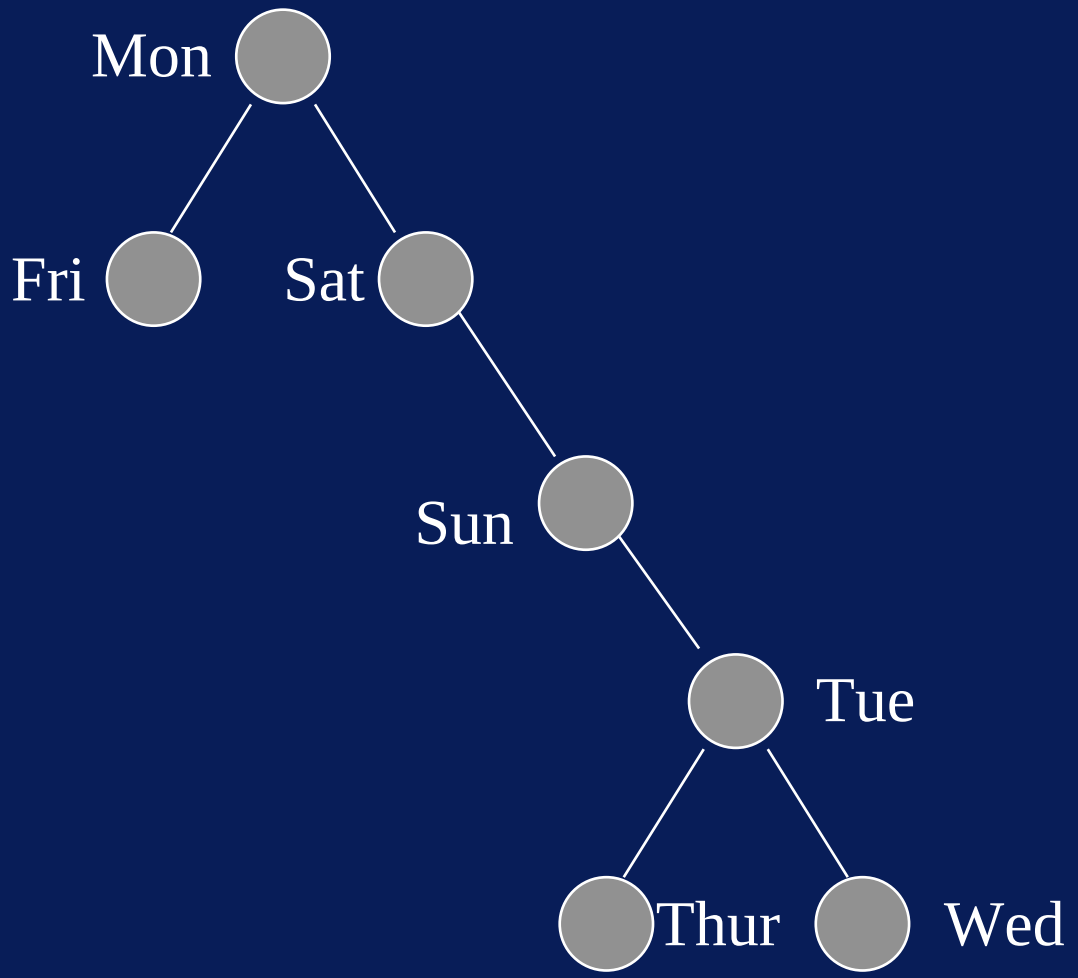
Binary Search Trees

- The main point to notice about such a tree is that, if traversed inorder, the keys of the tree (*i.e.* its data elements) will be encountered in a sorted fashion.
- Furthermore, efficient searching is possible using the binary search technique
  - search time is $O(log_2 n)$.

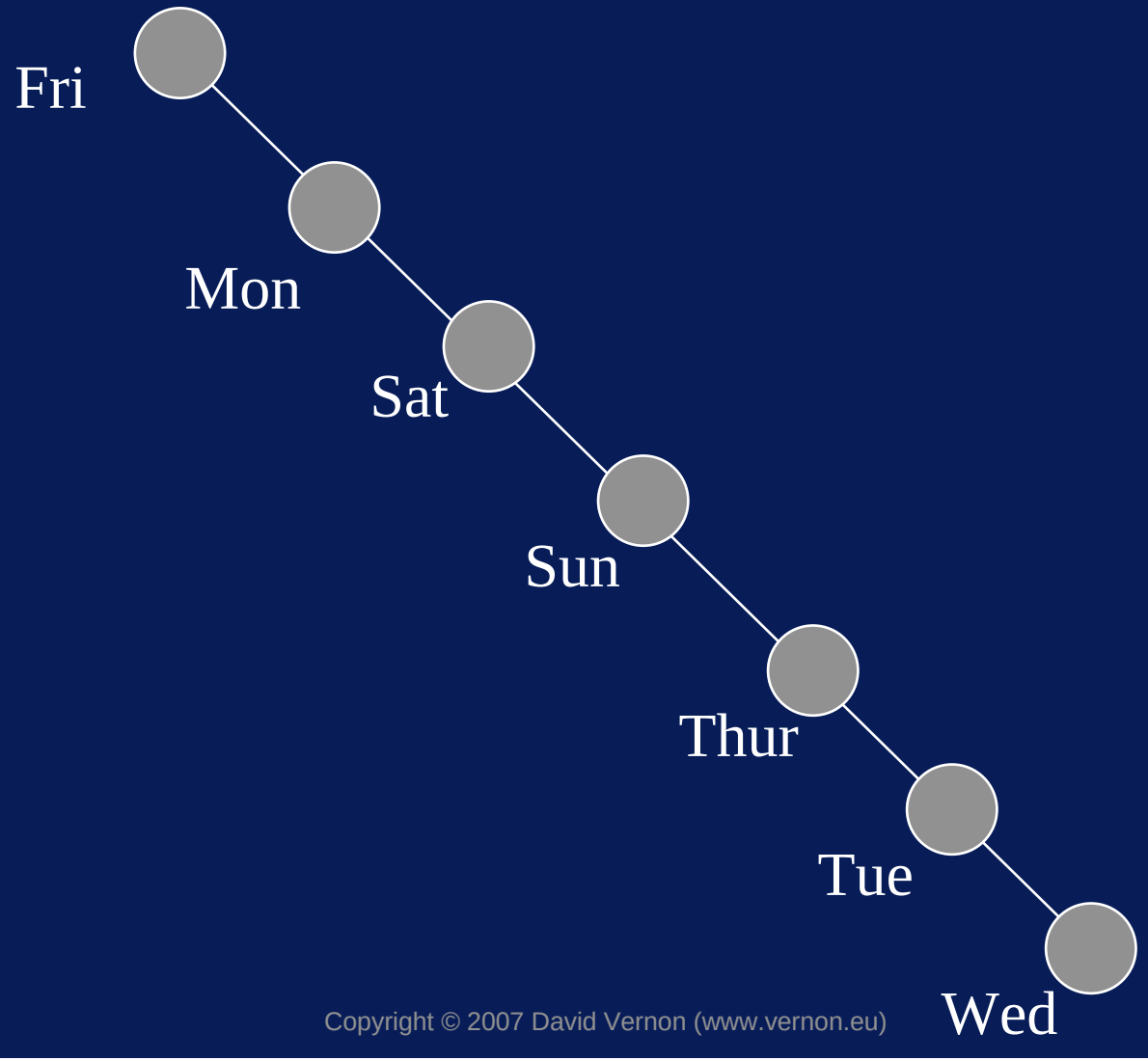
Binary Search Trees

- It should be noted that several binary search trees are possible for a given data set, *e.g,* consider the following tree:

Binary Search Trees



Binary Search Trees



Binary Search Trees

- Let us consider how such a situation might arise. To do so, we need to address how a binary search tree is constructed.
  - Assume we are building a binary search tree of words.
  - Initially, the tree is null, i.e. there are no nodes in the tree.
  - The first word is inserted as a node in the tree as the root, with no children.

Binary Search Trees

- On insertion of the second word, we check to see if it is the same as the key in the root, less than it, or greater than it.
  - » If it is the same, no further action is required (duplicates are not allowed).
  - » If it is less than the key in the current node, move to the left subtree and *compare again*.
  - » If the left subtree does not exist, then the word does not exist and it is inserted as a new node on the left.

Binary Search Trees

- » If, on the other hand, the word was greater than the key in the current node, move to the right subtree and compare again.
- » If the right subtree does not exist, then the word does not exist and it is inserted as a new node on the right.

– This insertion can most easily be effected in a recursive manner

Binary Search Trees

- The point here is that the structure of the tree depends on the order in which the data is inserted in the list.
- If the words are entered in sorted order, then the tree will degenerate to a 1-D list.

BST Operations

- *Insert*: E $\times$ BST $\rightarrow$ BST :

  The function value *Insert*(*e*,*T*) is the BST *T* with the element e inserted as a leaf node; if the element already exists, no action is taken.

BST Operations

- *Delete*: E $\times$ BST $\rightarrow$ BST :

  The function value *Delete*(*e*,*T*) is the BST *T* with the element e deleted; if the element is not in the BST exists, no action is taken.

Implementation of *Insert*(*e*, *T*)

- If *T* is empty (i.e. *T* is NULL)
  - create a new node for e
  - make *T* point to it
- If *T* is not empty
  - if *e* < element at root of *T*
    - » Insert *e* in left child of *T*: *Insert(e, T(1))*
  - if *e* > element at root of *T*
    - » Insert *e* in right child of *T*: *Insert(e, T(2))*

Implementation of *Insert*(*e*,*T*)

Implementation of *Delete*(*e, T*)

- First, we must locate the element e to be deleted in the tree
  - if e is at a leaf node
    - » we can delete that node and be done
  - if e is at an interior node at w
    - » we can't simply delete the node at w as that would disconnect its children
  - if the node at w has only one child
    - » we can replace that node with its child

Implementation of *Delete*(*e*, *T*)

- if the node at *w* has two children
  - » we replace the node at *w* with the lowest-valued element among the descendents of its right child
  - » this is the left-most node of the right tree
  - » It is useful to have a function DeleteMin with removes the smallest element from a non-empty tree and returns the value of the element removed

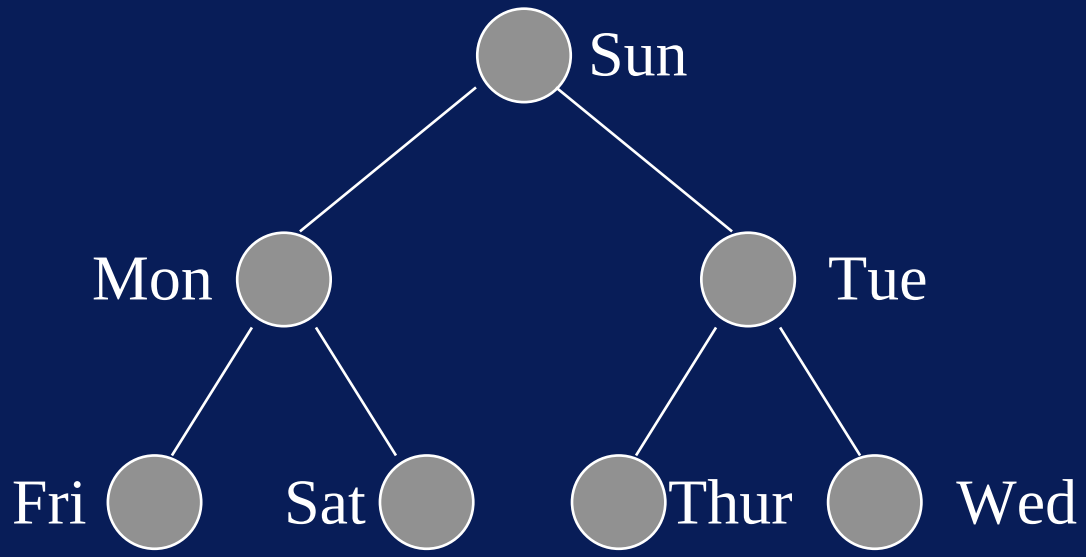
Implementation of *Delete*(*e*, *T*)

- If *T* is not empty
  - if *e* < element at root of *T*
    - » Delete *e* from left child of *T*: *Delete(e, T(1))*
  - if *e* > element at root of *T*
    - » Delete *e* from right child of *T*: *Delete(e, T(2))*
  - if *e* = element at root of *T* and both children are empty
    - » Remove *T*
  - if *e* = element at root of *T* and left child is empty
    - » Replace *T* with *T(2)*

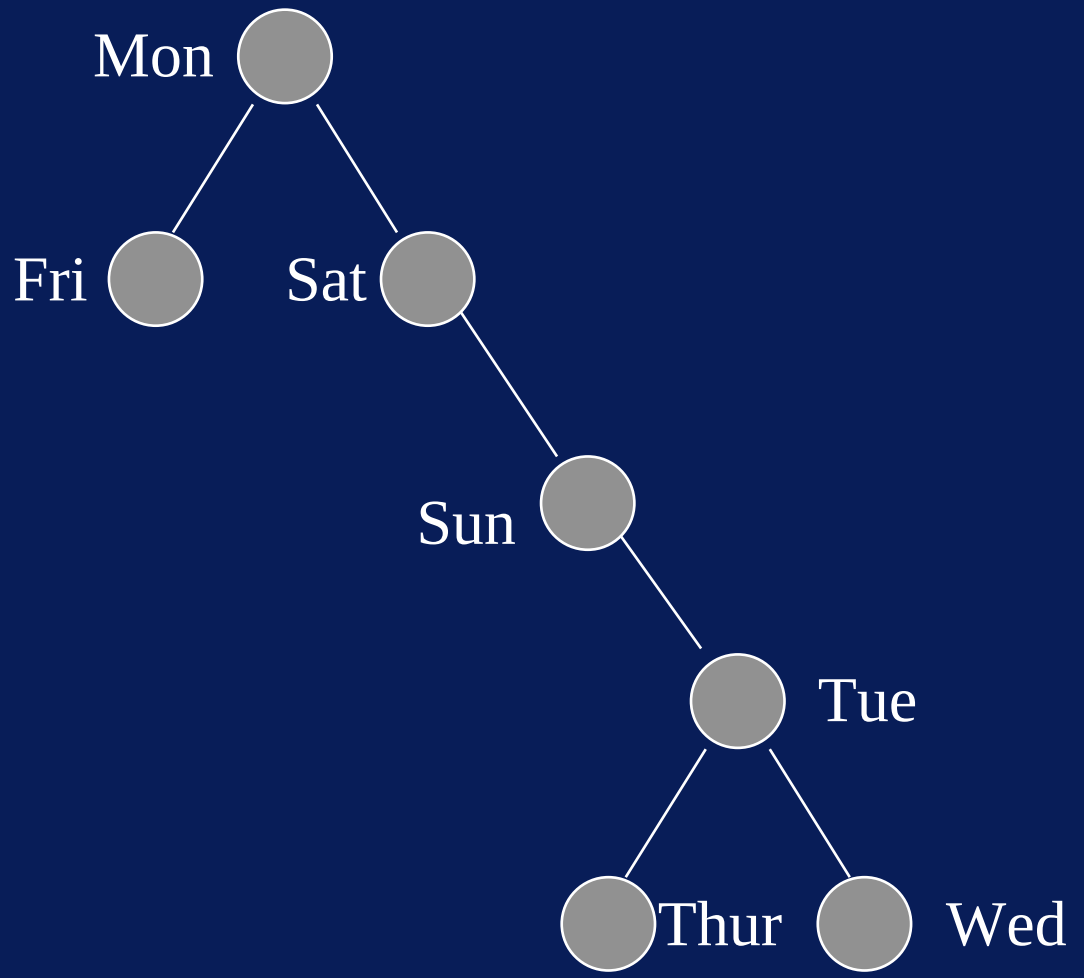
Implementation of *Delete*(*e*, *T*)

- if *e* = element at root of *T* and right child is empty
  - » Replace *T* with *T(1)*
- if *e* = element at root of *T* and neither child is empty
  - » Replace *T* with left-most node of *T(2)*

Implementation of *Delete*(*e*, *T*)



Implementation of *Delete*(*e*, *T*)



BST Implementation

```
/* implementation of BST ADT */

#include <stdio.h>
#include <math.h>
#include <string.h>

#define FALSE 0
#define TRUE 1

typedef struct {
           int number;
           char *string;
         } ELEMENT_TYPE;
```

BST Implementation

```
typedef struct node *NODE_TYPE;

typedef struct node{
          ELEMENT_TYPE element;
          NODE_TYPE left, right;
        } NODE;

typedef NODE_TYPE BST_TYPE;
typedef NODE_TYPE WINDOW_TYPE;
```

BST Implementation

```
/*** insert an element in a BST ***/

BST_TYPE *insert(ELEMENT_TYPE e, BST_TYPE *tree) {
   WINDOW_TYPE temp;
   if (*tree == NULL) {
      /* we are at an external node: create a new node */
      /* and insert it                                  */
      if ((temp =(NODE_TYPE) malloc(sizeof(NODE))) = NULL)
         error(“insert: unable to allocate memory”);
      else {
         temp->element = e;
         temp->left = NULL;
         temp->right = NULL;
         *tree = temp;
      }
```

BST Implementation

```
    }
    else if (e.number < (*tree)->element.number) {
        /* assume number field is the key */
        insert(e, &((*tree)->left));
    }
    else if (e.number > (*tree)->element.number) {
        insert(e, &((*tree)->right));
    }

    /* if e.number == (*tree)->element.number, e is */
    /* already in the tree so do nothing            */

    return(tree);
}
```

BST Implementation

```
/*** return and delete the smallest node in a tree ***/
/*** i.e. return and delete the left-most node     ***/

ELEMENT_TYPE delete_min(BST_TYPE *tree) {
   ELEMENT_TYPE e;
   BST_TYPE p;
   if ((*tree)->left == NULL) {

      /* (*tree) points to the smallest element */

      e = (*tree)->element;

      /* replace the node pointed to by tree */
      /* by its right child                  */
```

BST Implementation

```
        p = *tree;
        *tree = (*tree)->right;
        free(p);

        return (e);
    }
    else {

        /* the node pointed to by *tree has a left child */

        return(delete_min(&((*tree)->left)));
    }
}
```

BST Implementation

```
/*** delete an element from a BST ***/

BST_TYPE *delete(ELEMENT_TYPE e, BST_TYPE *tree) {
   BST_TYPE p;;
   if (*tree != NULL) {
      if (e.number < (*tree)->element.number)
         delete(e, &((*tree)->left));
      else (e.number > (*tree)->element.number)
         delete(e, &((*tree)->right));
      else if (((*tree)->left == NULL) &&
               ((*tree)->right == NULL)) {

         /* leaf node containing e: delete it */
```

BST Implementation

```
        /* leaf node containing e: delete it */

        p = *tree;
        free(p);
        *tree = NULL;
    }
    else if ((*tree)->left == NULL) {
        /* internal node containing e and it has only */
        /* a right child; delete it and make tree     */
        /* point to the right child                   */

        p = *tree;
        *tree = (*tree)->right;
        free(p);
    }
```

BST Implementation

```
else if ((*tree)->right == NULL) {

   /* internal node containing e and it has only */
   /* a left child; delete it and make tree     */
   /* point to the left child                   */

   p = *tree;
   *tree = (*tree)->left;
   free(p);
}
```

BST Implementation

```
        else {
            /* internal node containing e and it has both */
            /* left and right children; replace it with   */
            /* the leftmost node of the right child       */

            (*tree)->element = delete_min(&((*tree)->right));

        }
    }
}
```

BST Implementation

```
/*** inorder traversal of a tree,                          ***/
/*** printing node elements                                ***/
/*** parameter n is the current level in the tree          ***/

int inorder(BST_TYPE *tree, int n) {
   int i;
   if (*tree != NULL) {
      inorder(tree->left, n+1);
      for (i=0; i<n; i++) printf("      ");
      printf("%d %s\n",tree->element.number,
                       tree->element.string);
      inorder(tree->right, n+1);
   }
}
```

BST Implementation

```
/*** print all elements in a tree by traversing   ***/
/*** inorder                                      ***/

int print(BST_TYPE *tree) {

   printf("Contents of tree by inorder traversal: \n");

   inorder(tree, 0);

   printf("-----------");
}
```

BST Implementation

```
/*** error handler: print message passed as argument and
     take appropriate action                              ***/
int error(char *s); {
   printf("Error: %s\n", s);
   exit(0);
}

/*** assign values to an element ***/

int assign_element_values(ELEMENT_TYPE *e, int number,
   char s[]) {
   e->string = (char *) malloc(sizeof(char) * strlen(s));
   strcpy(e->string, s);
   e->number = number;
}
```

BST Implementation

```
/*** main driver routine ***/

   ELEMENT_TYPE e;
   BST_TYPE list;
   int i;

   print(tree);

   assign_element_values(&e, 3, “...”);
   insert(e, &tree);
   print(tree);

   assign_element_values(&e, 1, “+++”);
   insert(e, &tree);
   print(tree);
```

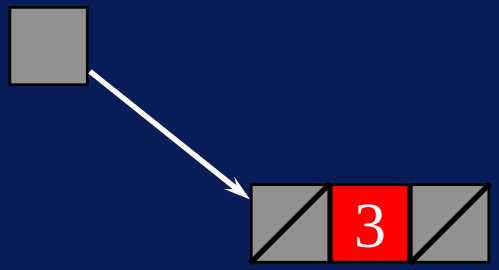
BST Implementation

```
assign_element_values(&e, 5, “---”);
insert(e, &tree);
print(tree);

assign_element_values(&e, 2, “,,,”);
insert(e, &tree);
print(tree);

assign_element_values(&e, 4, “***”);
insert(e, &tree);
print(tree);

assign_element_values(&e, 6, “000”);
insert(e, &tree);
print(tree);
```

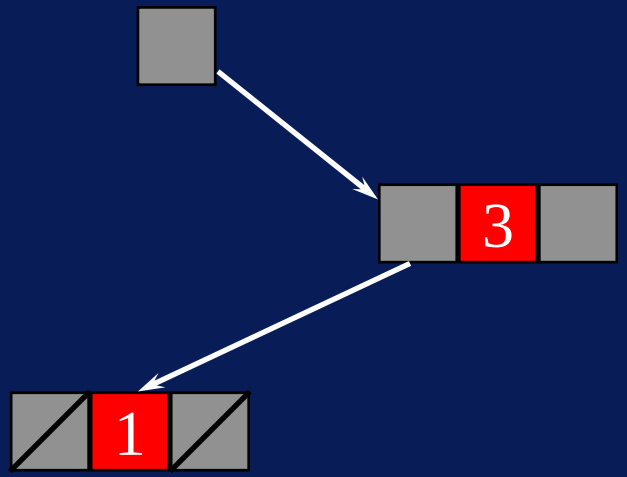
BST Implementation

```
    assign_element_values(&e, 3, “...”);
    insert(e, &tree);
    print(tree);
}
```

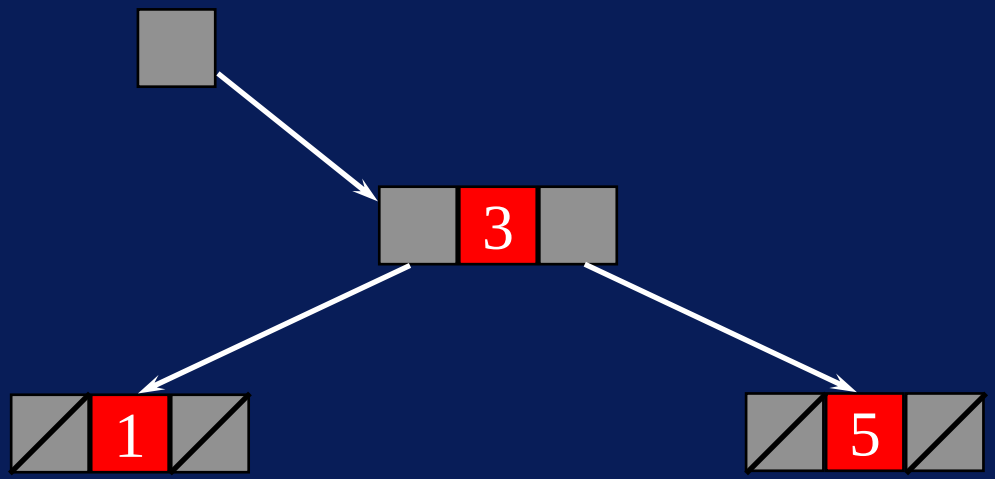
BINARY_TREE Implementation



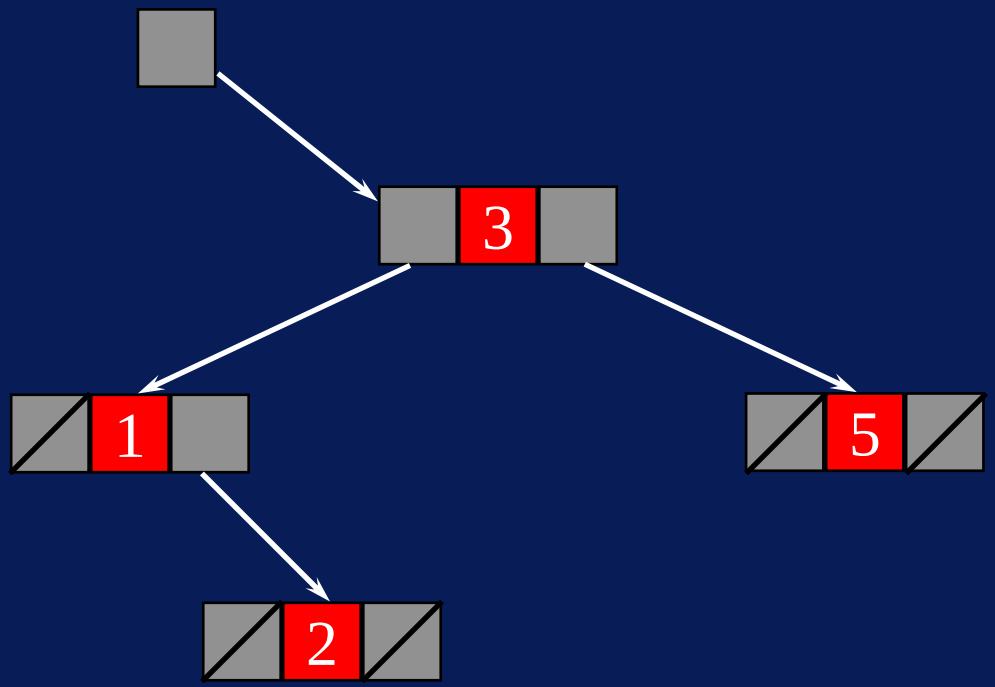
BINARY_TREE Implementation



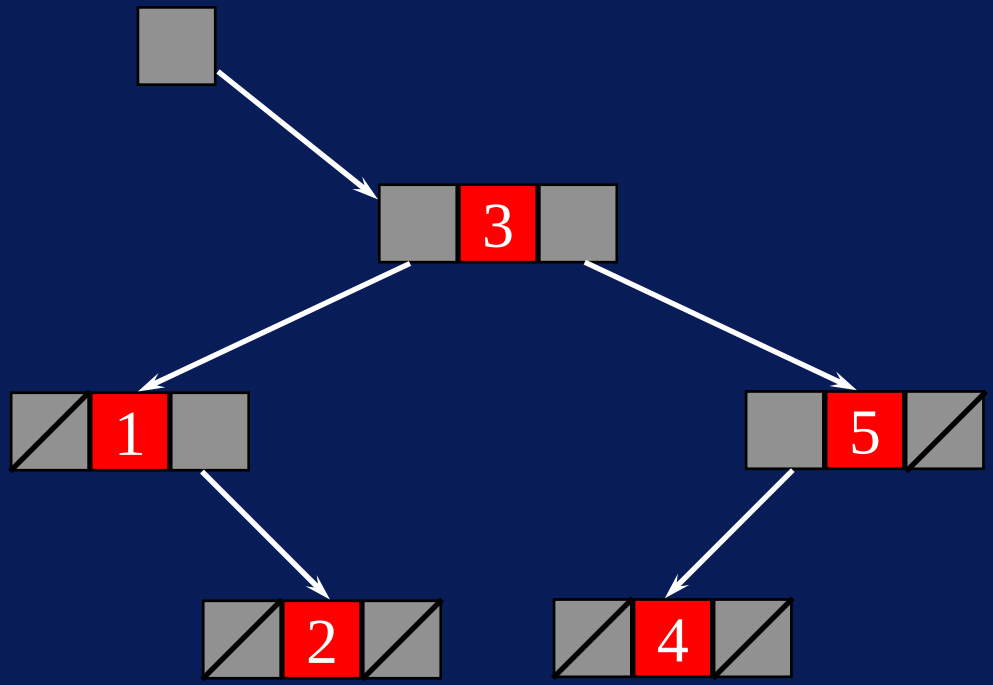
BINARY_TREE Implementation



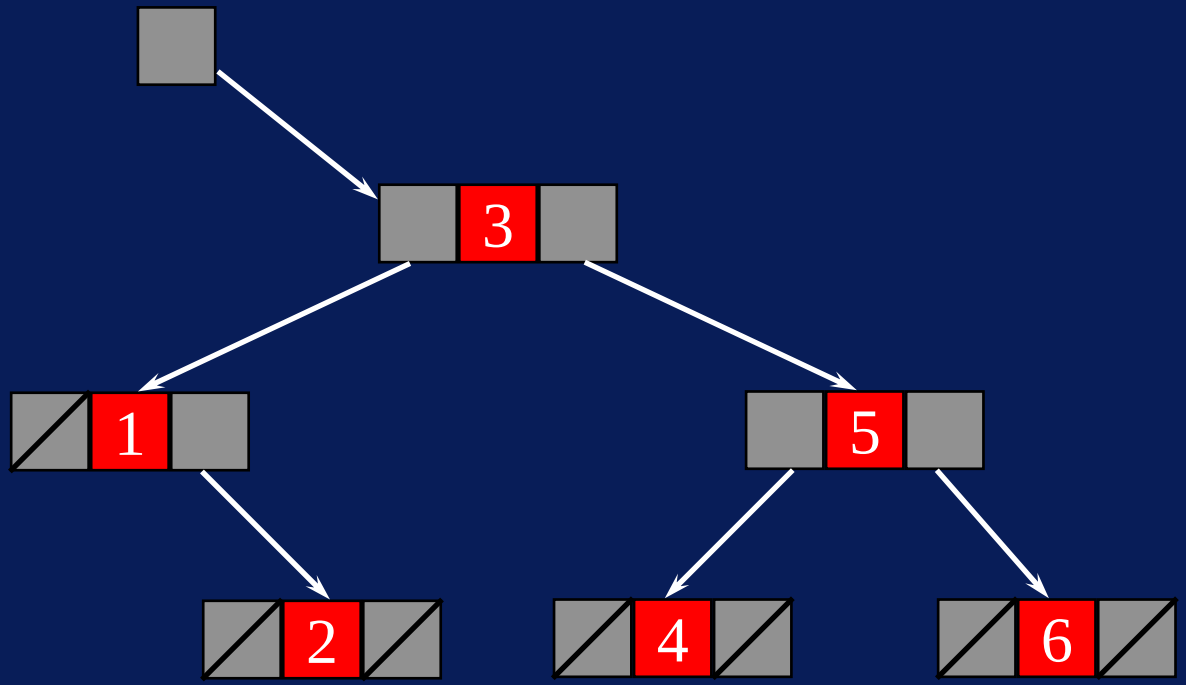
BINARY_TREE Implementation



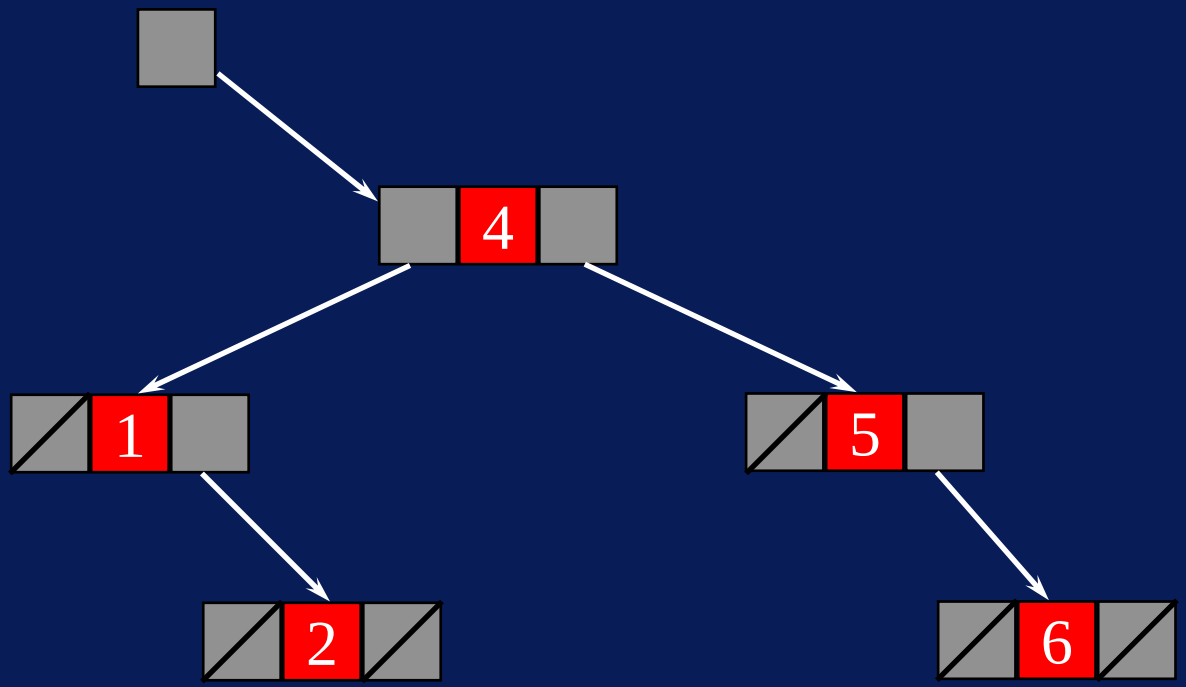
BINARY_TREE Implementation



BINARY_TREE Implementation



BINARY_TREE Implementation



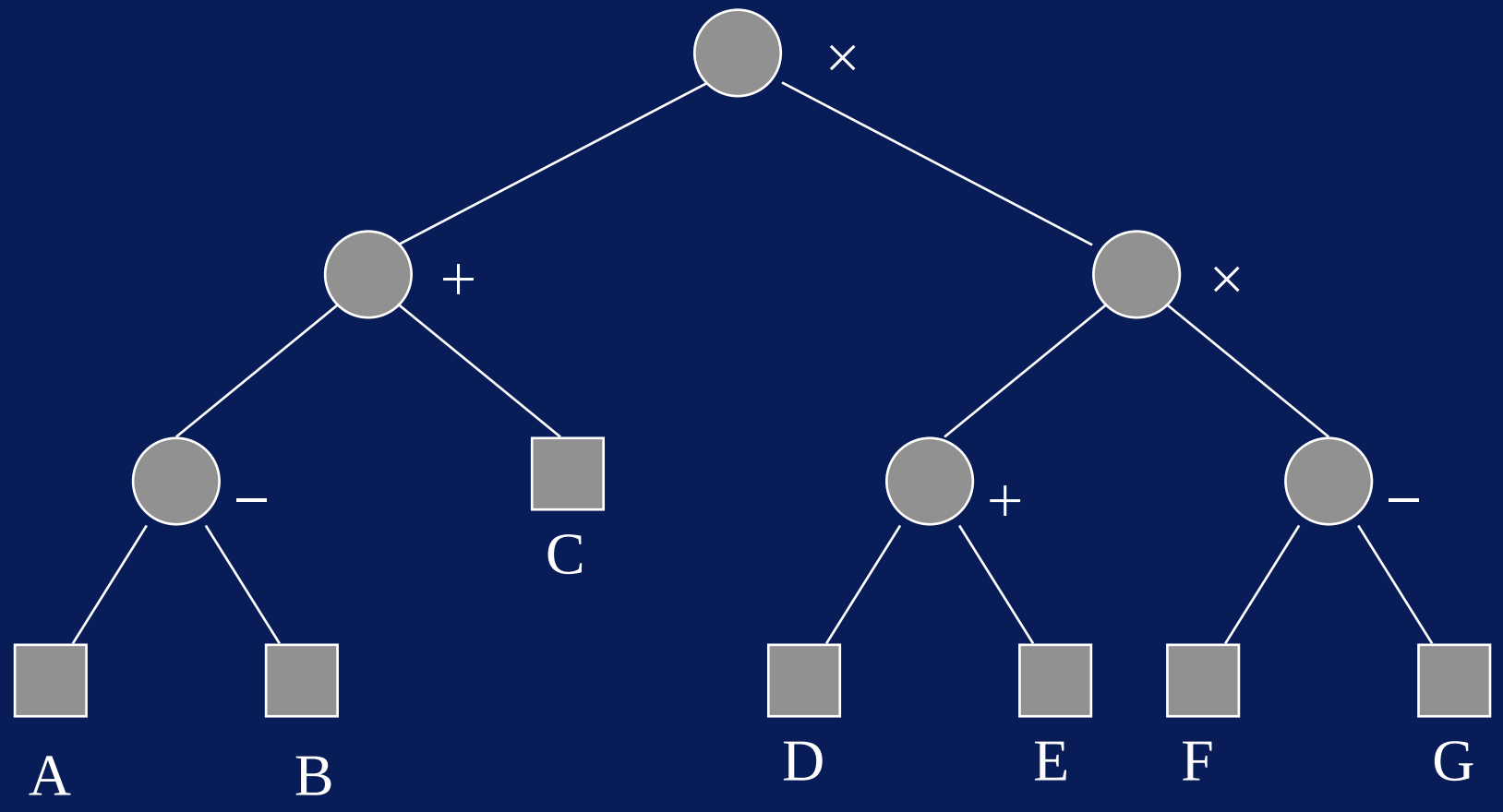
Tree Traversals

- To perform a traversal of a data structure, we use a method of visiting every node in some predetermined order
- Traversals can be used
  - to test data structures for equality
  - to display a data structure
  - to construct a data structure of a give size
  - to copy a data structure

Depth-First Traversals

- There are 3 depth-first traversals
  - inorder
  - postorder
  - preorder
- For example, consider the expression tree:

Example: Expression Tree



Depth-First Traversals

- Inorder traversal
  A – B + C × D + E × F – G
- Postorder traversal
  A B – C + D E + F G – × ×
- Preorder traversal
  × +–A B C × + D E – F G

Depth-First Traversals

- The parenthesised Inorder traversal

  $((A - B) + C) \times ((D + E) \times (F - G))$

  This is the infix expression corresponding to the expression tree
- Postorder traversal gives a postfix expression
- Preorder traversal gives a prefix expression

Depth-First Traversals

- Recursive definition of inorder traversal

  Given a binary tree *T*

  if *T* is empty

  visit the external node

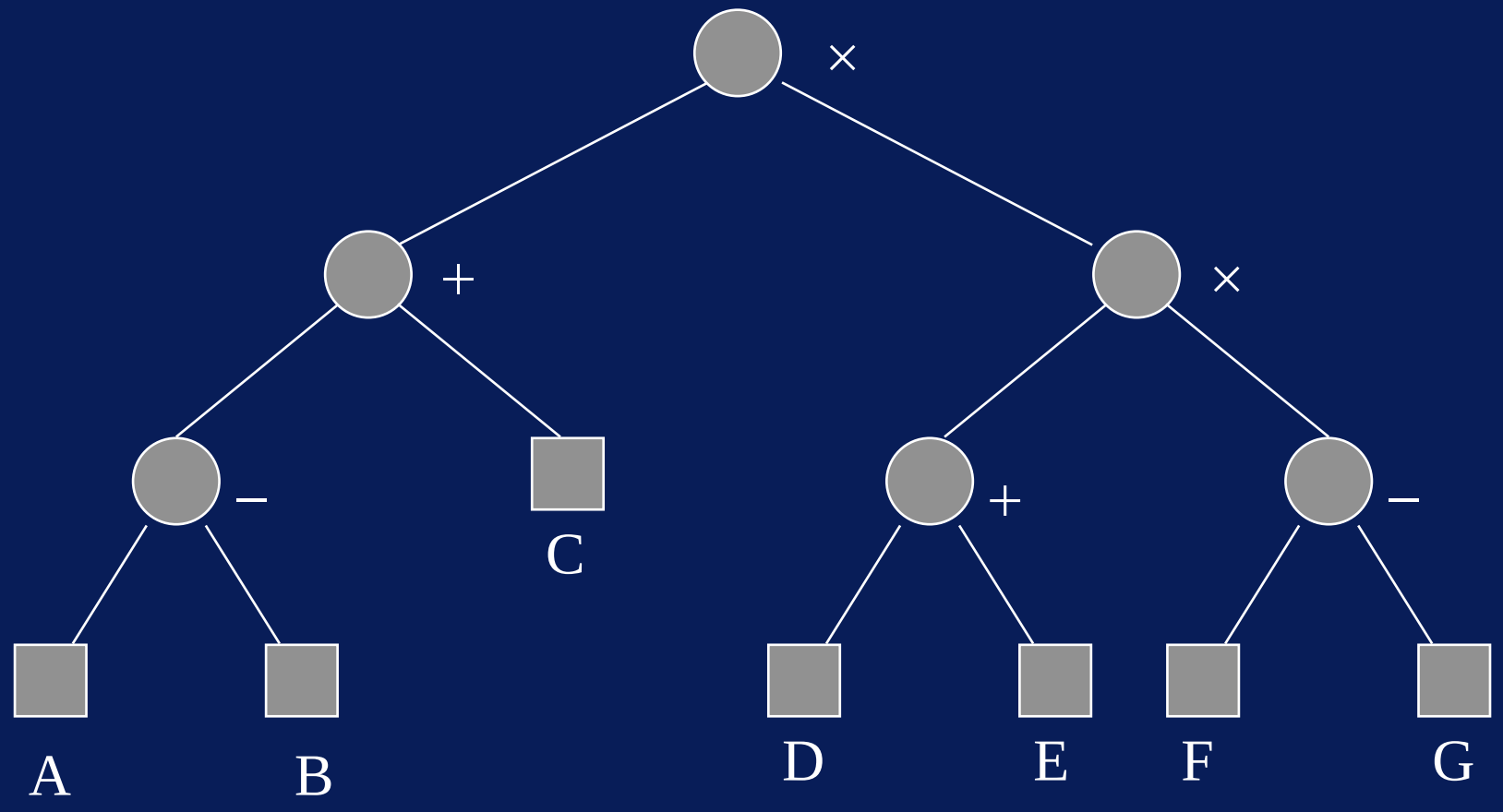
  otherwise

  perform an inorder traversal of *Left*(*T*)

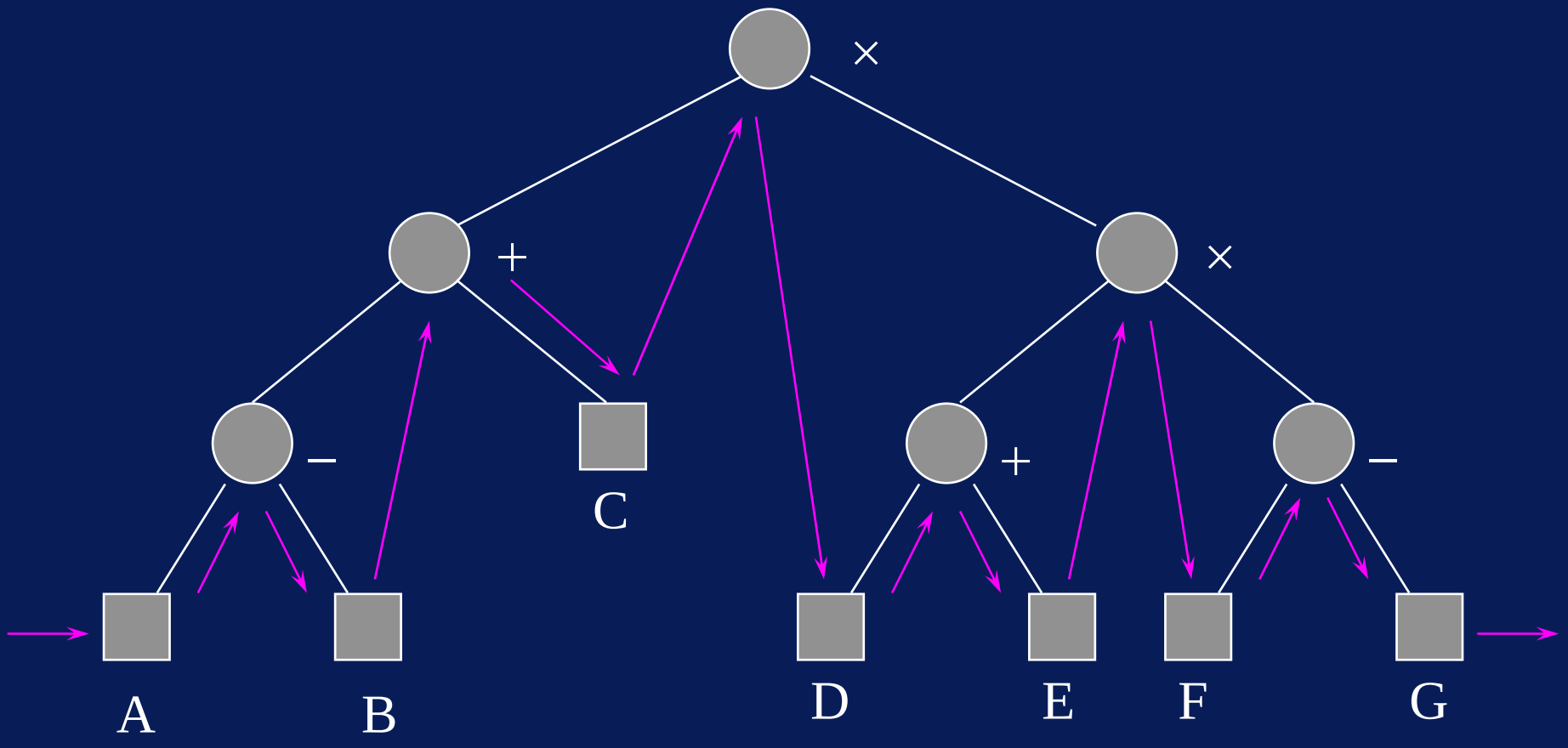
  visit the root of *T*

  perform an inorder traversal of *Right*(*T*)

Example: Inorder Traversal



Example: Inorder Traversal



Depth-First Traversals

- Recursive definition of postorder traversal

  Given a binary tree *T*

  if *T* is empty

  visit the external node

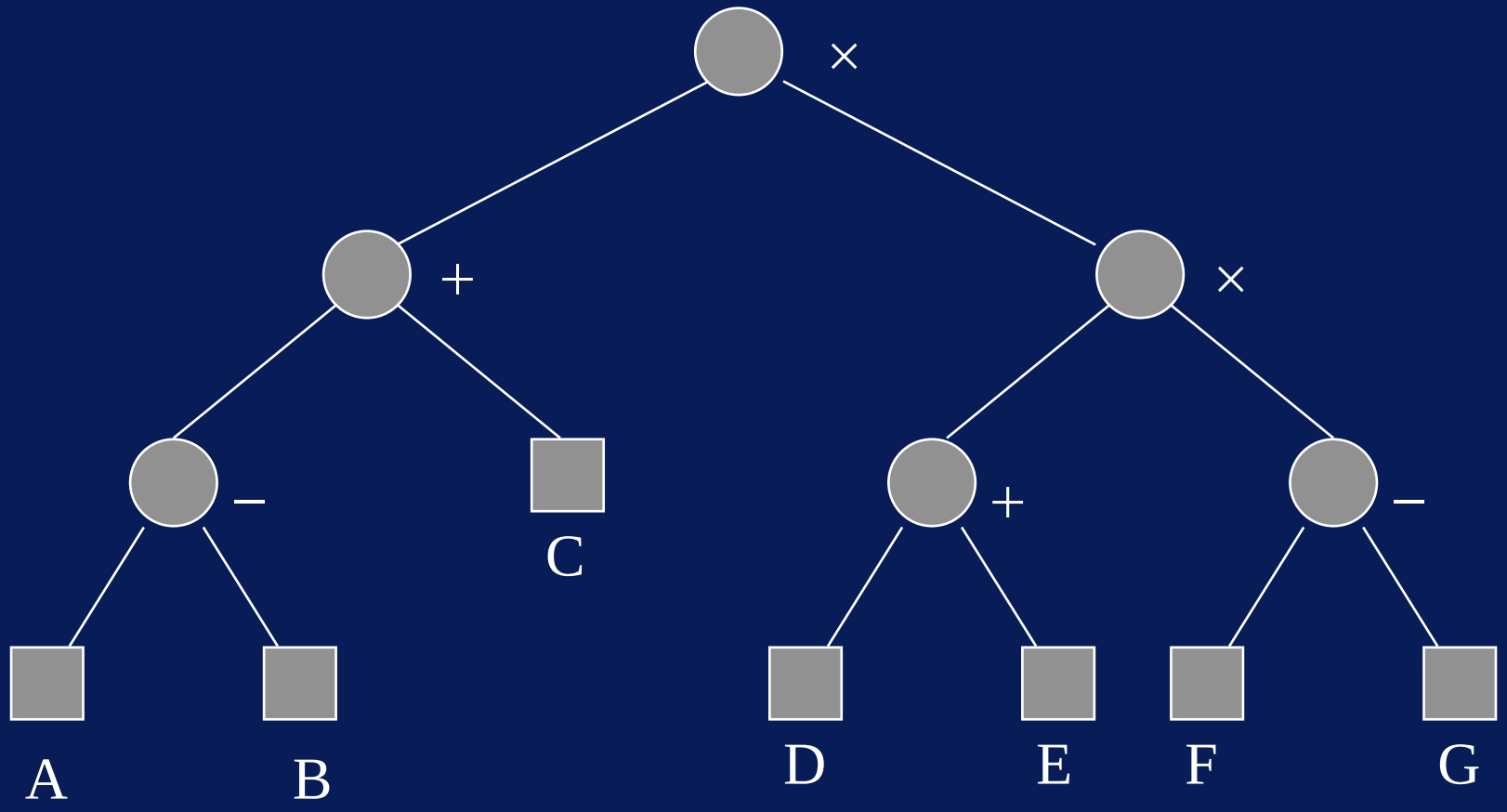
  otherwise

  perform an postorder traversal of *Left*(*T*)

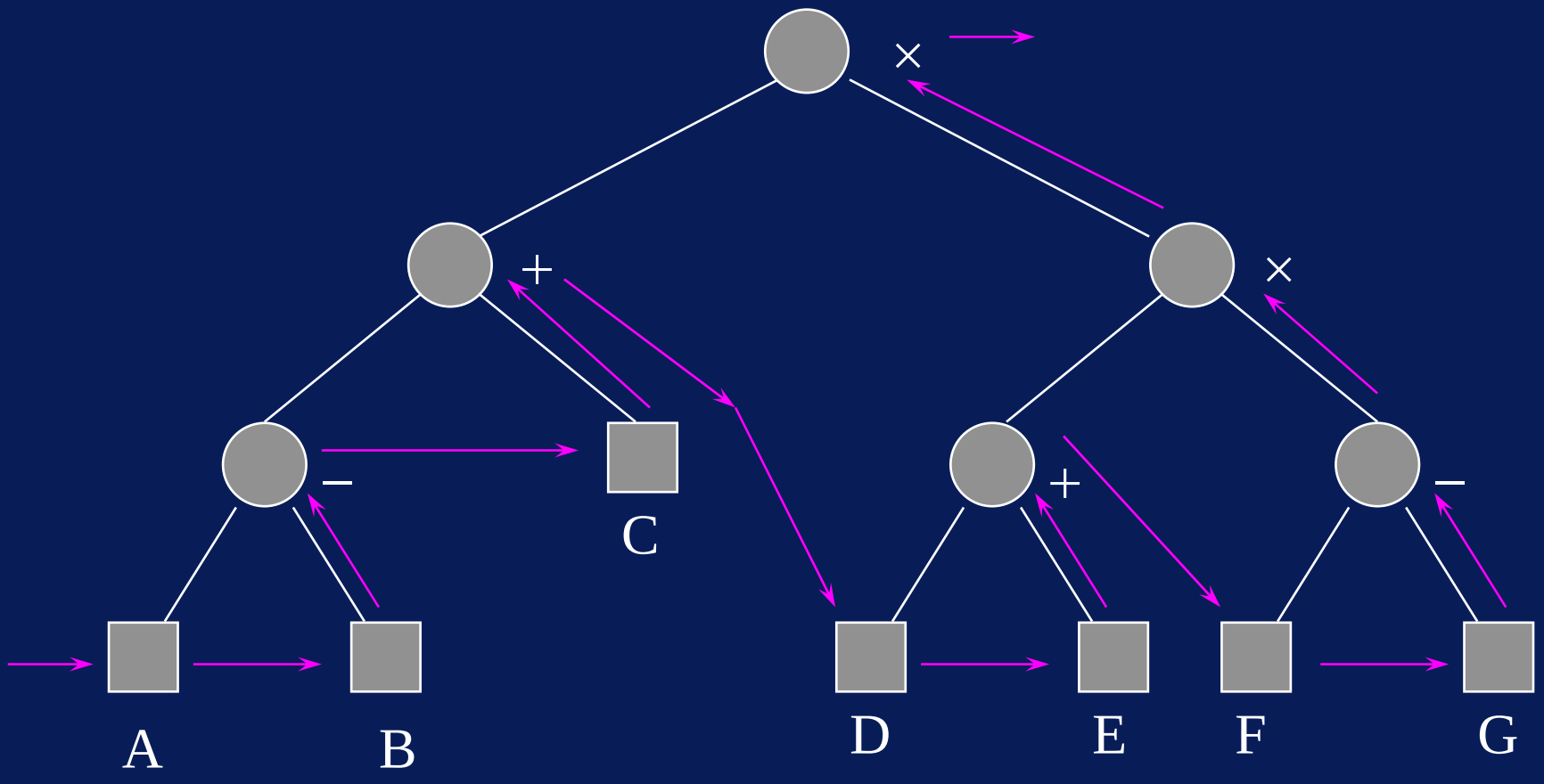
  perform an postorder traversal of *Right*(*T*)

  visit the root of *T*

Example: Postorder Traversal



Example: Postorder Traversal



Depth-First Traversals

- Recursive definition of preorder traversal

  Given a binary tree *T*

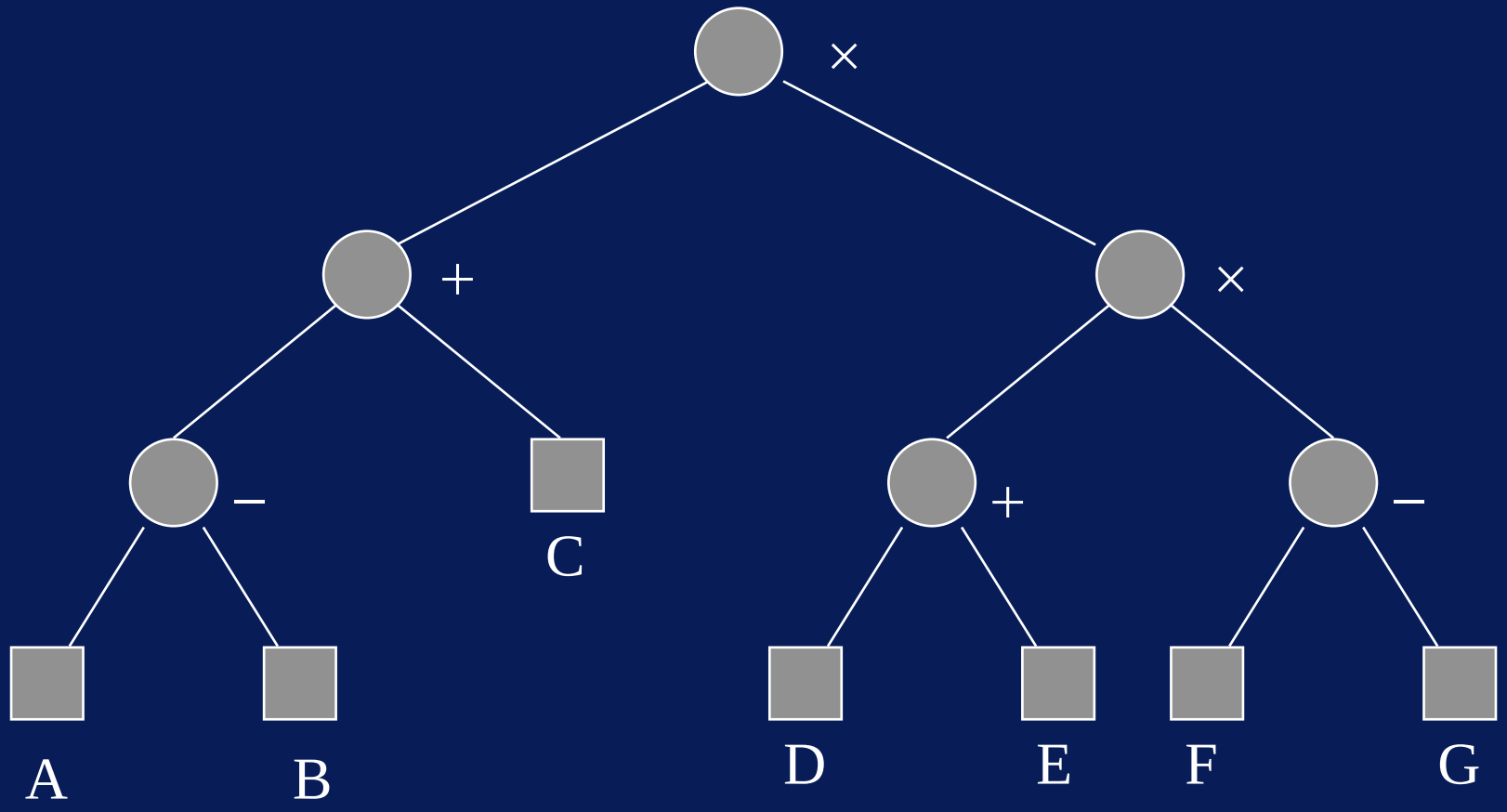
  if *T* is not an external node

  visit the root of *T*

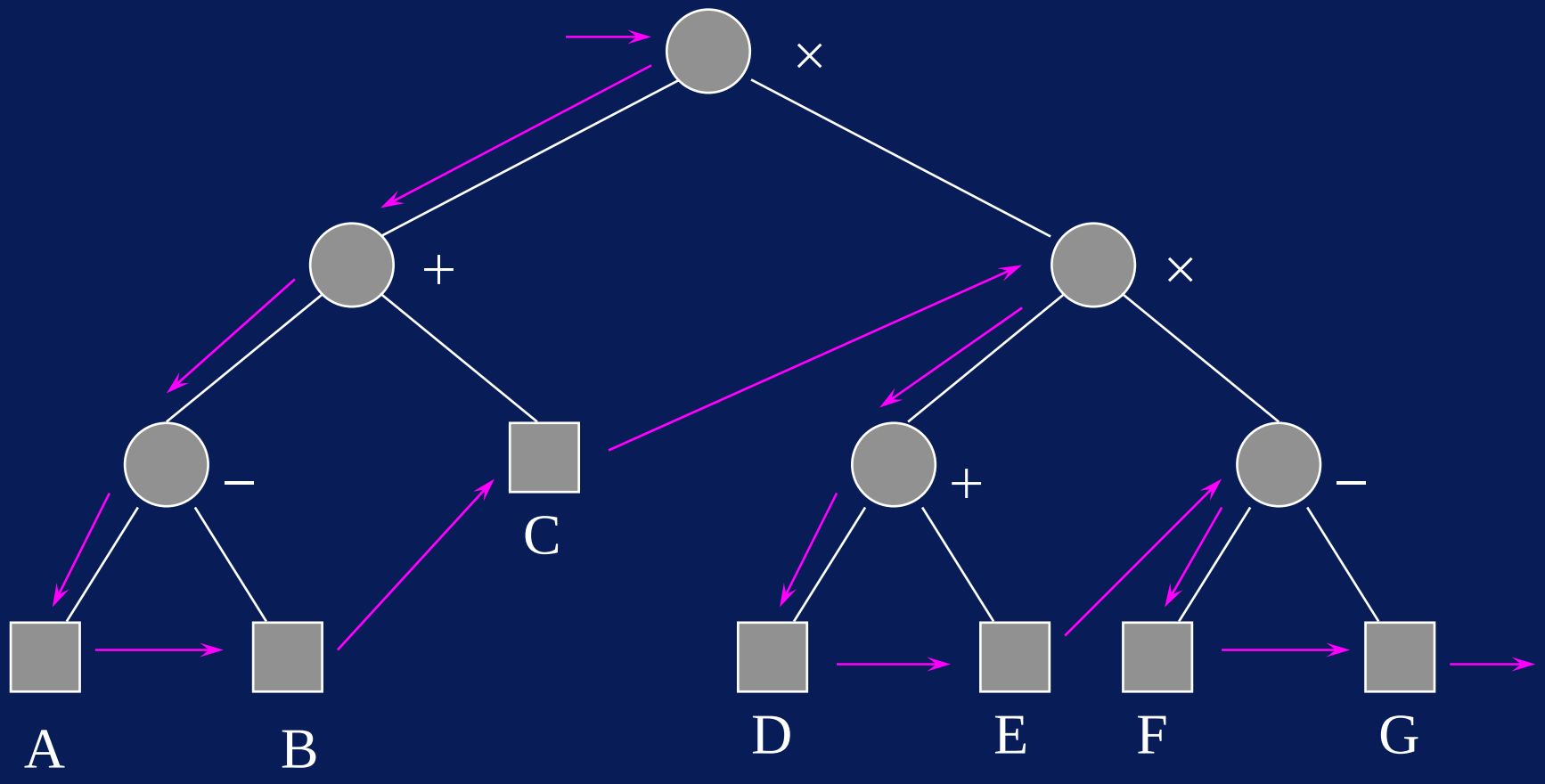
  perform an inorder traversal of *Left*(*T*)

  perform an inorder traversal of *Right*(*T*)

Example: Preorder Traversal



Example: Preorder Traversal



Applications of Trees

- First application: coding and data compression
- We will define optimal variable-length binary codes and code trees
- We will study Huffman's algorithm which constructs them
- Huffman's algorithm is an example of a Greedy Algorithms, an important class of simple optimization algorithms

Text, Codes, and Compression

- Computer systems represent data as bit strings
- Encoding: transformation of data into bit strings
- Decoding: transformationof bit strings into data
- The code defines the transformation

Text, Codes, and Compression

- For example: ASCII, the international coding standard, uses a 7-bit code
- HEX Code - Character
- 20 - <space>
- 41 - A
- 42 - B
- 61 - a

Text, Codes, and Compression

- Such encodings are called
  - fixed-length or
  - block codes
- They are attractive because the encoding and decoding is extremely simple
  - For coding, we can use a block of integers or codewords indexed by characters
  - For decoding, we can use a block of characters indexed by codewords

Text, Codes, and Compression

- For example: the sentence
  The cat sat on the mat

  is encoded in ASCII as

  1010100 110100 011001 0101 .....
- Note that the spaces are there simply to improve readability ... they don't appear in the encoded version.

Text, Codes, and Compression

- The following bit string is an ASCII encoded message:

  1000100110010111000111101111110
  0100110100111011101100111010000
  0110100111100110100001100101111
  0000111100111111001

Text, Codes, and Compression

- And we can decode it by chopping it into smaller strings eachs of 7 bits in length and by replacing the bit strings with their corresponding characters:

  1000100(D)1100101(e)1100011(c)1101111(o)1100100(d)1101001(i)1101110(n)1100111(g)0100000()1101001(i)1110011(s)0100000()1100101(e)1100001(a)1110011(s)1111001(y)

Text, Codes, and Compression

- Every code can be thought of in terns of
- a finite alphabet of source symbols
- a finite alphabet of code symbols
- Each code maps every finite sequence or string of source symbols into a string of code symbols

Text, Codes, and Compression

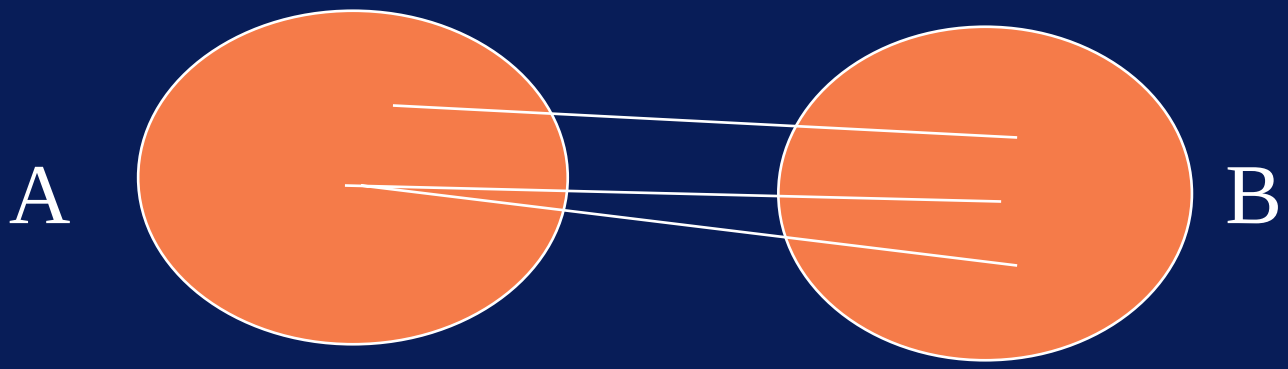
- Let A be the source alphabet
- Let B be the code alphabet
- A code f is an injective map

  f: $S_A \rightarrow S_B$

- where S_A is the set of all strings of symbols from A
- where S_B is the set of all strings of symbols from B

Text, Codes, and Compression

- Injectivity ensures that each encoded string can be decoded uniquely (we do not want two source strings that are encoded as the same string)



Injective Mapping: each element in the range is related to at most one element in the domain

Text, Codes, and Compression

- We are primarily interested in the code alphabet {0, 1} since we want to code source symbols strings as bit strings

Text, Codes, and Compression

- There is a problem with block codes: n symbols produce nb bits with a block code of length b
- For example,
  - if n = 100,000 (the number of characters in a typical 200-page book)
  - b = 7 (e.g. 7-bit ASCII code)
  - then the characters are encoded as 700,000 bits

Text, Codes, and Compression

- While we cannot encode the ASCII characters with fewer than 7 bits
- We can encode the characters with a different number of bits, depending on their frequency of occurence.
- Use fewer bits for the more frequent characters
- Use more bits for the less frequent characters
- Such a code is called a variable-length

Text, Codes, and Compression

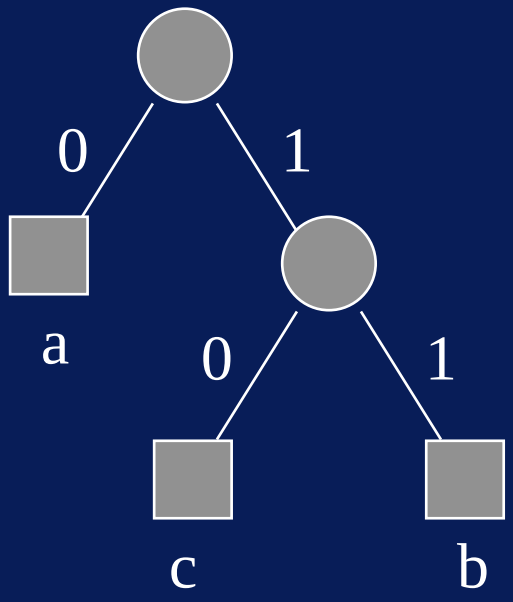
- First problem with variable length codes:
  - when scanning an encoded text from left to right (decoding it)
  - How do we know when one codeword finishes and another starts?
- We require each codeword not be a prefix of any other codeword
- So, for the binary code alphabet, we should base the codes on binary code

Text, Codes, and Compression

- Binary code trees:
- binary tree whose external nodes are labelled uniquely with the source alphabet symbols
- Left branches are labelled 0
- Right branches are labelled 1

Text, Codes, and Compression

A binary code tree and its prefix code



a 0
b 11
c 10

Text, Codes, and Compression

- The codeword corresponding to a symbol is the bit string given by the path from the root to the external node labeled with the symbol
- Note that, as required, no codeword is a prefix for any other codeword
  - This follows directly from the fact that source symbols are only on external nodes
  - and there is only one (unique) path to that symbol

Text, Codes, and Compression

- Codes that satisfy the prefix property are called prefix codes
- Prefix codes are important because
  - we can uniquely decode an encoded text with a left-to-right scan of the encoded text
  - by consideringly only the current bit in the encoded text
  - decoder uses the code tree for this purpose

Text, Codes, and Compression

- Read the encoded message bit by bit
- Start at the root
- if the bit is a 0, move left
- if the bit is a 1, move right
- if the node is external, output the corresponding symbol and begin again at the root

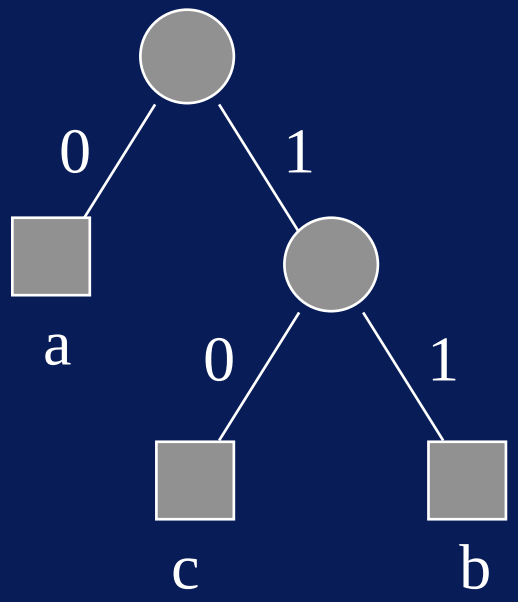
Text, Codes, and Compression

- Encoded message:

  0 0 1 1 1 0 0

- Decoded message:

  A A B C A



Optimal Variable-Length Codes

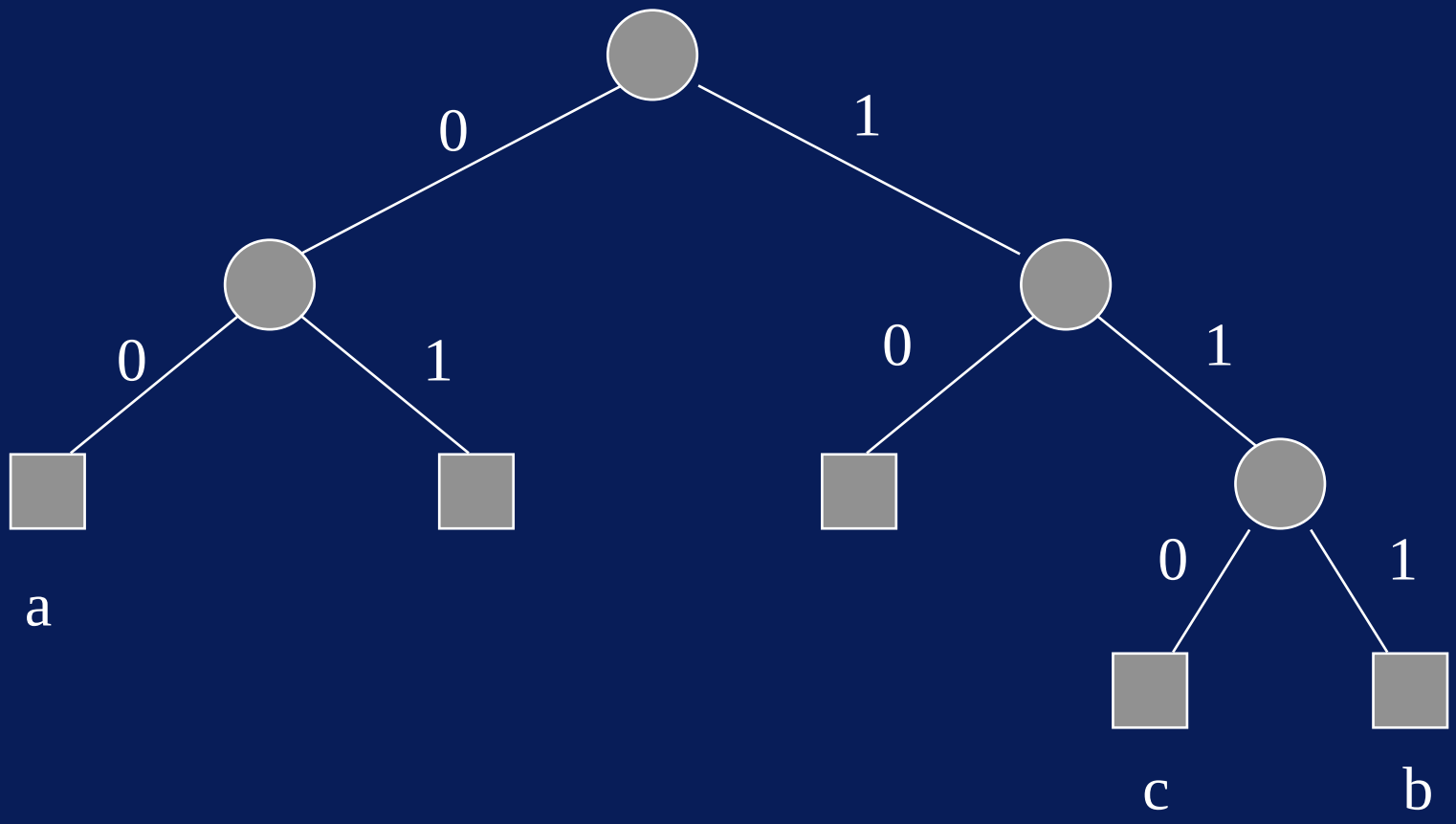
- What makes a good variable length code?
- Let A = a_1, ..., a_n, n>=1, be the alphabet of source symbols
- Let P = p_1, ..., p_n, n>=1, be their probability of occurrence
- We obtain these probabilities by analysing are representative sample of the type of text we wish to encode

Optimal Variable-Length Codes

- Any binary tree with n external nodes labelled with the n symbols defines a prefix code
- Any prefix code for the n symbols defines a binary tree with at least n external nodes
- Such a binary tree with exactly n external nodes is a reduced prefix code (tree)
- Good prefix codes are always reduced

Non-Reduced Prefix Code (Tree)

a 000
b 111
c 110



Optimal Variable-Length Codes

- Comparison of prefix codes - compare the number of bits in the encoded text
- Let $A = a_1, ..., a_n$, $n>=1$, be the alphabet of source symbols
- Let $P = p_1, ..., p_n$ be their probability of occurrence
- Let $W = w_1, ..., w_n$ be a prefix code for $A = a_1, ..., a_n$
- Let $L = l_1, ..., l_n$ be the lengths of $W = w_1, ..., w_n$

Optimal Variable-Length Codes

- Given a source text T with f_1, ..., f_n occurrences of a_1, ..., a_n respectively
- The total number of bits when T is encoded is
  $\sum_{i=1}^{n} f_i l_i$
- The total number of source symbols is
  $\sum_{i=1}^{n} f_i$
- The average length of the W-encoding is
  $Alength(T, W) = \sum_{i=1}^{n} f_i l_i \; / \; \sum_{i=1}^{n} f_i$

Optimal Variable-Length Codes

- For long enough texts, the probability p_i of a given symbol occurring is approximately

  $$p_i = f_i \; / \; \sum_{i=1}^{n} f_i$$

- So the expected length of the W-encoding is

  $$\text{Elength}(W, P) = \sum_{i=1}^{n} p_i \, l_i$$

Optimal Variable-Length Codes

- To compare two different codes W_1 and W_2 we can compare either
  - Alength(T, W_1) and Alength(T, W_2) or
  - Elength(W_1, P) and Elength(W_2 , P)
- We say W_1 is no worse than W_2 if

  Elength(W_1, P) <= Elength(W_2 , P)
- We say W_1 is optimal if

  Elength(W_1, P) <= Elength(W_2 , P)

  for all possible prefix codes W_2 of A

Optimal Variable-Length Codes

- Huffman's Algorithm
- We wish to solve the following problem:
- Given n symbols
  $A = a_1, ..., a_n$, $n>=1$
  and the probability of their occurrence
  $P = p_1, ..., p_n$, respectively,
  construct an optimal prefix code for A and P

Optimal Variable-Length Codes

- This problem is an example of a global optimization problem
- Brute force (or exhaustive search) techniques are too expensive to compute:

  Given A and P
  Compute the set of all reduced prefix codes
  Choose the minimal expected length

Optimal Variable-Length Codes

- This algorithm takes $O(n^n)$ time, where n is the size of the alphabet
- Why? because any binary tree of size n-1 (i.e. with n external nodes) is a valid reduced prefix tree and there are n! ways of labelling the external nodes
- Since n! is approximately n^n we see that there are approximately $O(n^n)$ steps to go through when constructing all the trees to check

Optimal Variable-Length Codes

- Huffman's Algorithm is only $O(n^2)$
- This is significant: if n = 128 (number of symbols in a 7-bit ASCII code)
- $O(n^n) = 128^{128} = 5.28 \times 10^{269}$
- $O(n^2) = 128^2 = 1.6384 \times 10^4$
- There are 31536000 seconds in a year and if we could compute 1000 000 000 steps a second then the brute force technique would still take 1.67×10^{253} years

Optimal Variable-Length Codes

- The age of the universe is estimated to be between 7 and 20 billion years, i.e.,

  $7x10^9$ and $20x10^9$ years

- A long way off 1.67×10^{253} years!

Optimal Variable-Length Codes

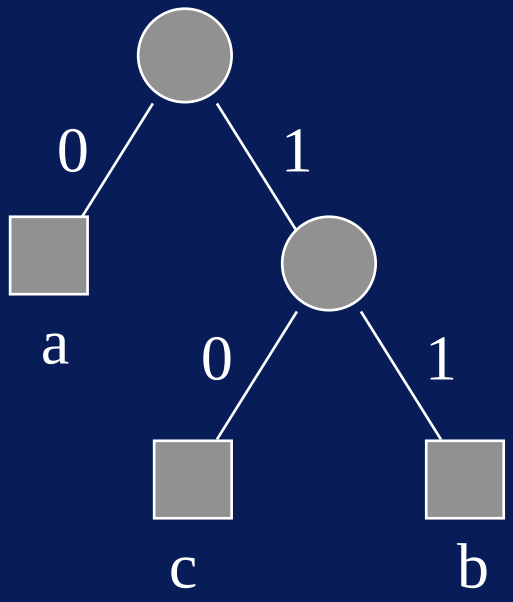
- Huffman's Algorithm uses a technique called Greediness
- It uses local optimization to achieve a globally optimum solution
  - Build the code incrementally
  - Reduce the code by one symbol at each step
  - Merge the two symbols that have the smallest probabilities into one new symbol

Optimal Variable-Length Codes

- Before we begin, note that we'd like a tree with the symbols which have the lowest probability to be on the longest path
- Why?
- Because the length of the codeword is equal to the path length and we want
  - short codewords for high-probability symbols
  - longer codewords for low-probability

Text, Codes, and Compression

A binary code tree and its prefix code



a 0
b 11
c 10

Huffman's Algorithm

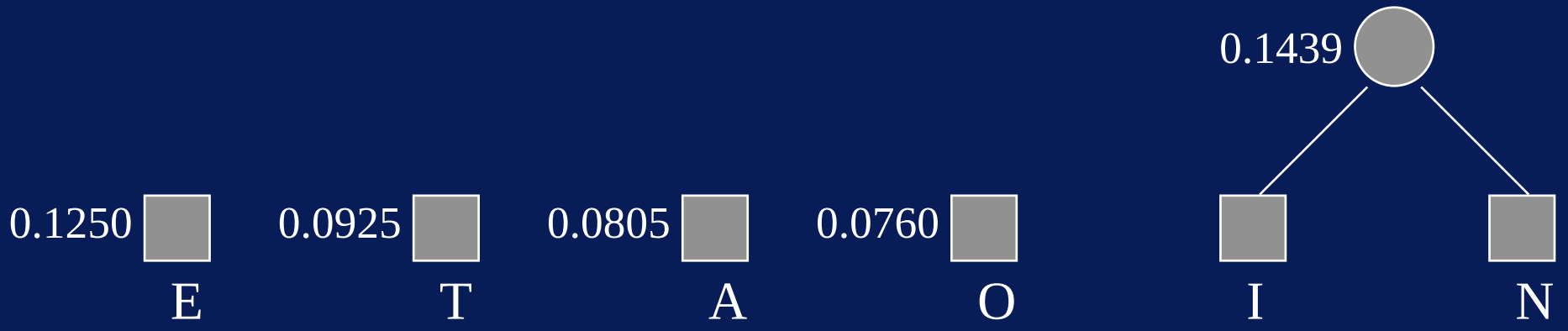
- We will treat Huffman's Algorithm for just six letters, i.e, n = 6, and there are six symbols in the source alphabet.
- These are, with their probabilities,
  - E - 0.1250
  - T - 0.0925
  - A - 0.0805
  - O - 0.0760
  - I - 0.0729
  - N - 0.710

Huffman's Algorithm

- Step 1:
- Create a forest of code trees, one for each symbol
- Each tree comprises a single external node (empty tree) labelled with its symbol and weight (probability)

Huffman's Algorithm

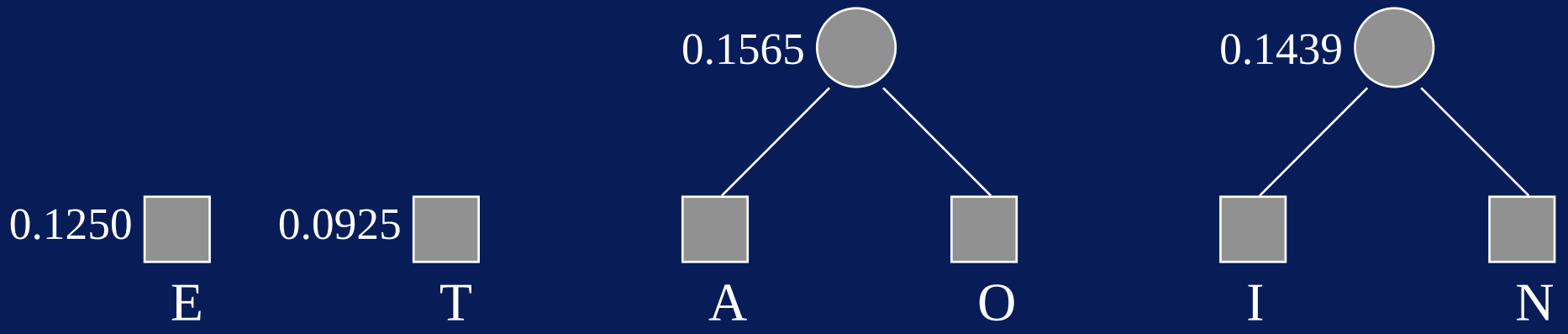
- Step 2:
  - Choose the two binary trees, B1 and B2, that have the smallest weights
  - Create a new root node with B1 and B2 as its children and with weight equal to the sum of these two weights



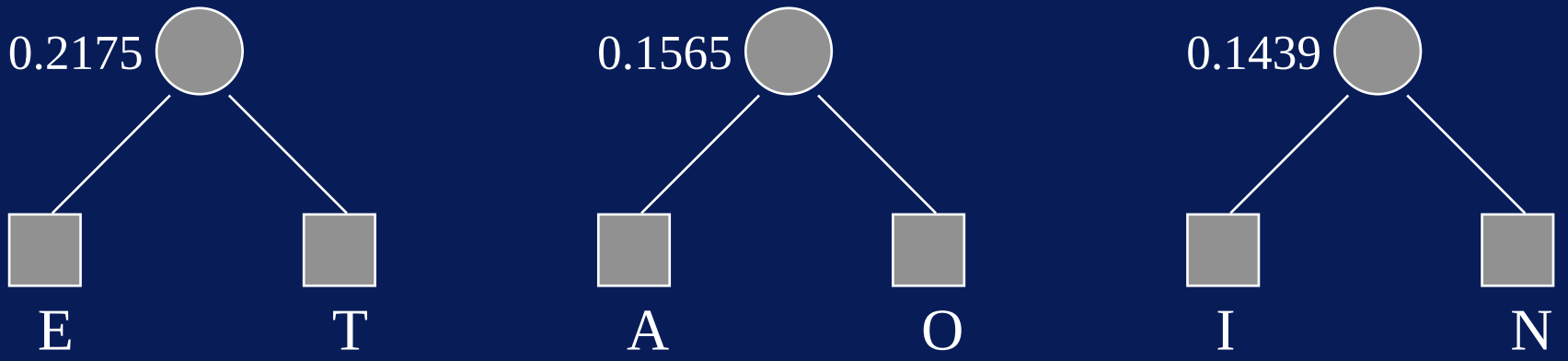
Huffman's Algorithm

- Step 3:
  - Repeat step 2!

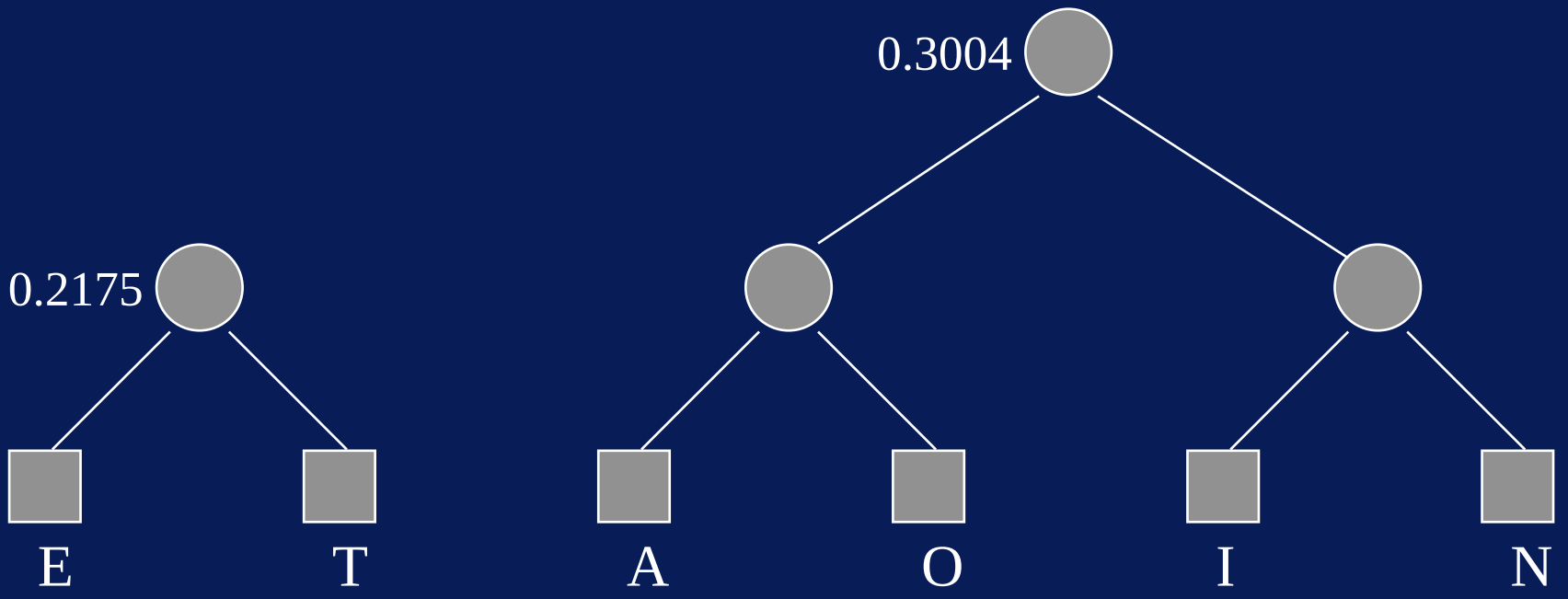
Huffman’s Algorithm



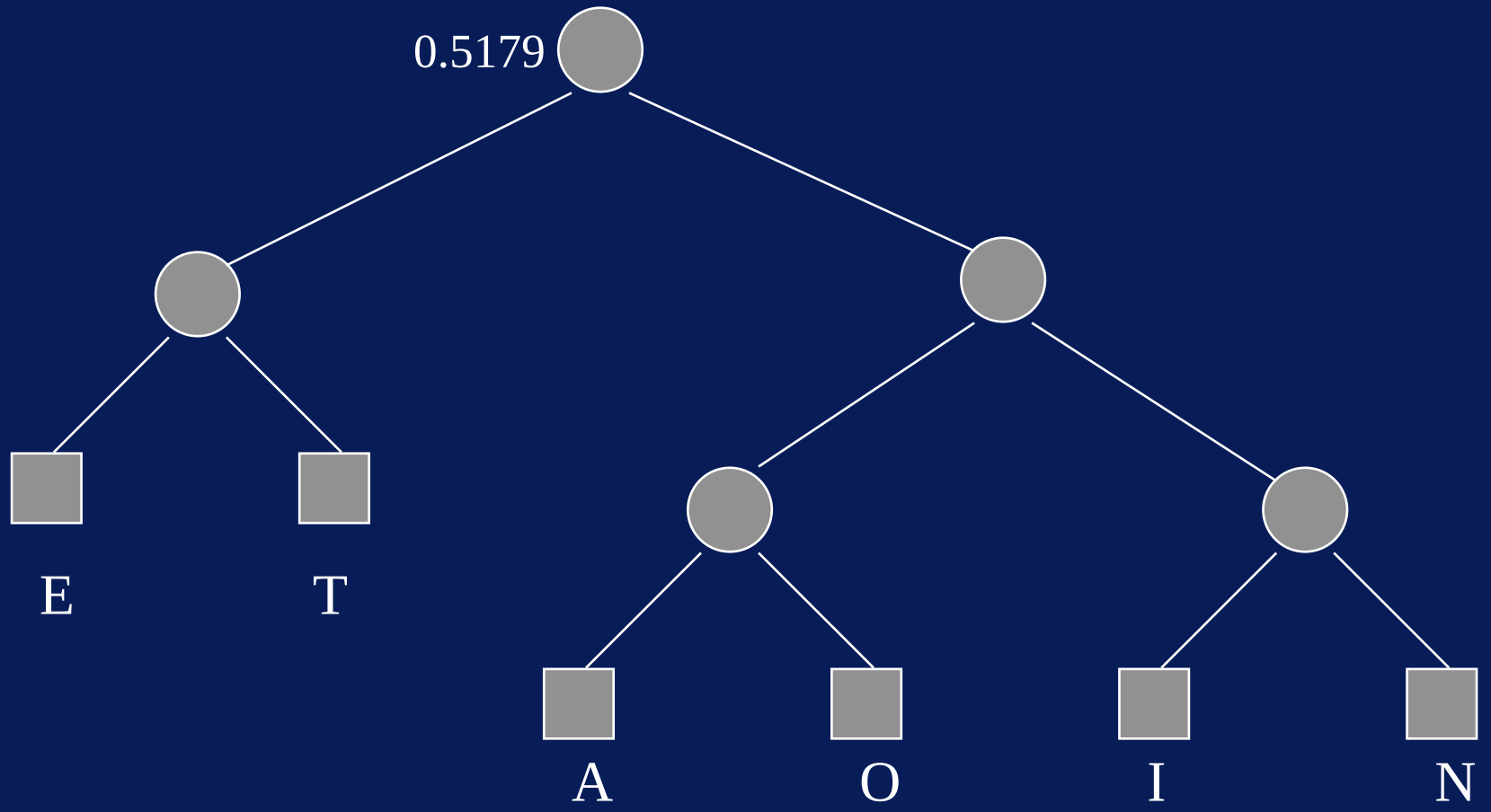
Huffman's Algorithm



Huffman’s Algorithm



Huffman’s Algorithm



Huffman's Algorithm

- The final prefix code is:
  - A 100
  - E 00
  - I 110
  - N 111
  - O 101
  - T 01

Huffman’s Algorithm

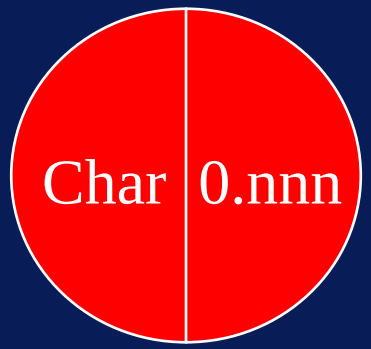
- Three phases in the algorithm
- Initialize the forest of code trees
- Construct an optimal code tree
- Compute the encoding map

Huffman's Algorithm

- Phase 1: Initialize the forest of code trees
  - How will we represent the forest of trees?
  - Better question: how will we represent our tree ... have to store both alphanumeric characters and probabilities?
  - Need some kind of composite node
  - Opt to represent this composite node as an INTERNAL node

Huffman's Algorithm

- Consequently, the initial tree is simply one internal node
- That is, it is a root (with two external nodes)

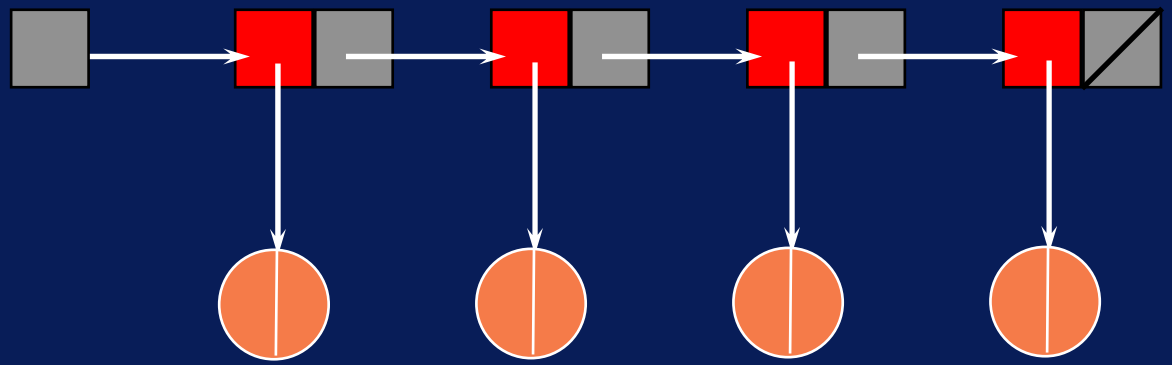


Huffman's Algorithm

- So, to create such a tree we simply invoke the following operations:
  - Initialize the tree … tree()
  - Add a node … addnode(char, weight, T)

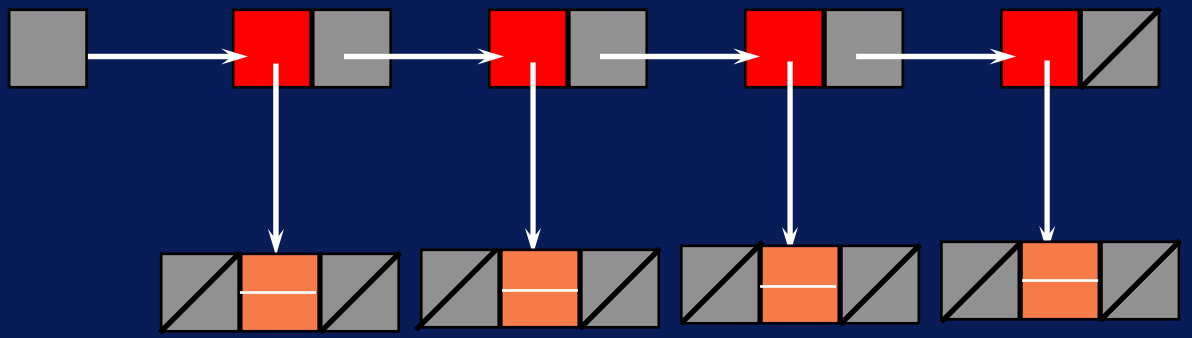
Huffman's Algorithm

- We must also keep track of our forest
- Could represent it as a linked list of pointers to Binary trees ...



Huffman’s Algorithm

- Represented as:



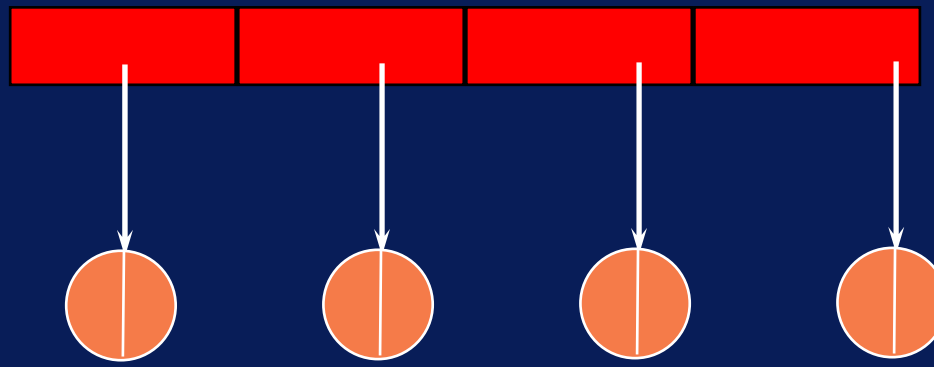
Huffman's Algorithm

- Is there an alternative?
- Question: why do we use dynamic datastructures?
- Answer:
  - When we don't know in advance how many elements are in our data set
  - When the number of elements varies significantly
- Is this the case here?
- No!

Huffman’s Algorithm

- So, our alternatives are? .....
- An array, indexed by number, of type ...
- *binary_tree*, i.e., each element in the array can point to a binary code tree

Huffman’s Algorithm



Huffman’s Algorithm

- What will be the dimension of this array?
- n, the number of symbols in our source alphabet since this is the number of trees we start out with in our forest initially

Huffman’s Algorithm

- Phase 2: construct the optimal code tree

Huffman's Algorithm

```
Pseudo-code algorithm

Find the tree with the smallest weight - A, at
   element i
Find the tree with the next smallest weight - B,
   at element j

Construct a tree, with right sub-tree A, left
   sub-tree B, with root having weight = sum of
   the roots of A and B

Let array element i point to the new tree
Delete tree at element j
```

Huffman's Algorithm

let n be the number of trees initially

Repeat

```
Find the tree with the smallest weight - A, at
  element i
Find the tree with the next smallest weight - B,
  at element j

Construct a tree, with right sub-tree A, left
  sub-tree B, with root having weight = sum of
  the roots of A and B

Let array element i point to the new tree
Delete tree at element j
```

Until only one tree left in the array

Huffman's Algorithm

- Phase 3: Compute the encoding map
  - We need to write out a list of source symbols together with their prefix code
  - We need to write out the contents of each external node (or each frontier internal node) together with the path to that node
  - We need to **traverse** the binary code tree in some manner

- But …. we want to print out the symbol and the prefix code:

  i.e. the symbol at the leafnode

  and the path by which we got to that node

- How will we represent the path?
- As an array of binary values (representing the left and right links on the path)

Huffman's Algorithm

```
// new tree definition

struct node
{
   char symbol;
   float probability;
   node *pleft, *pright;
};
```

Huffman's Algorithm

```
class tree // from previous part of the course
{
public:
   tree();
   ~tree();
   void add(int n) {addnode(n,root);}
   void print() {pr(root,0);}
   node* &search(int n);
   int delnode(int x);
private
   node *root;
   void deltree(node *p);
   void addnode(int n, node* &p);
   void pr(const node *p, int nspace) const;
};
```

Huffman's Algorithm

```
class tree // modified for this application
{
public:
   tree();
   ~tree();
   void add(char s, float p) {addnode(s,p,root);}
   void print() {pr(root,0);}
   node* &search(int n);
   int delnode(int x);
private
   node *root;
   void deltree(node* &p); // NB
   void addnode(char s, float p, node* &p);
   void pr(const node *p, int nspace) const;
};
```

Huffman's Algorithm

```
void tree::deltree(node* &p) {
// modified parameter to reference parameter
   if (p != NULL) {
      deltree(p->pleft);
      deltree(p->pright);
      delete p;
      p = NULL;  // return null pointer
   }
}
```

Huffman's Algorithm

```
class forest {
public:
  forest(int size);
  ~forest();
  void initialize_forest();
  void add_to_tree(int tree_number,
                   char symbol, float probability);
  void print_forest() const;
  void print_tree(int tree_number);
  void join_trees(int tree_1, int tree_2);
  int empty_tree(int tree_number);
  float root_probability(int tree_number);
private:
  tree tree_array[MAXIMUM_NUMBER_OF_TREES];
  int forest_size;
};
```

Huffman's Algorithm

```
// inorder traversal from previous part of the course
//
// recursive function to print the contents of the
// binary search tree

void tree::prorder(const node *p) const
{
   if (p!=NULL)
   {
      prorder(p->pleft);
      cout << p->data << " ";
      prorder(p->pright);
   }
}
```

Huffman's Algorithm

```
// inorder traversal to print only leaf nodes

void tree::leafnode_traversal(const node *p) const
{
   if (p != NULL) {
      if (at_leafnode) {    // PSEUDO CODE
         visit this node
      }
      else {
         leafnode_traversal(p->pleft);
         leafnode_traversal(p->pright);
      }
   }
}
```

Huffman's Algorithm

```
// inorder traversal to print only leaf nodes

void tree::leafnode_traversal(const node *p) const
{
   if (p != NULL) {
      if ((p->pleft == NULL) &&
          (p->right == NULL)) {     // leafnode
         cout << p->symbol << p->probability <<endl;
      }
      else {
         leafnode_traversal(p->pleft);
         leafnode_traversal(p->pright);
      }
   }
}
```

Huffman's Algorithm

```
// pseudocode version of compute_map
// to traverse tree and print leaf node and path
// to leaf node

void tree::traverse_leaf_nodes(const node *p, path)
{
   if (at leaf node) {
      print out symbol and path
   }
   else {
      add_to_path(path, 0);  // left
      traverse_leaf_nodes(p->pleft, path);
      remove_element_from_path(path);
```

Huffman’s Algorithm

```
add_to_path(path, 1);  // right
traverse_leaf_nodes(p->pright, path);
remove_element_from_path(path);
```

Huffman's Algorithm

```
// Definition of path

#define MAX_PATH_LENGTH 20
class path {
public:
   path();
   ~path();
   add_to_path(int direction);
   remove_from_path();
   print_path();
private:
   int path_components[MAX_PATH_LENGTH];
   int path_length;
}
```

Huffman’s Algorithm

```
// Definition of path

path::path()
{
   int i;
   for (i=0; i<MAX_PATH_LENGTH; i++) {
      path_components[i] =  0;
   }
   path_length = 0;
}
```

Huffman’s Algorithm

```
// Definition of path

path::~path()
{
}
```

Huffman's Algorithm

```
// Definition of path

path::add_to_path(int direction)
{
   if (path_length < MAX_PATH_LENGTH) {
      path_components[path_length] = direction;
      path_length++;
   }
   else {
      cout << "Error maximum path length reached";
   }
}
```

Huffman’s Algorithm

```
// Definition of path

path::remove_from_path()
{
   if (path_length > 0) {
      path_length--;
   }
   else {
      cout << “Error: no path exists”;
   }
}
```

Huffman’s Algorithm

```
// Definition of path

path::print_path()
{
   for (i=0; i<path_length; i++) {
      cout << path_components[i];
   }
   cout << “ “;
}
```

Huffman's Algorithm

```
// Definition of traverse_leaf_nodes
// to traverse tree and print leaf node and path
// to leaf node

void tree::traverse_leaf_nodes(const node *p, path &code)
{
  if (p != NULL) {
    if ( (p->pleft == NULL) &&
          (p->pright == NULL)) {  // leaf node
        cout << p->symbol << " ";
        code.print_path();
        cout << endl;
    }
    else {
```

Huffman's Algorithm

```
        code.add_to_path(0);  // left
        traverse_leaf_nodes(p->pleft, code);
          code.remove_from_path();

          code.add_to_path(1);  // right
          traverse_leaf_nodes(p->pright, code);
          code.remove_from_path();
      }
    }
}
```

Huffman's Algorithm

```
void tree::compute_map() {
                    // new function to print leaf nodes
  path code;        // and the path to leaf nodes
  traverse_leaf_nodes(root, code);
}
```

Huffman’s Algorithm

```
void forest::compute_map() {
   int i;

   for (i=0; i<MAXIMUM_NUMBER_OF_TREES; i++) {
      if (tree_array[i].empty_tree() == FALSE) {
            tree_array[i].compute_map();
      }
   }
}
```

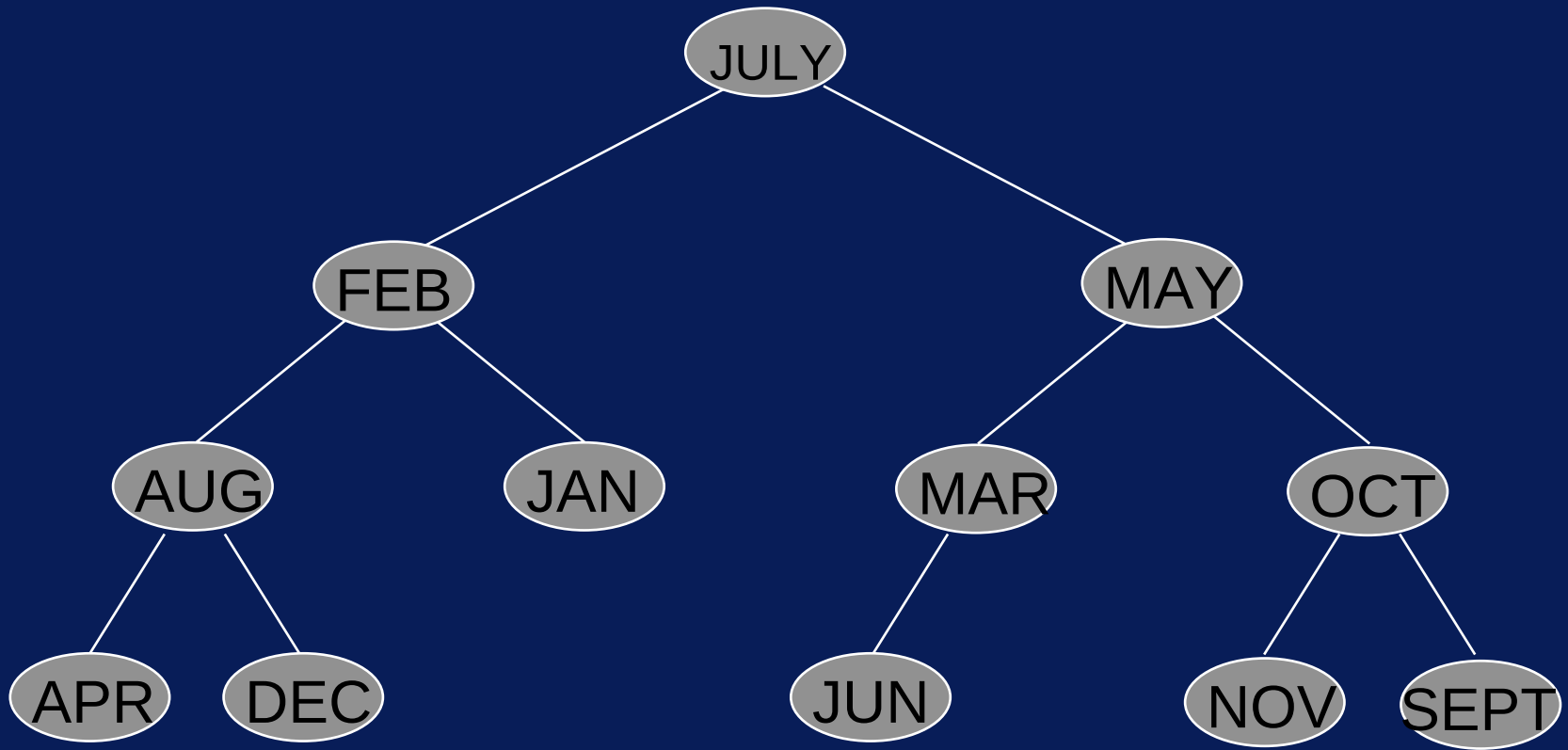
Height-Balanced Trees

AVL Trees

AVL Trees

- We know from our study of Binary Search Trees (BST) that the average search and insertion time is O(log n)
  - If there are n nodes in the binary tree it will take, on average, log2 n comparisons/probes to find a particular node (or find out that it isn't there)
- However, this is only true if the tree is 'balanced'
  - Such as occurs when the elements are inserted in random order

AVL Trees



A Balanced Tree for the Months of the Year

AVL Trees

- However, if the elements are inserted in lexicographic order (i.e. in sorted order) then the tree degenerates into a skinny tree

AVL Trees

A Degenerate Tree for the Months of the Year

AVL Trees

- If we are dealing with a dynamic tree
- Nodes are being inserted and deleted over time
  - For example, directory of files
  - For example, index of university students
- we may need to restructure - balance - the tree so that we keep it
  - Fat
  - Full
  - Complete

AVL Trees

- Adelson-Velskii and Landis in 1962 introduced a binary tree structure that is balanced with respect to the heights of its subtrees
- Insertions (and deletions) are made such that the tree
  - starts off
  - and remains
- Height-Balanced

AVL Trees

- Definition of AVL Tree
- An empty tree is height-balanced
- If T is a non-empty binary tree with left and right sub-trees T_1 and T_2, then
- T is height-balanced iff
  - T_1 and T_2 are height-balanced, and
  - $|height(T_1) - height(T_2)| \leq 1$

AVL Trees

- So, every sub-tree in a height-balanced tree is also height-balanced

Recall: Binary Tree Terminology

- The height of T is defined recursively as

  0 if T is empty and

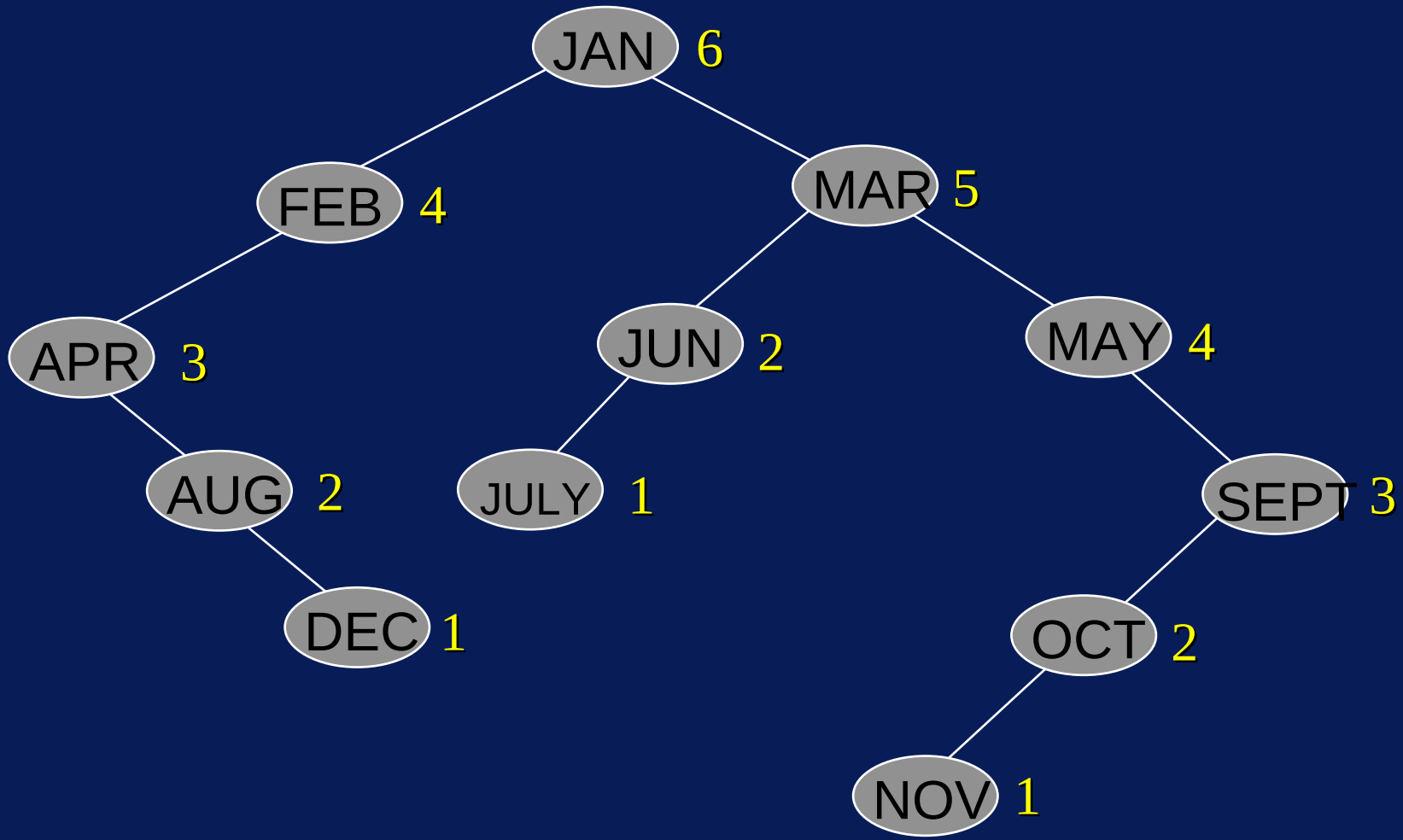
  1 + $max(height(T_1), height(T_2))$ otherwise, where T_1 and T_2 are the subtrees of the root.

- The height of a tree is the length of a longest chain of descendents

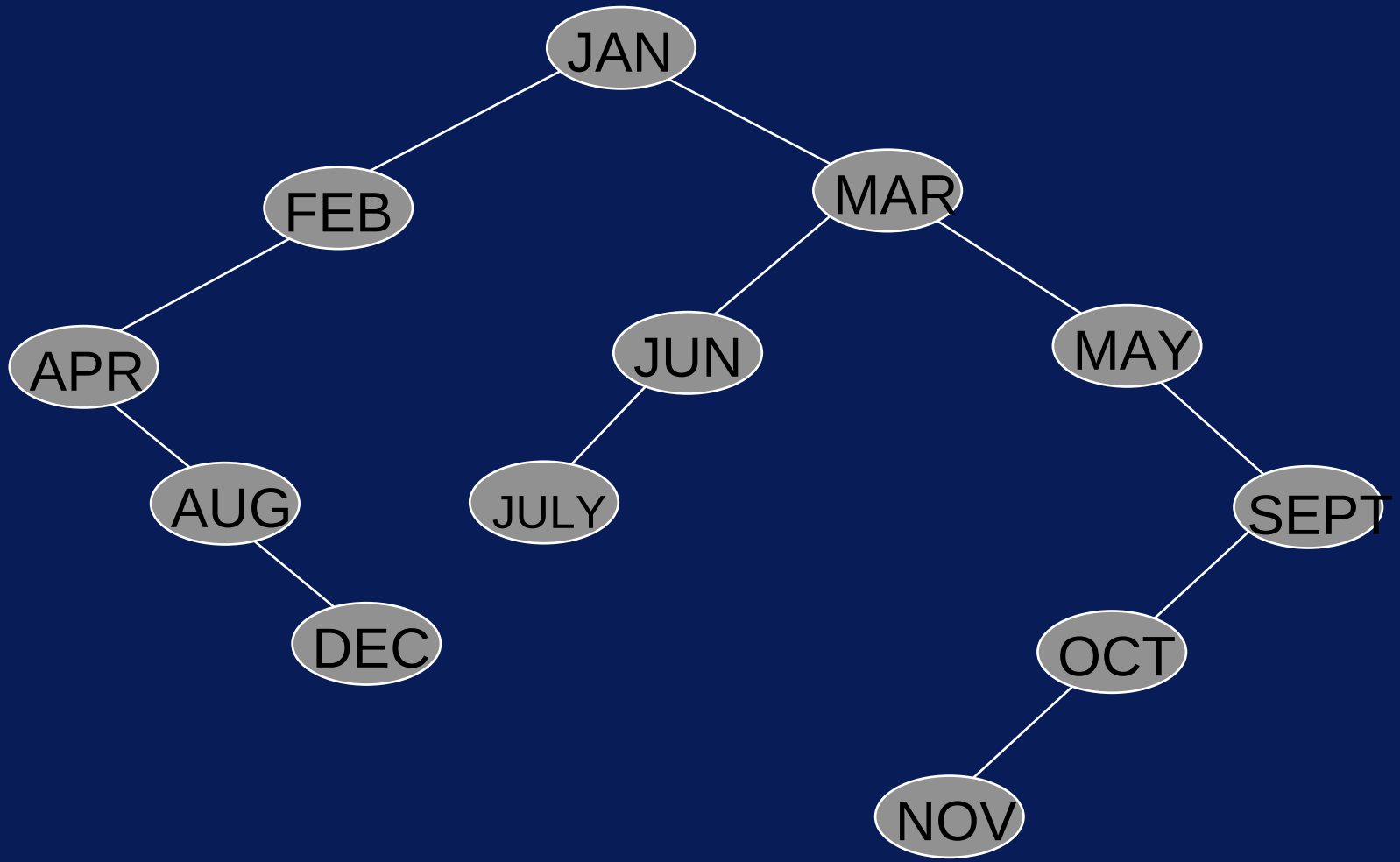
Recall: Binary Tree Terminology

- Height Numbering
  - Number all external nodes 0
  - Number each internal node to be one more than the maximum of the numbers of its children
  - Then the number of the root is the height of *T*
- The height of a node *u* in *T* is the height of the subtree rooted at *u*

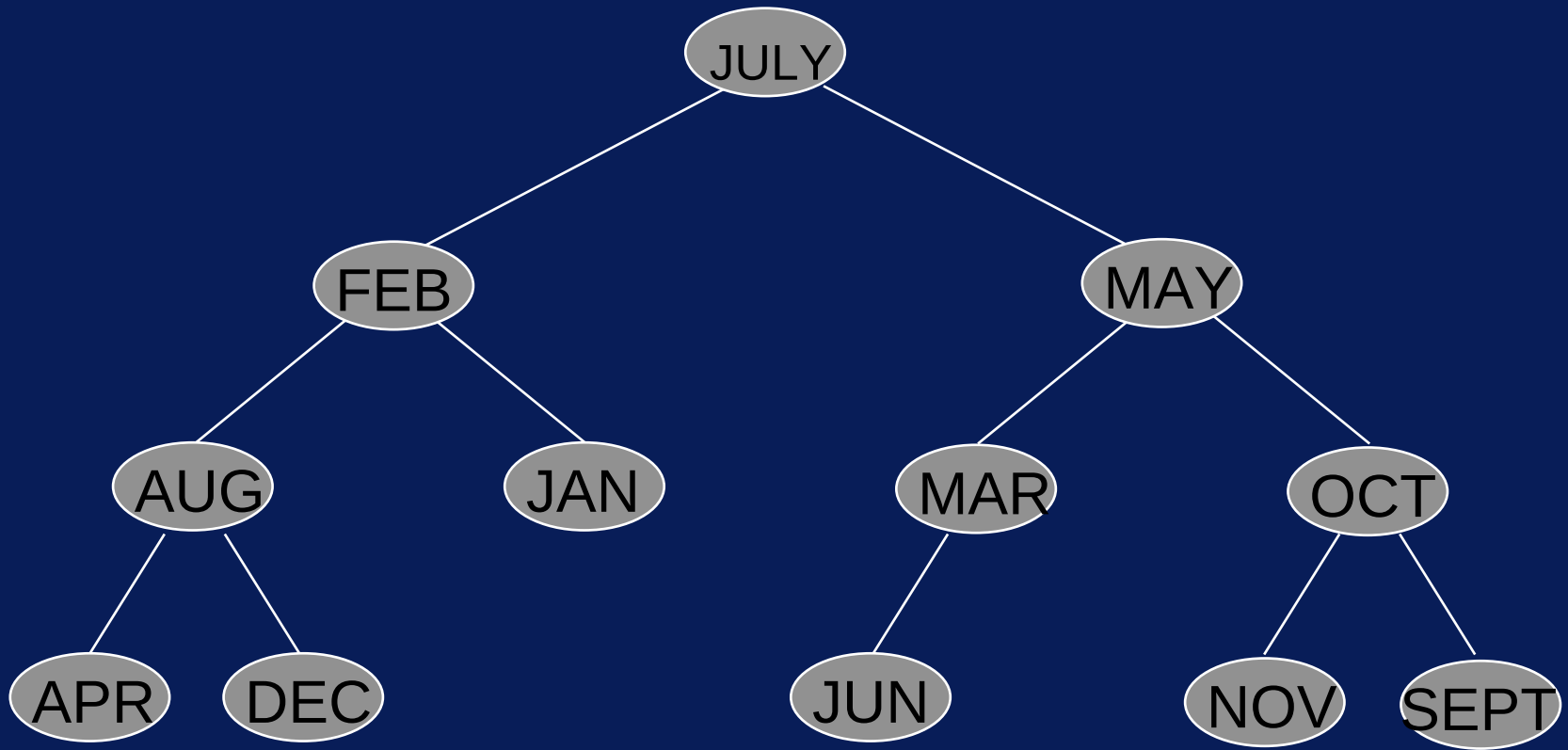
AVL Trees



AVL Trees

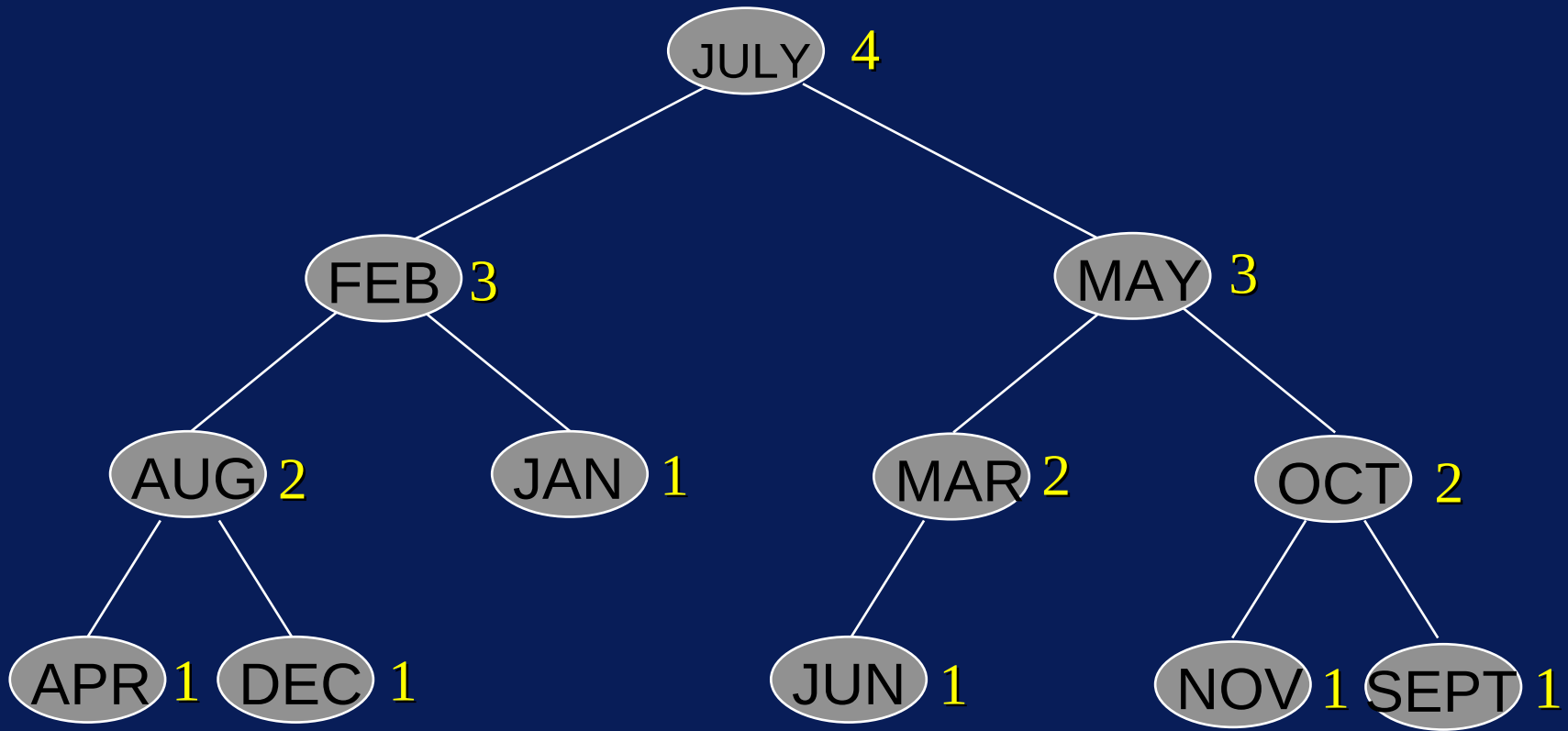


AVL Trees



A Balanced Tree for the Months of the Year

AVL Trees



A Balanced Tree for the Months of the Year

AVL Trees

- Let’s construct a height-balanced tree
- Order of insertions:

  March, May, November, August, April, January, December, July, February, June, October, September

- Before we do, we need a definition of a balance factor

AVL Trees

- Balance Factor *BF*(*T*) of a note T in a binary tree is defined to be

  $height(T_1) - height(T_2)$

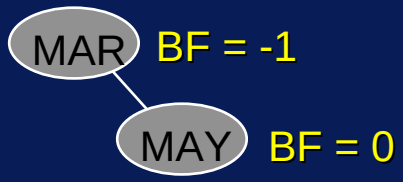
  where T_1 and T_2 are the left and right subtrees of *T*

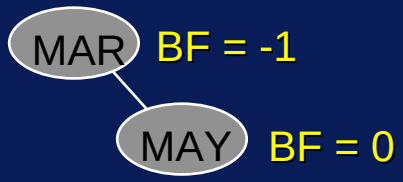
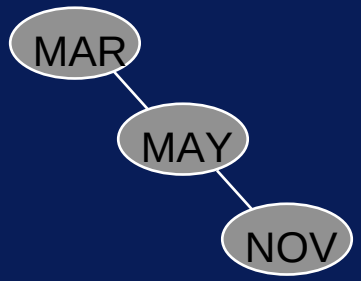
- For any node T in an AVL tree *BF*(*T*) = -1, 0, +1

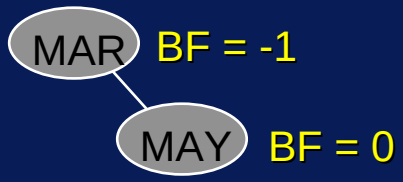
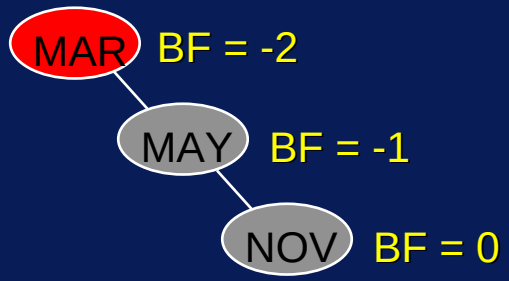
New Identifier	After Insertion	After Rebalancing
MARCH	MAR	

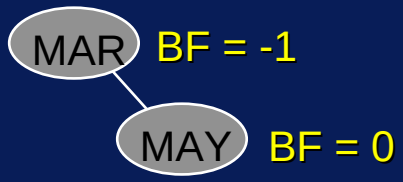
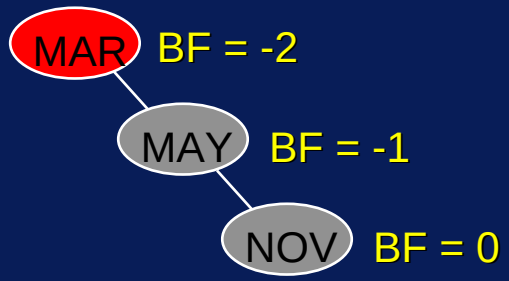
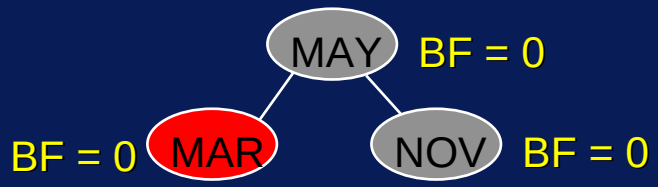
New Identifier	After Insertion	After Rebalancing
MARCH	MAR BF = 0	NO REBALANCING NEEDED

New Identifier	After Insertion	After Rebalancing
MARCH		NO REBALANCING NEEDED
MAY		

New Identifier	After Insertion	After Rebalancing
MARCH		NO REBALANCING NEEDED
MAY		NO REBALANCING NEEDED

New Identifier	After Insertion	After Rebalancing
MARCH		NO REBALANCING NEEDED
MAY		NO REBALANCING NEEDED
NOVEMBER		

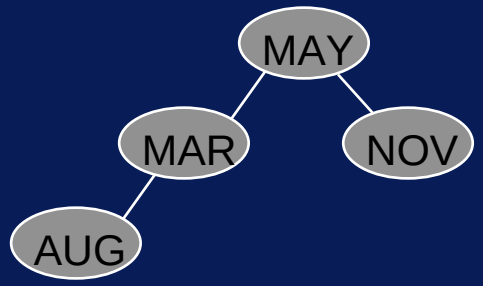
New Identifier	After Insertion	After Rebalancing
MARCH		NO REBALANCING NEEDED
MAY		NO REBALANCING NEEDED
NOVEMBER		

New Identifier	After Insertion	After Rebalancing
MARCH		NO REBALANCING NEEDED
MAY		NO REBALANCING NEEDED
NOVEMBER		 RR rebalancing

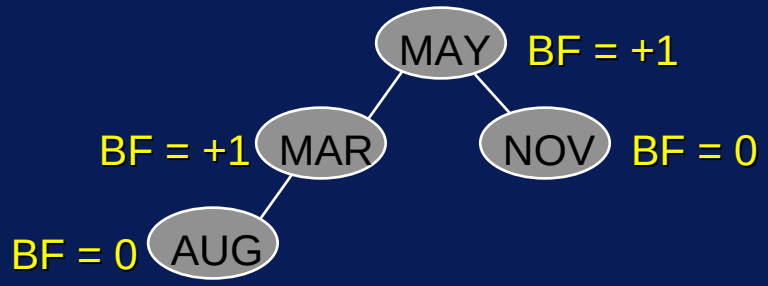
New Identifier

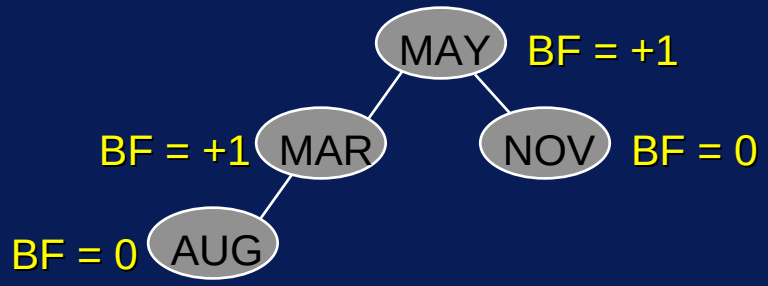
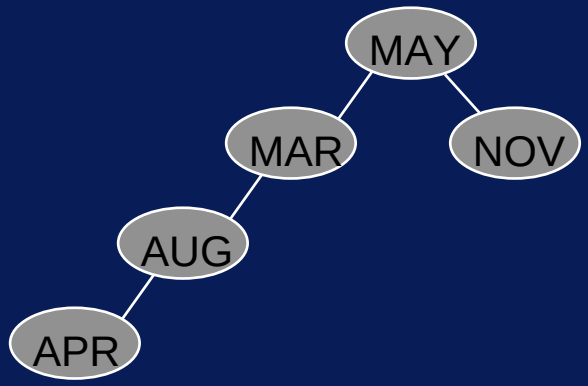
AUGUST

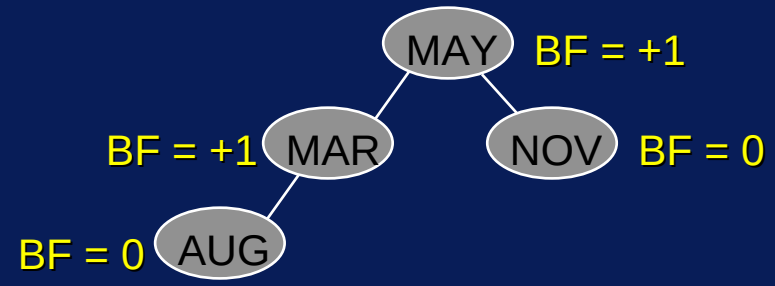
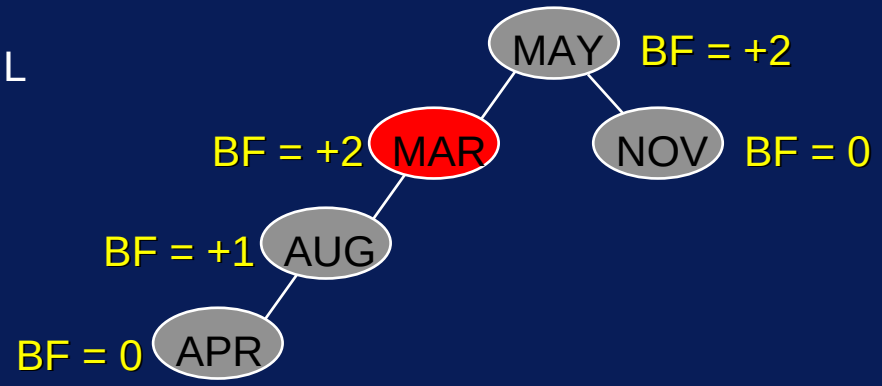
After Insertion

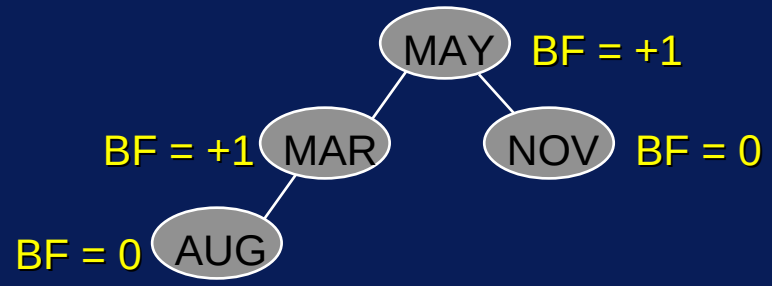
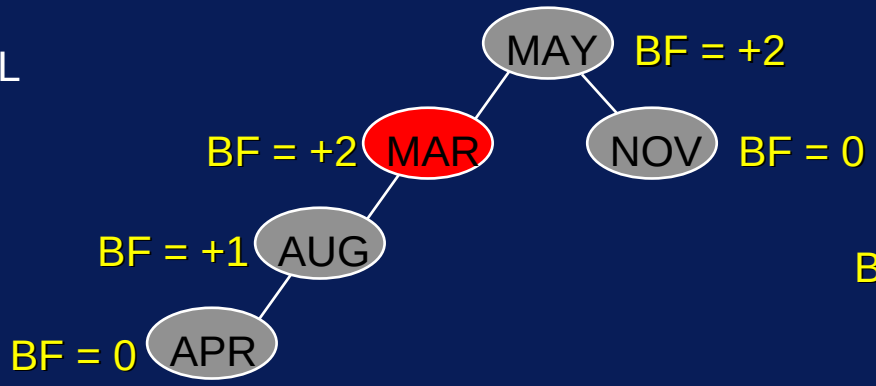
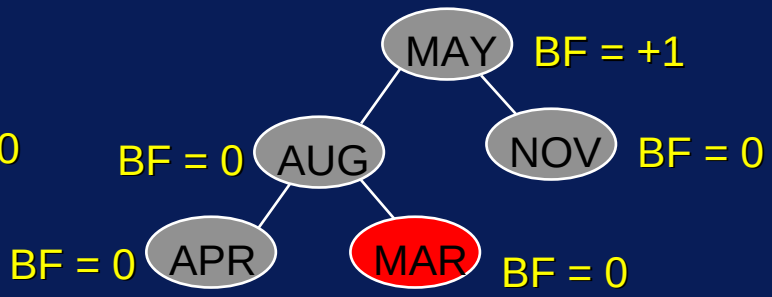


After Rebalancing

New Identifier	After Insertion	After Rebalancing
AUGUST		NO REBALANCING NEEDED

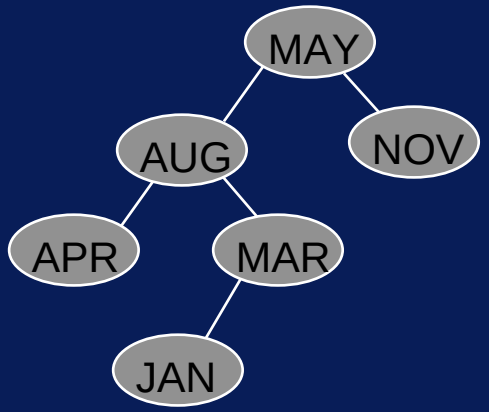
New Identifier	After Insertion	After Rebalancing
AUGUST		NO REBALANCING NEEDED
APRIL		

New Identifier	After Insertion	After Rebalancing
AUGUST		NO REBALANCING NEEDED
APRIL		

New Identifier	After Insertion	After Rebalancing
AUGUST		NO REBALANCING NEEDED
APRIL		 LL rebalancing

New Identifier | After Insertion | After Rebalancing

JANUARY

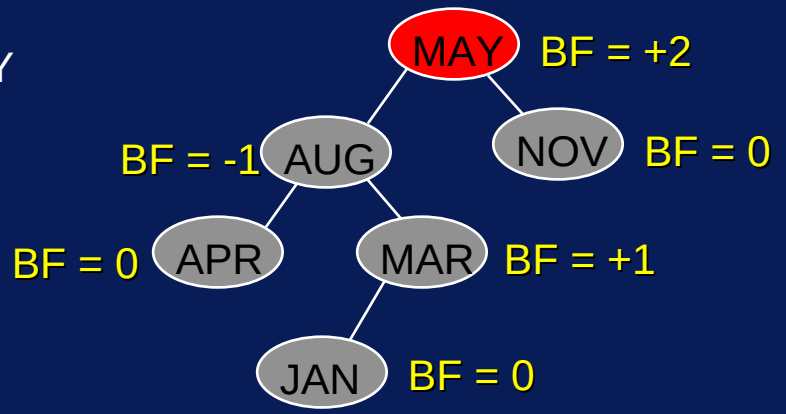


New Identifier

After Insertion

After Rebalancing

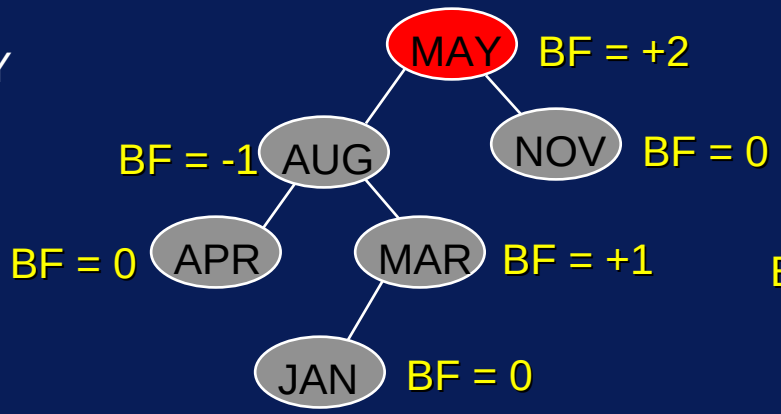
JANUARY



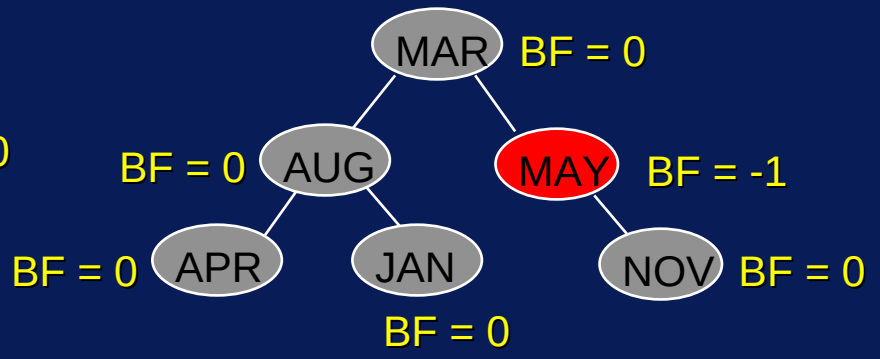
New Identifier

JANUARY

After Insertion



After Rebalancing



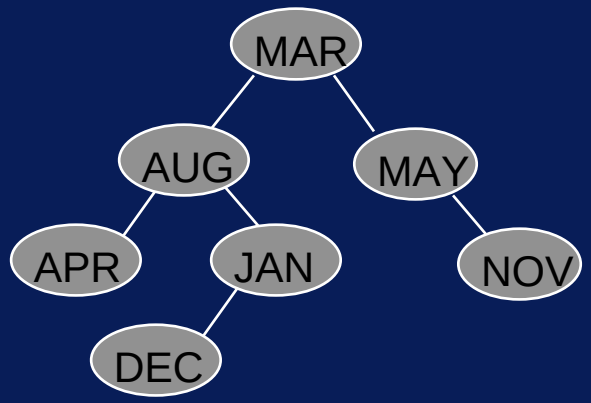
LR rebalancing

New Identifier

After Insertion

After Rebalancing

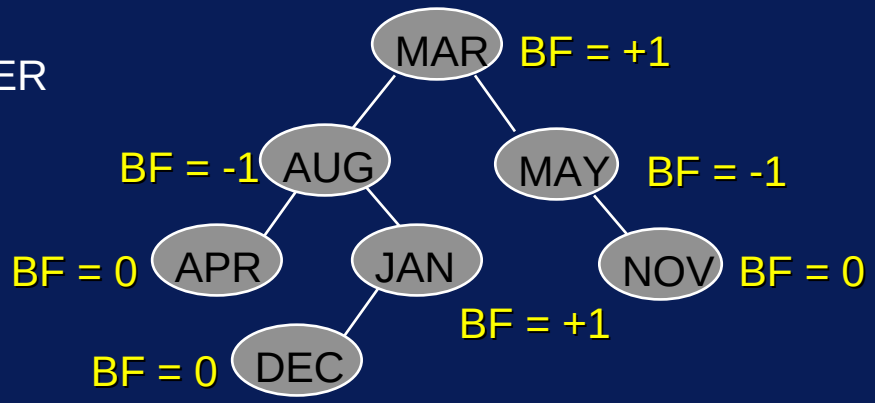
DECEMBER



New Identifier

DECEMBER

After Insertion



After Rebalancing

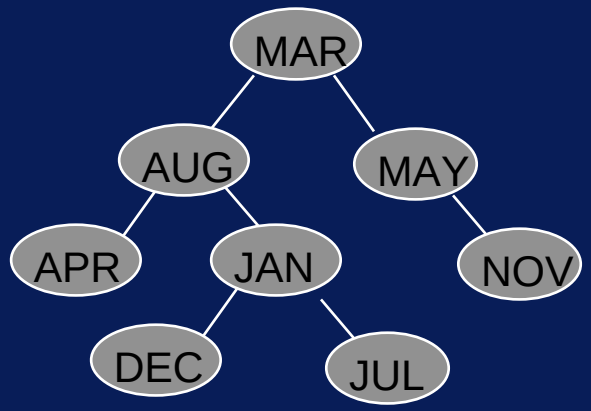
NO REBALANCING NEEDED

New Identifier

After Insertion

After Rebalancing

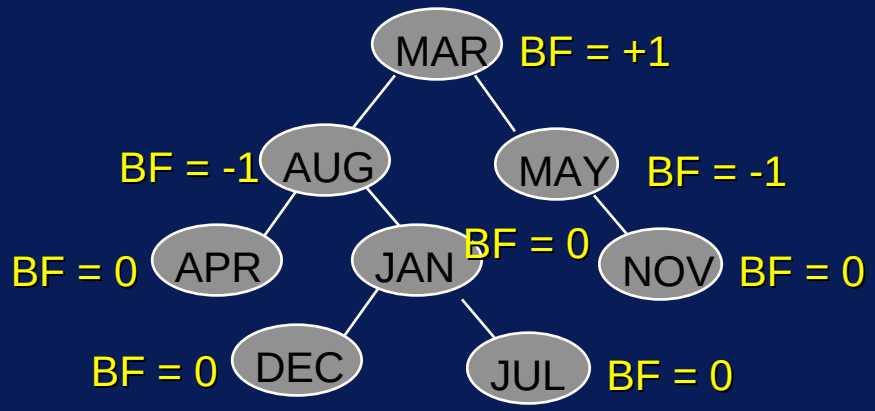
JULY



New Identifier

JULY

After Insertion

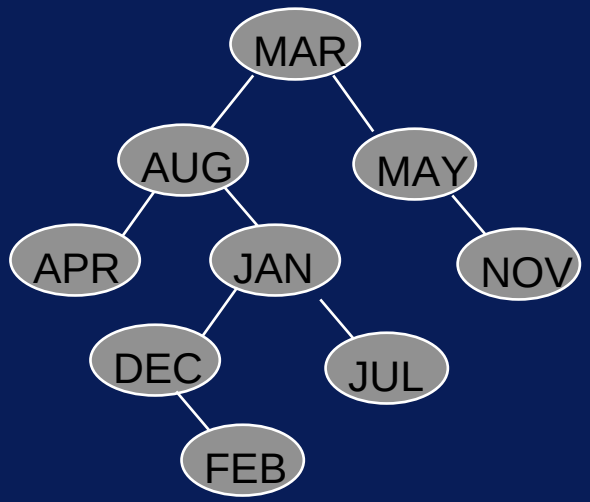


After Rebalancing

NO REBALANCING NEEDED

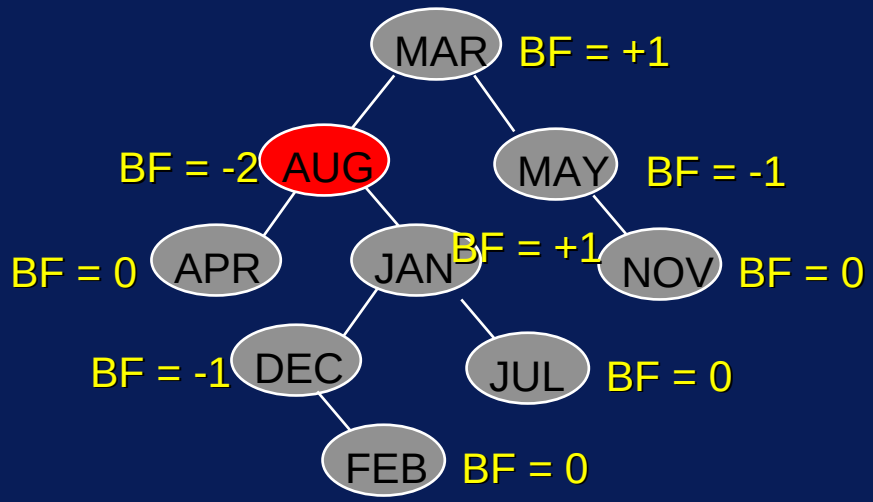
New Identifier | After Insertion | After Rebalancing

FEBRUARY



New Identifier | After Insertion | After Rebalancing

FEBRUARY

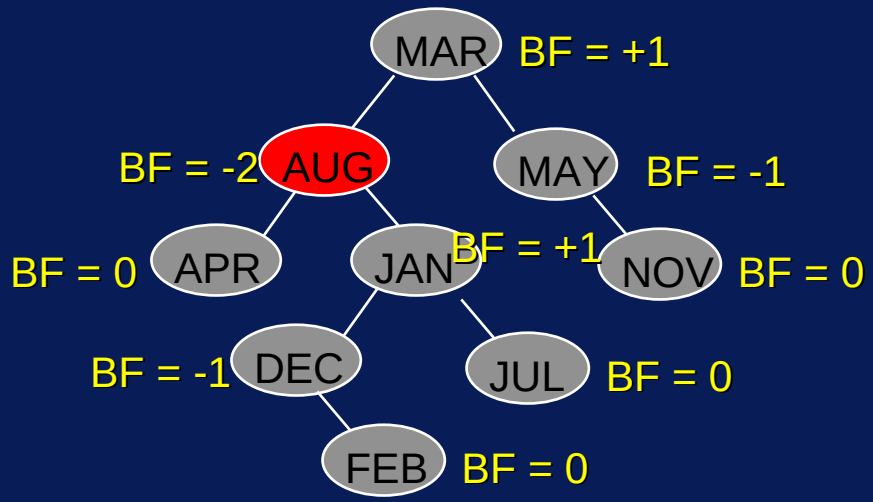


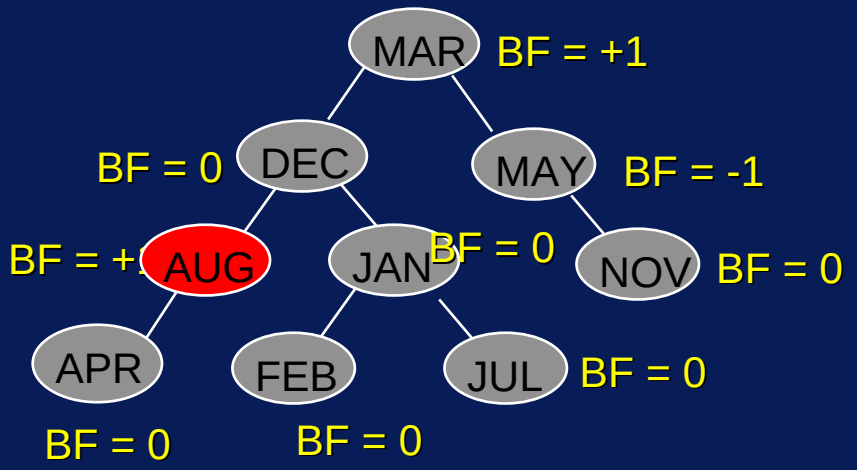
New Identifier

After Insertion

After Rebalancing

FEBRUARY

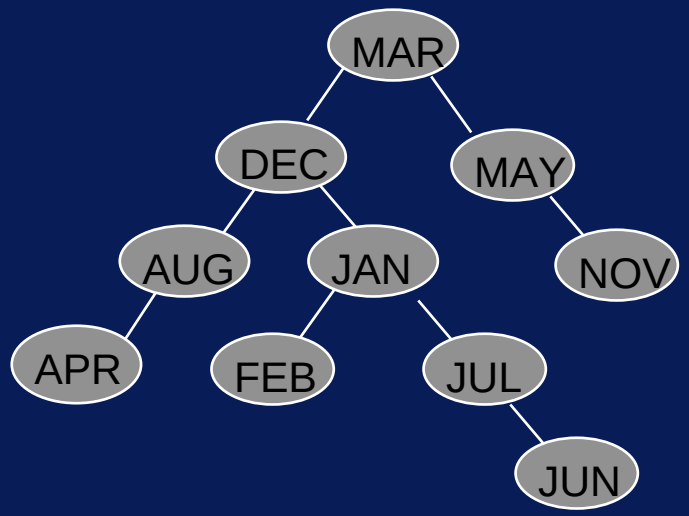




RL rebalancing

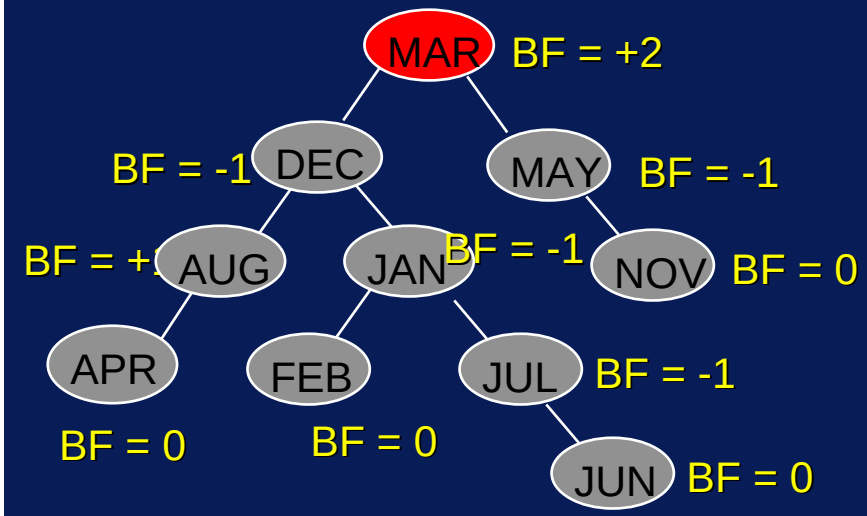
New Identifier | After Insertion | After Rebalancing

JUNE



New Identifier | After Insertion | After Rebalancing

JUNE

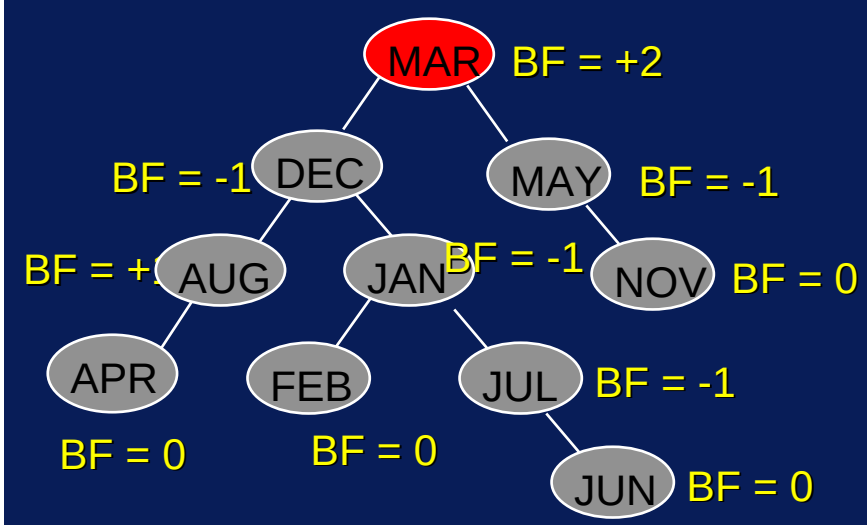


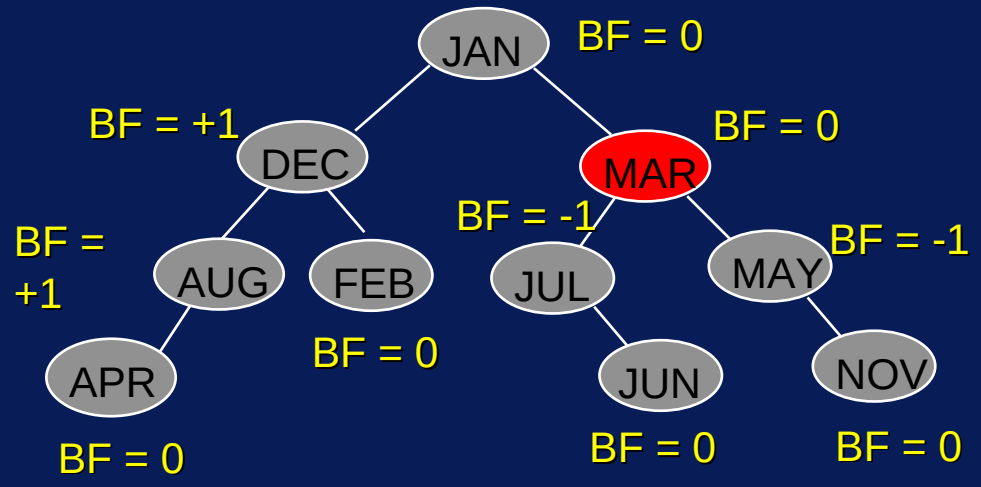
New Identifier

After Insertion

After Rebalancing

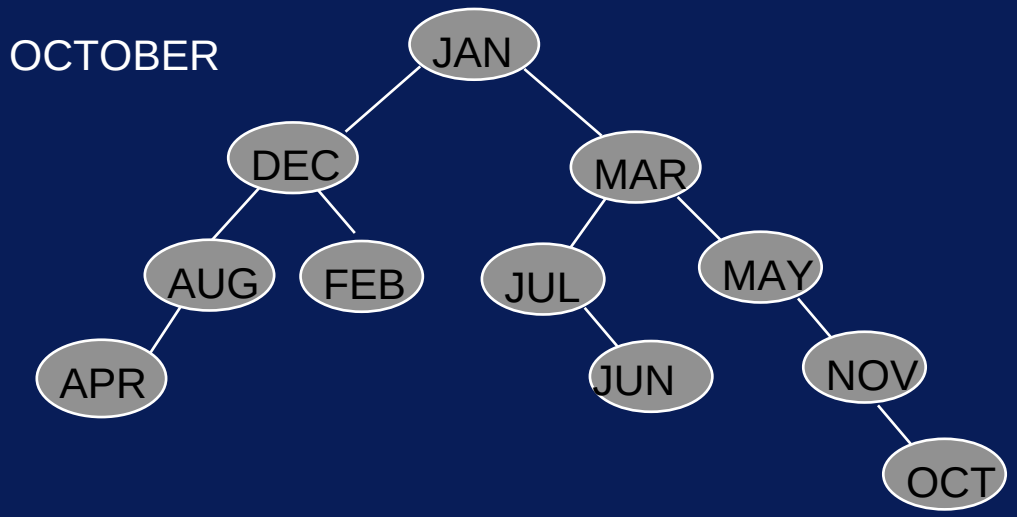
JUNE



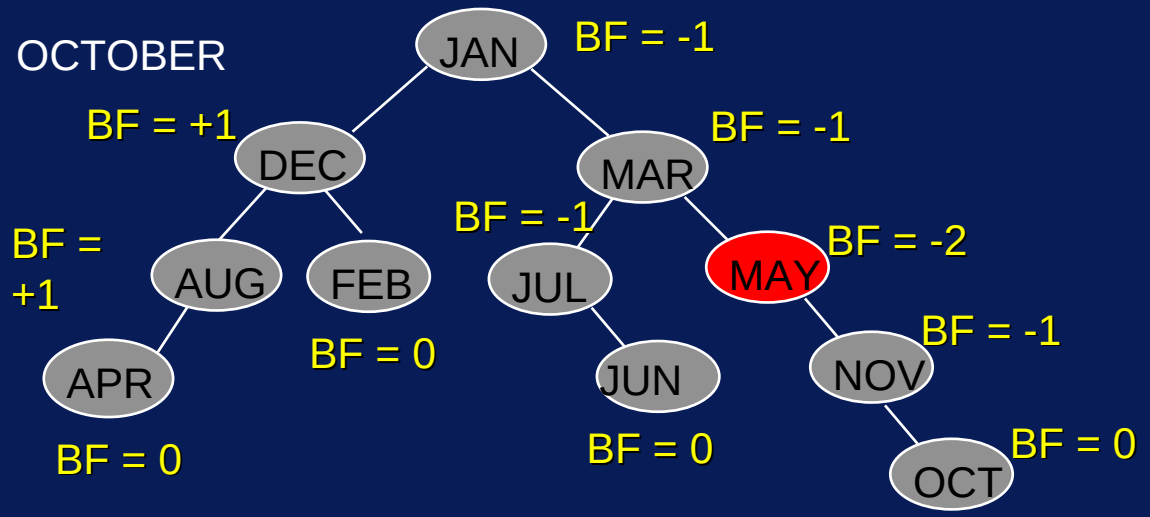


LR rebalancing

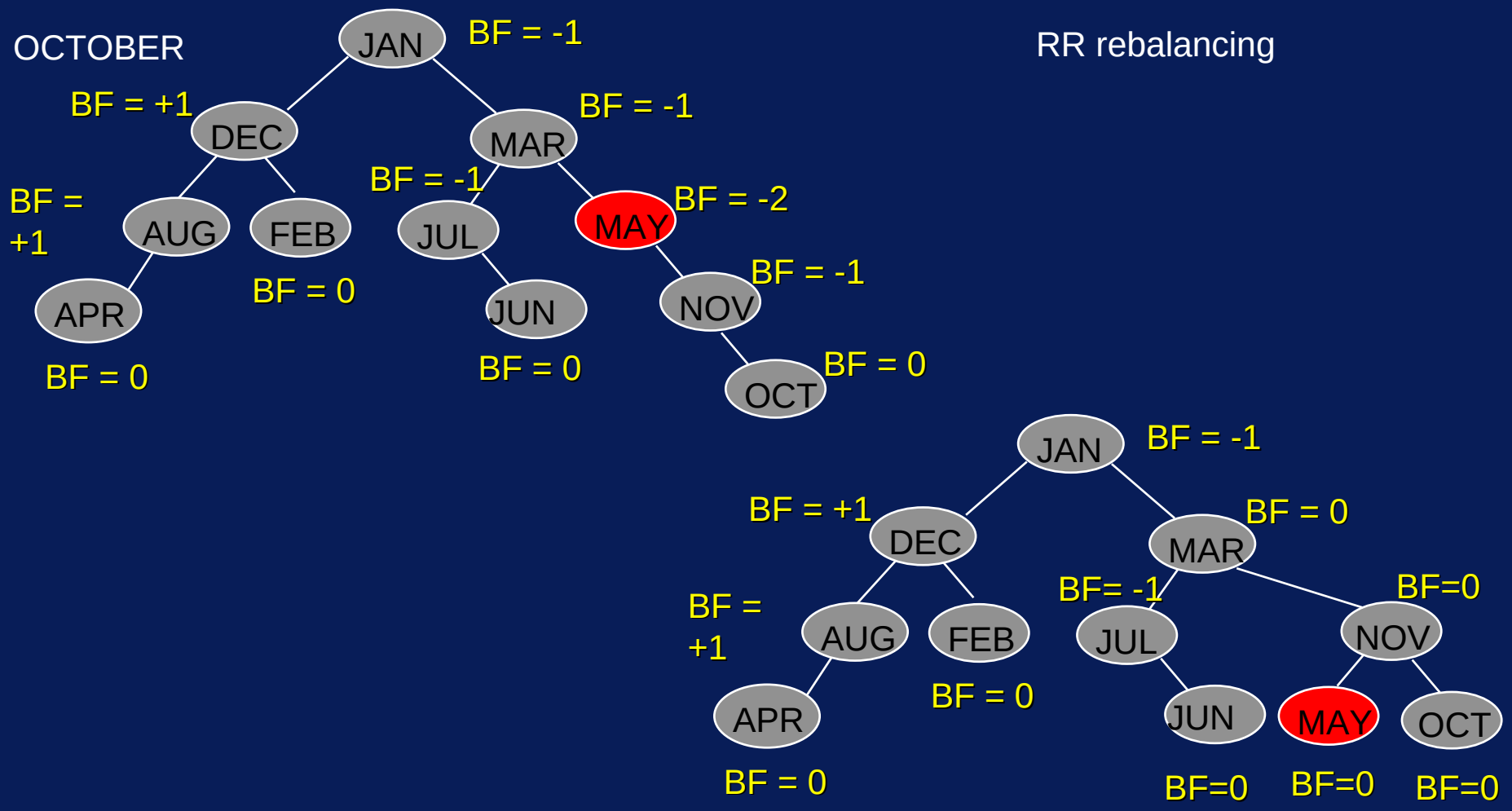
New Identifier | After Insertion | After Rebalancing



New Identifier | After Insertion | After Rebalancing

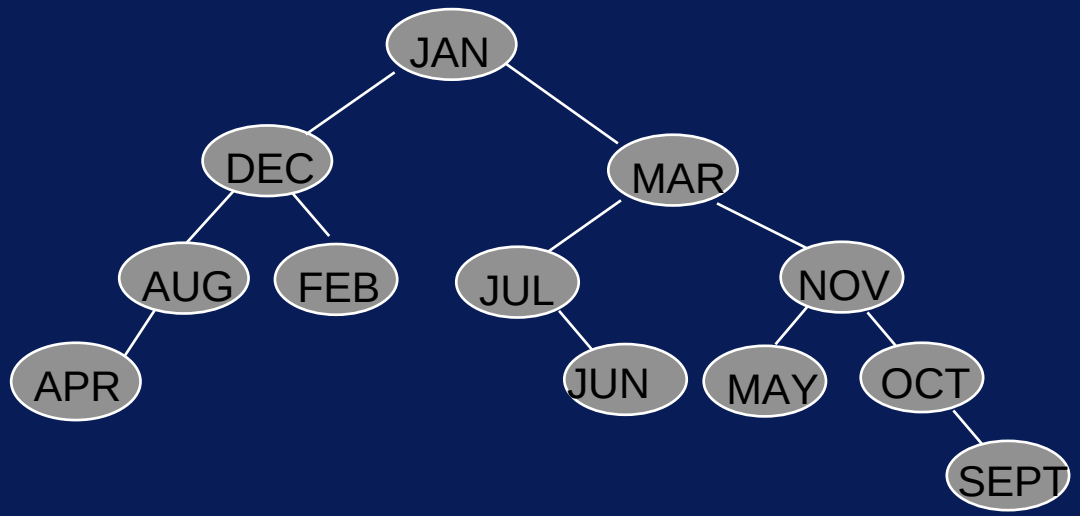


New Identifier | After Insertion | After Rebalancing



New Identifier | After Insertion | After Rebalancing

SEPTEMBER



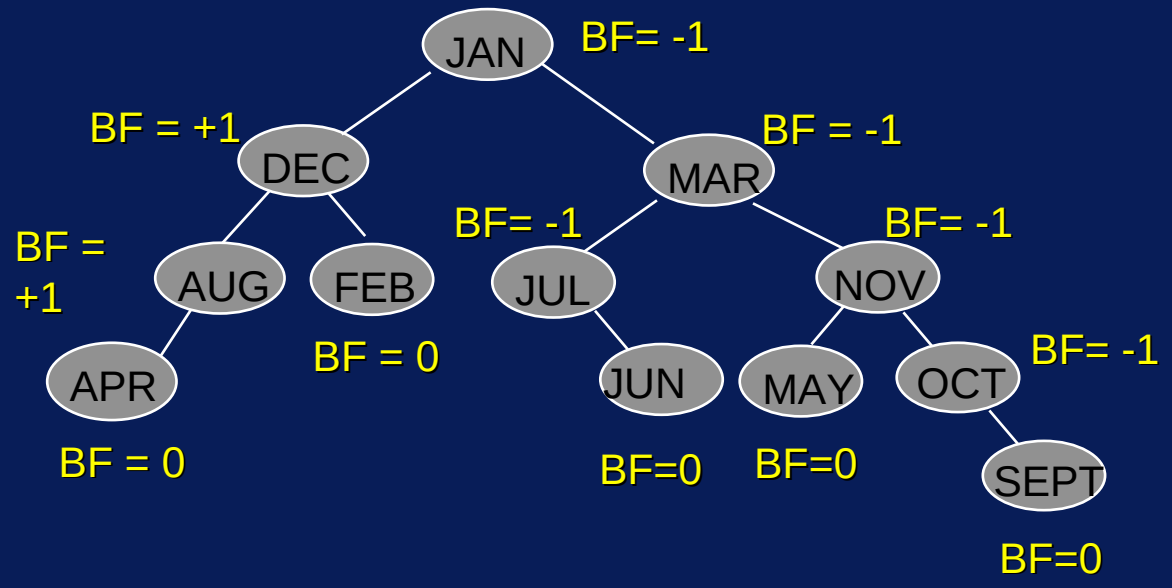
New Identifier

After Insertion

After Rebalancing

SEPTEMBER

NO REBALANCING NEEDED



AVL Trees

- All re-balancing operations are carried out with respect to the closest ancestor of the new node having balance factor +2 or -2
- There are 4 types of re-balancing operations (called rotations)
  - RR
  - LL (symmetric with RR)
  - RL
  - LR (symmetric with RL)

AVL Trees

- Let’s refer to the node inserted as Y
- Let’s refer to the nearest ancestor having balance factor +2 or -2 as A

AVL Trees

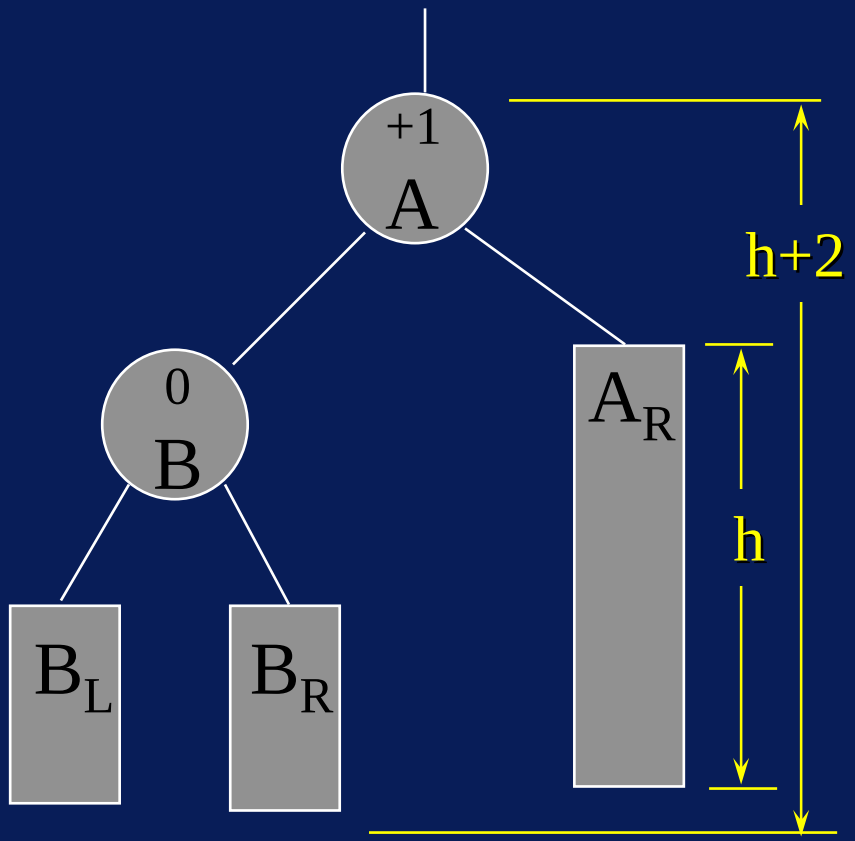
- LL: Y is inserted in the Left subtree of the Left subtree of A
  - LL: the path from A to Y
  - Left subtree then Left subtree
- LR: Y is inserted in the Right subtree of the Left subtree of A
  - LR: the path from A to Y
  - Left subtree then Right subtree

AVL Trees

- RR: Y is inserted in the Right subtree of the Right subtree of A
  - RR: the path from A to Y
  - Right subtree then Right subtree
- RL: Y is inserted in the Left subtree of the Right subtree of A
  - LL: the path from A to Y
  - Right subtree then Left subtree

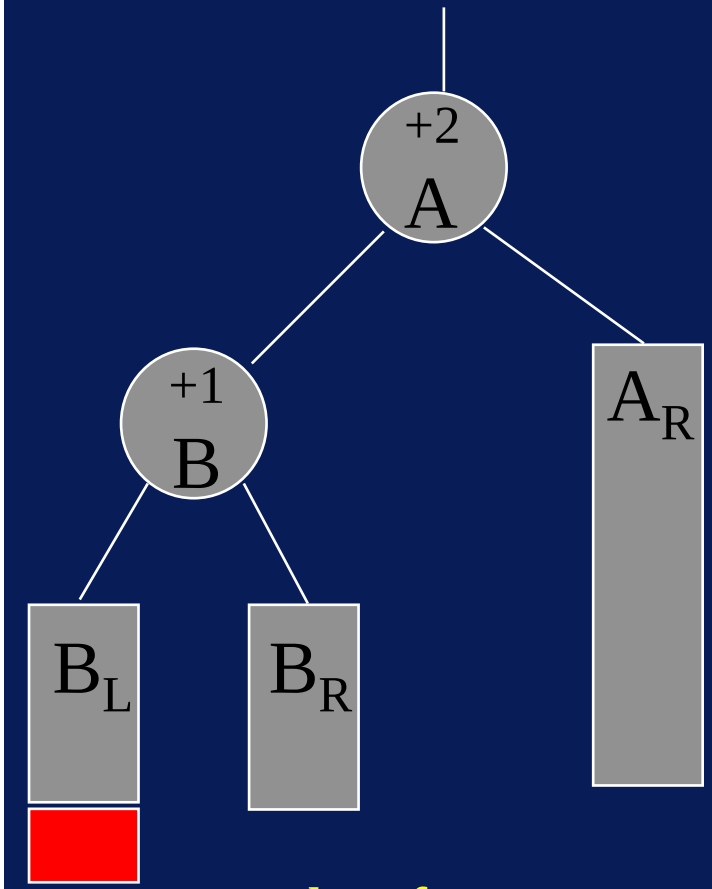
AVL Trees

Balanced Subtree



AVL Trees

Unbalanced following insertion

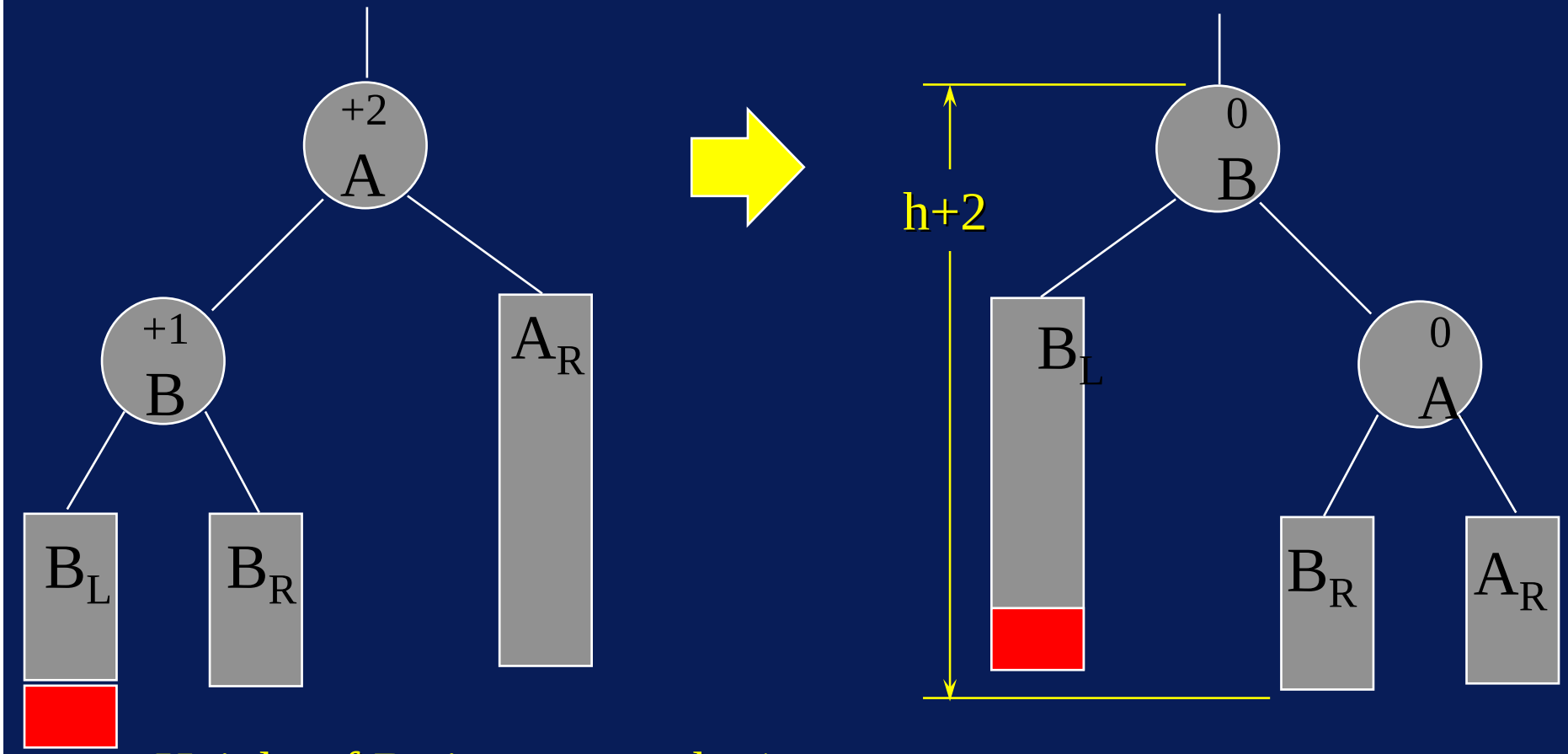


Height of B_L inceases to h+1

AVL Trees - LL rotation

Unbalanced following insertion

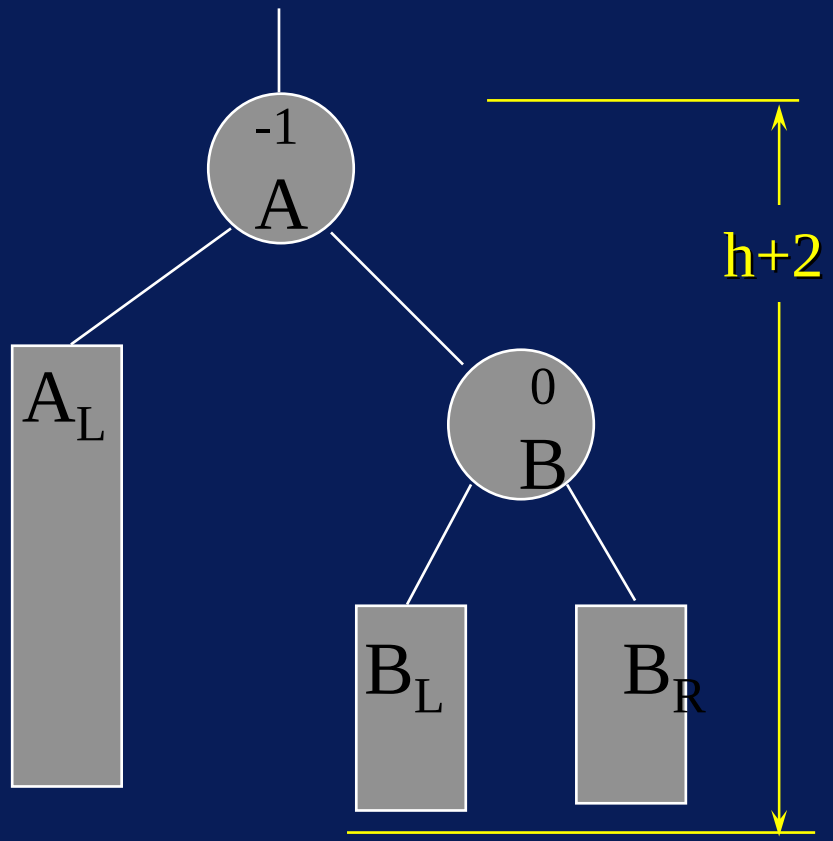
Rebalanced subtree



Height of B_L inceases to h+1

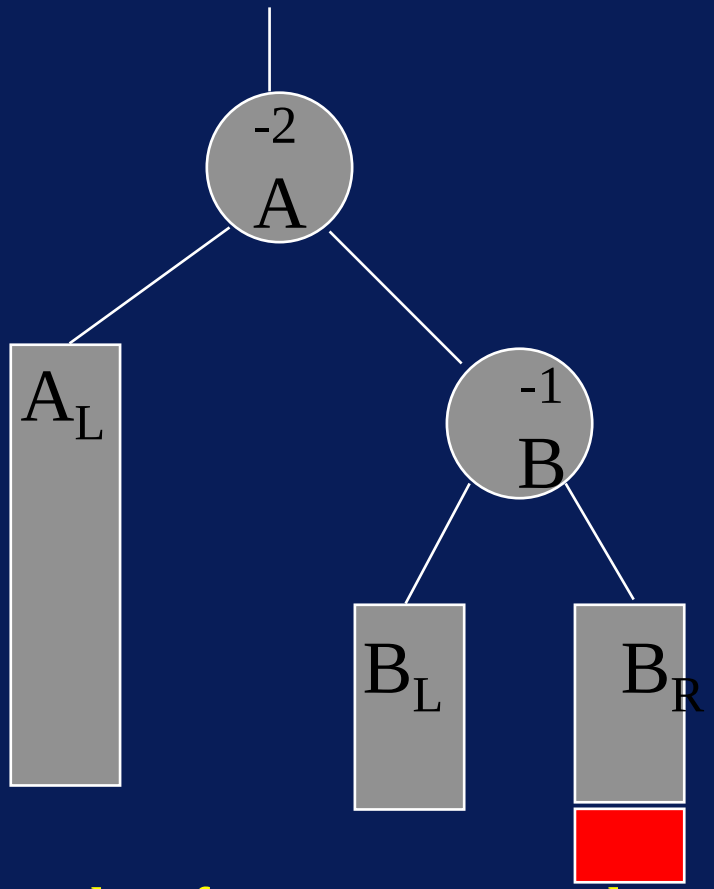
AVL Trees

Balanced Subtree



AVL Trees

Unbalanced following insertion

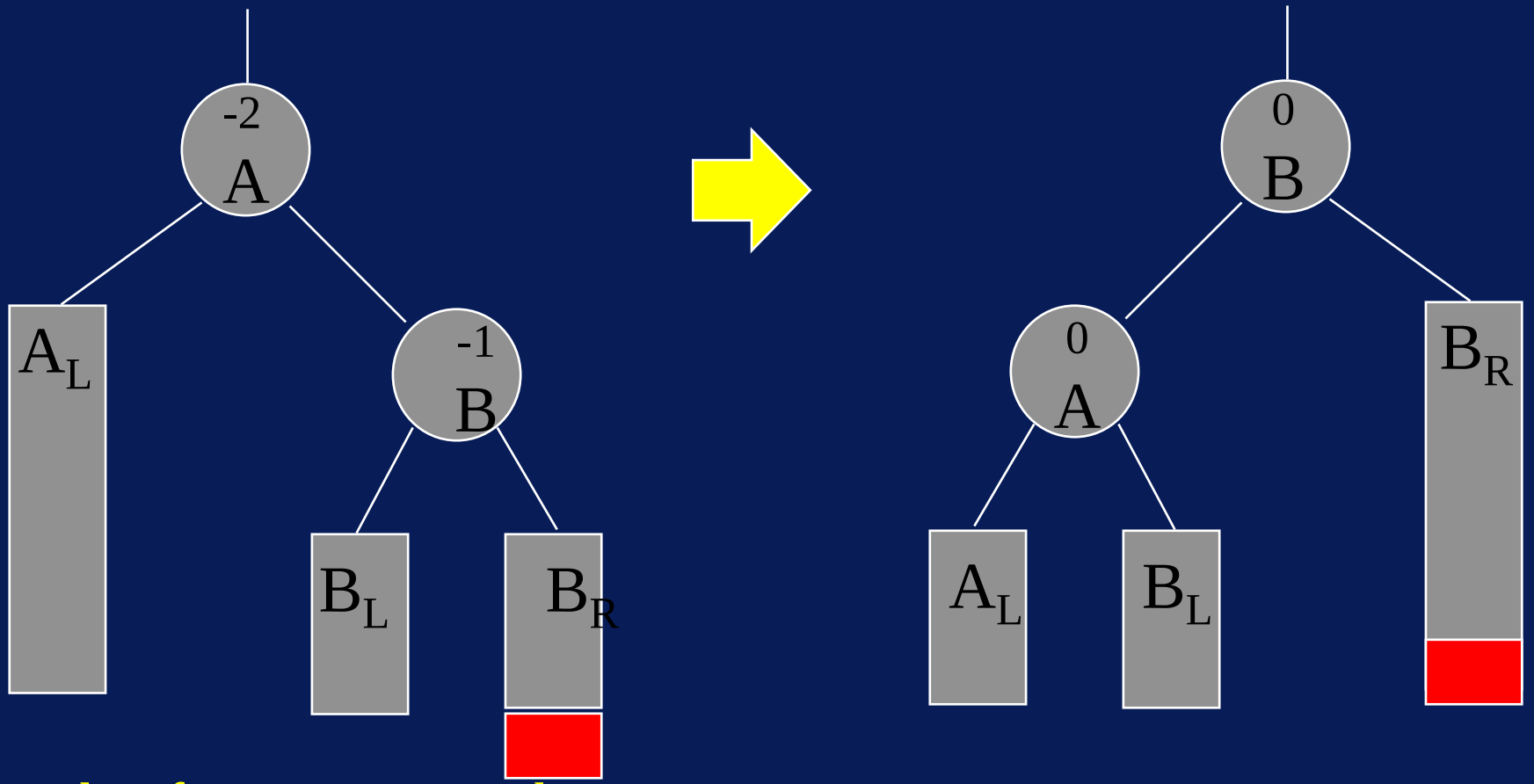


Height of B_R inceases to h+1

AVL Trees - RR Rotation

Unbalanced following insertion

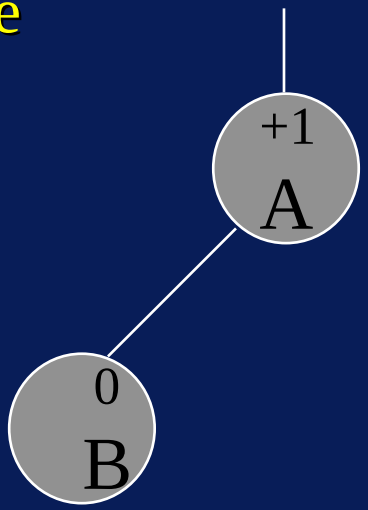
Rebalanced subtree



Height of B_R inceases to h+1

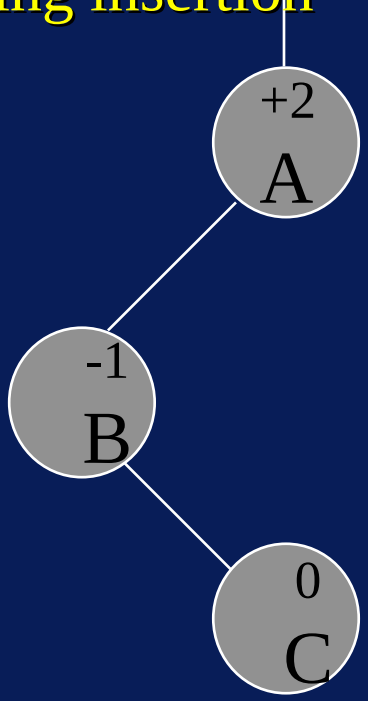
AVL Trees

Balanced Subtree

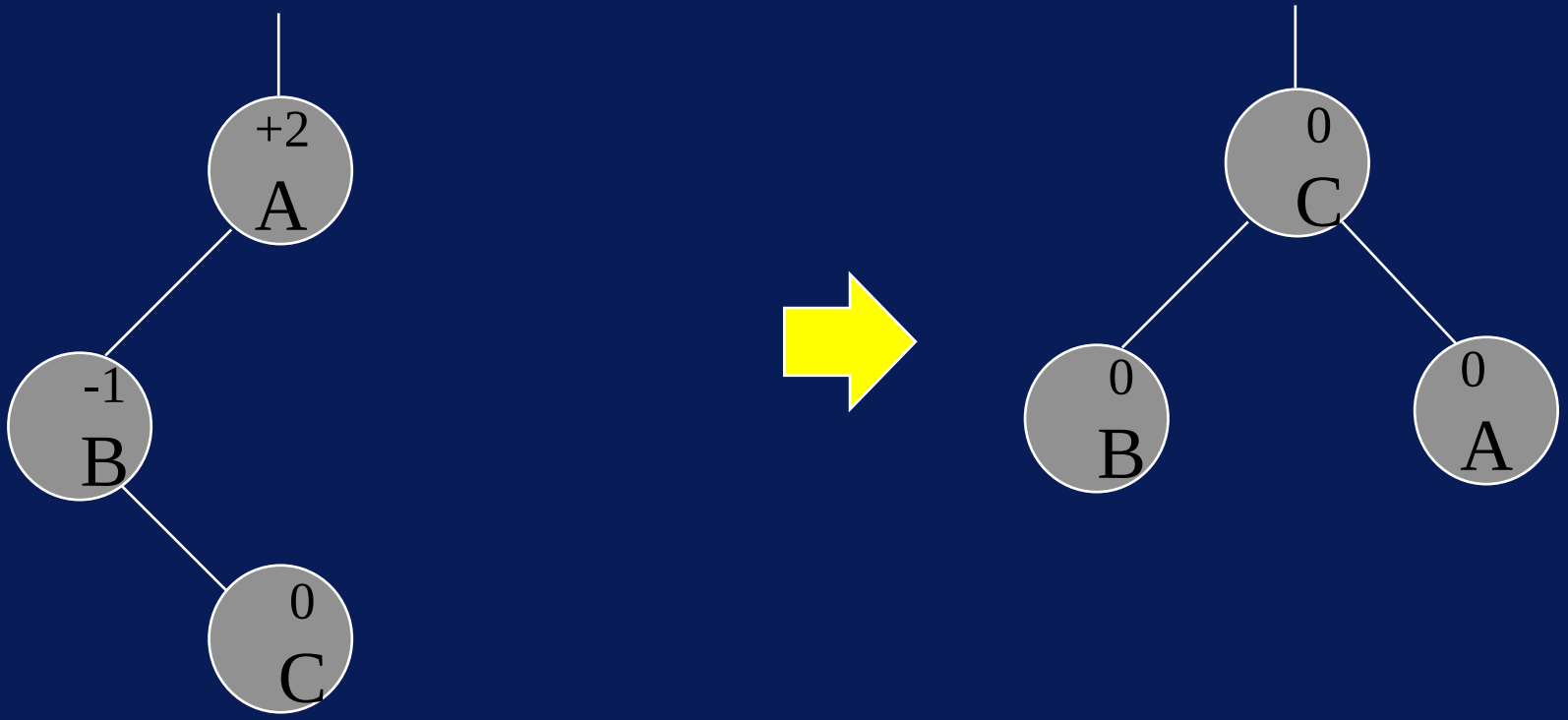


AVL Trees

Unbalanced following insertion

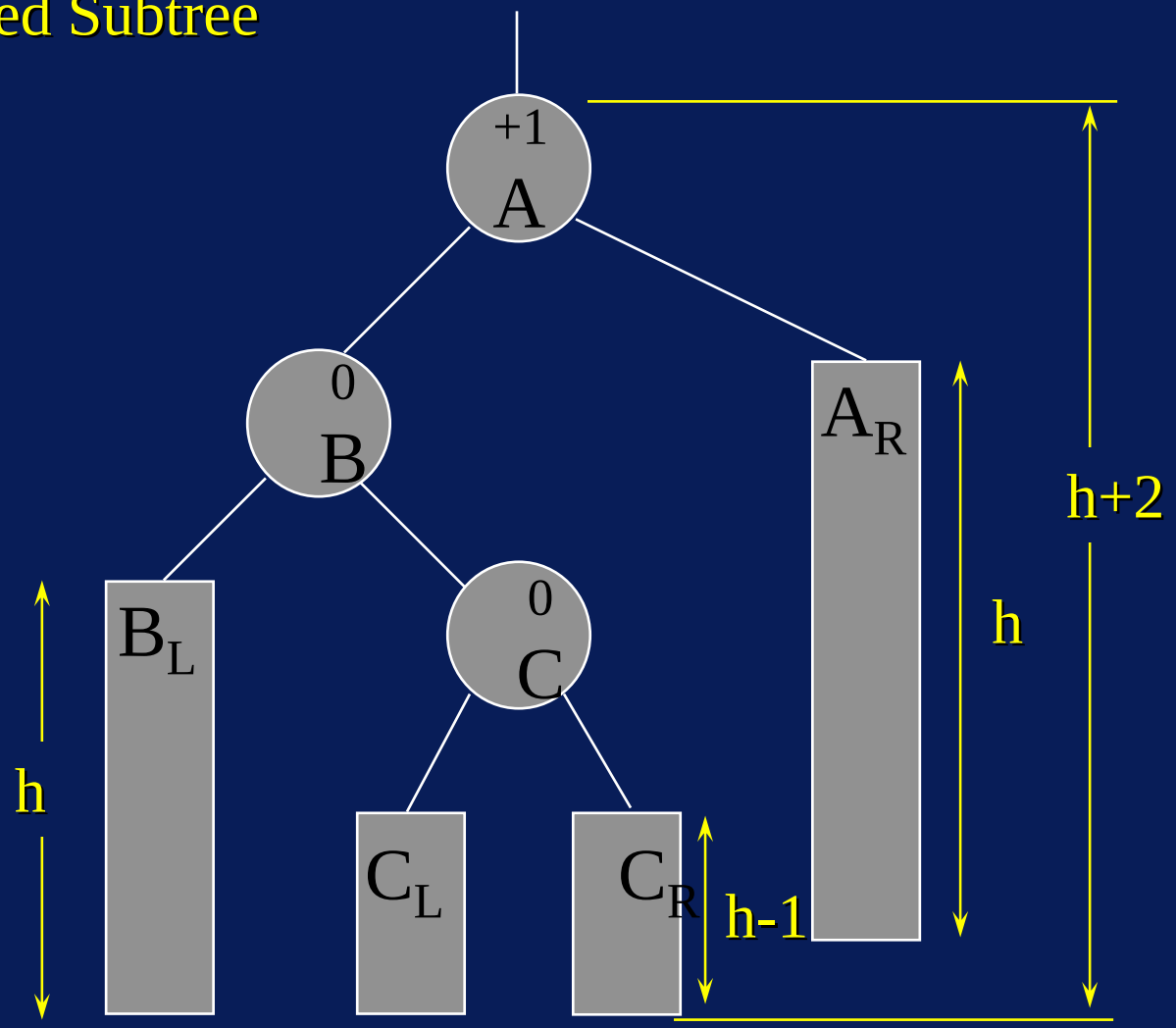


AVL Trees - LR rotation (a)



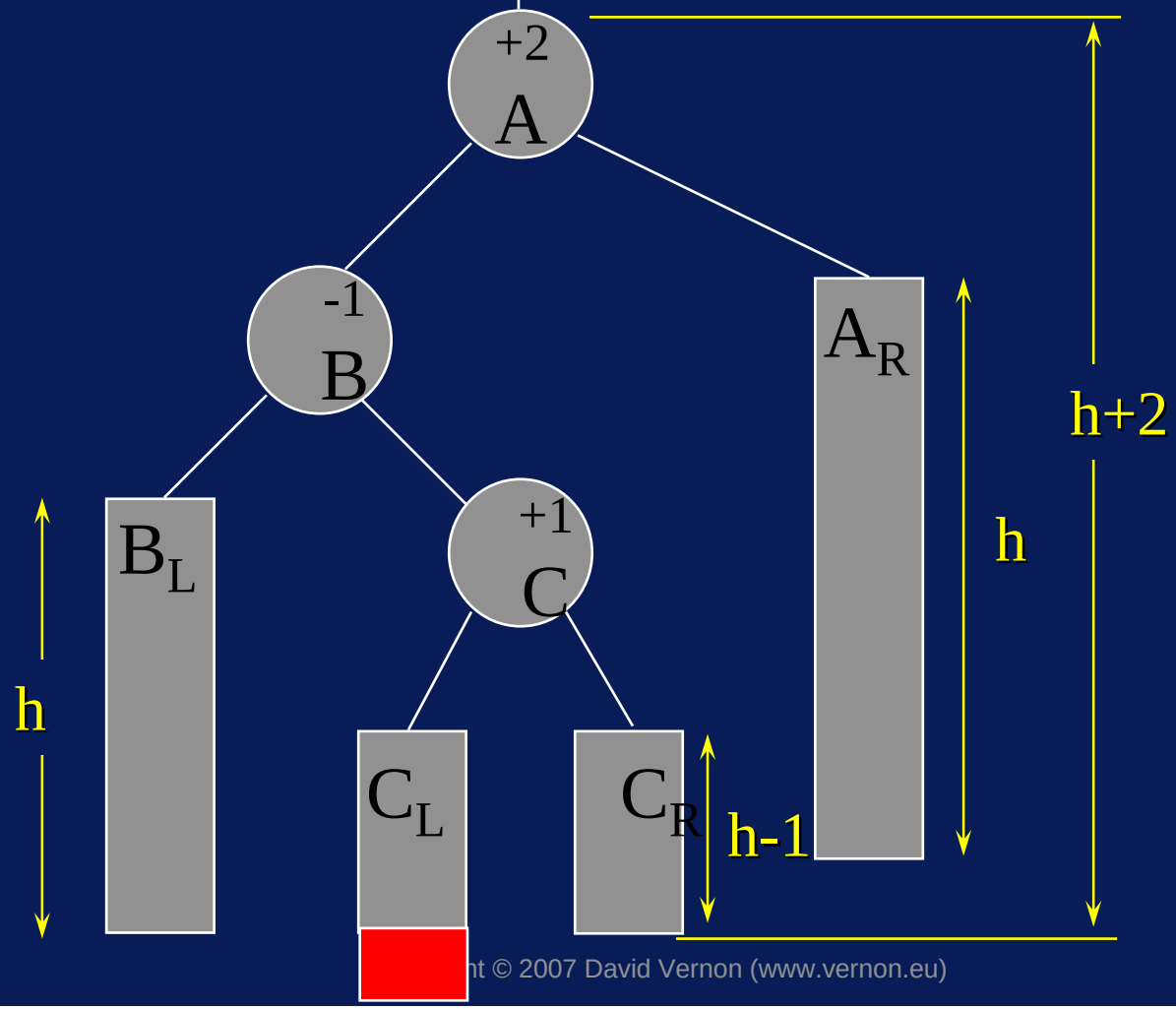
AVL Trees

Balanced Subtree

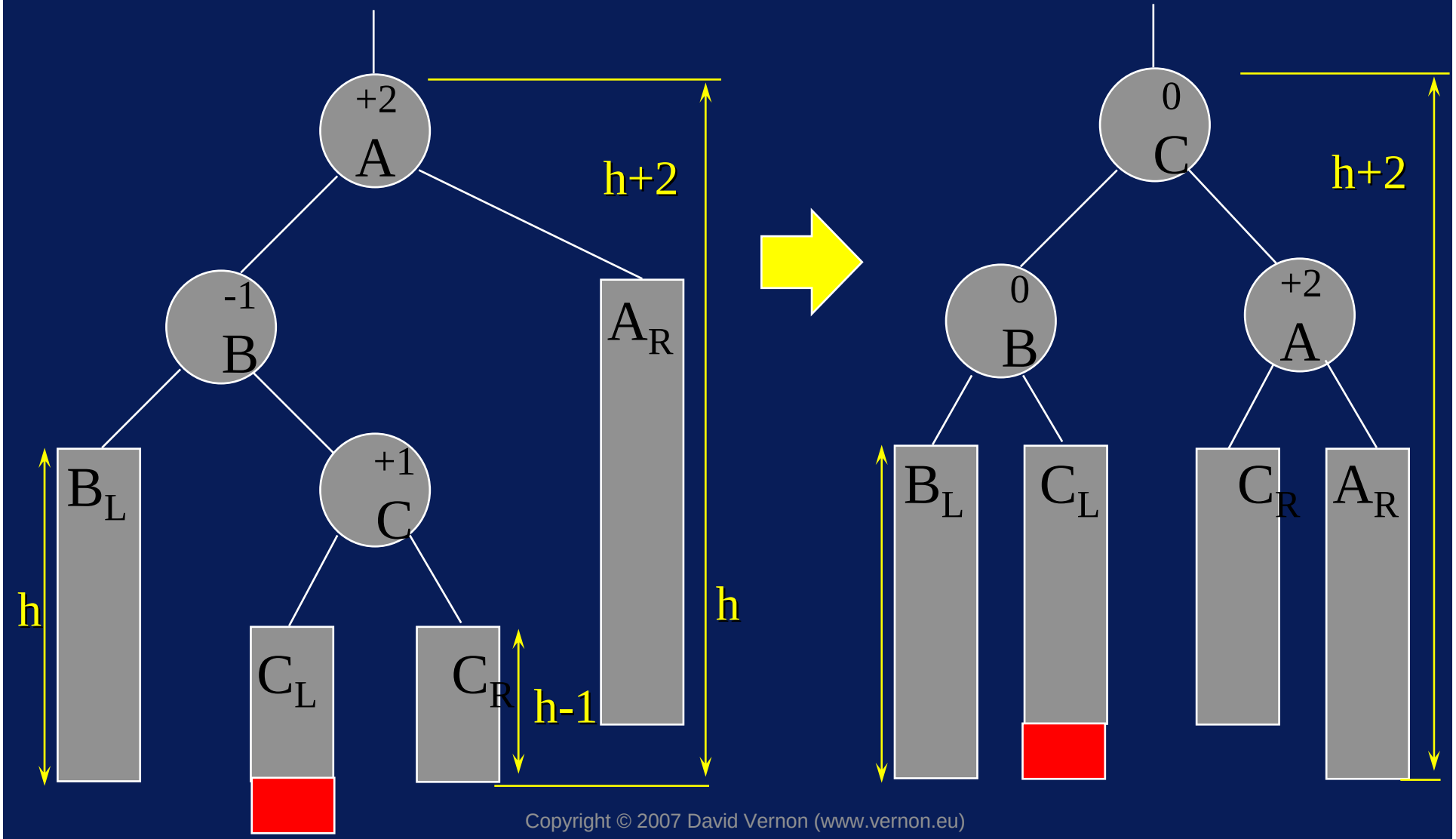


AVL Trees

Unbalanced following insertion

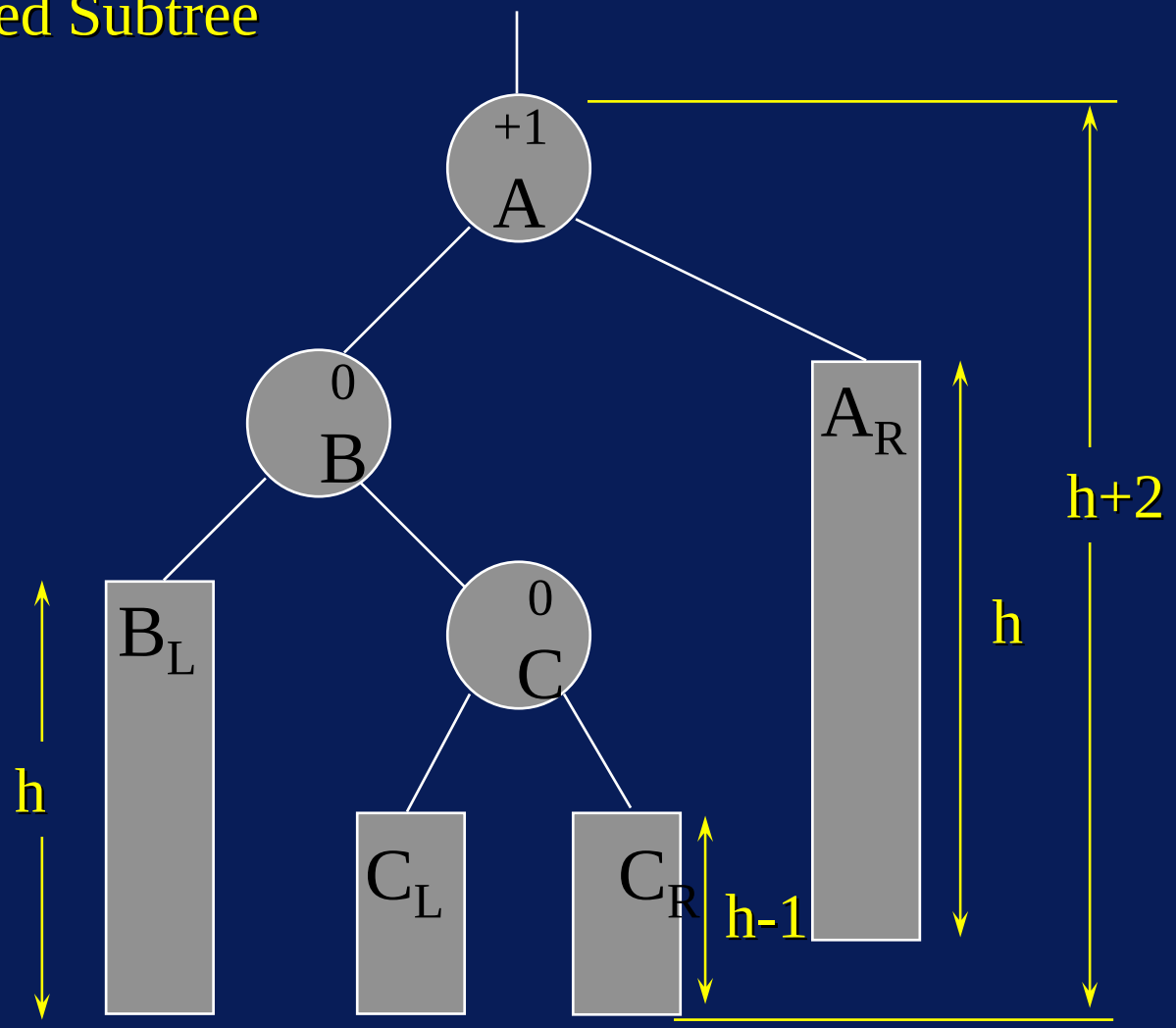


AVL Trees - LR rotation (b)



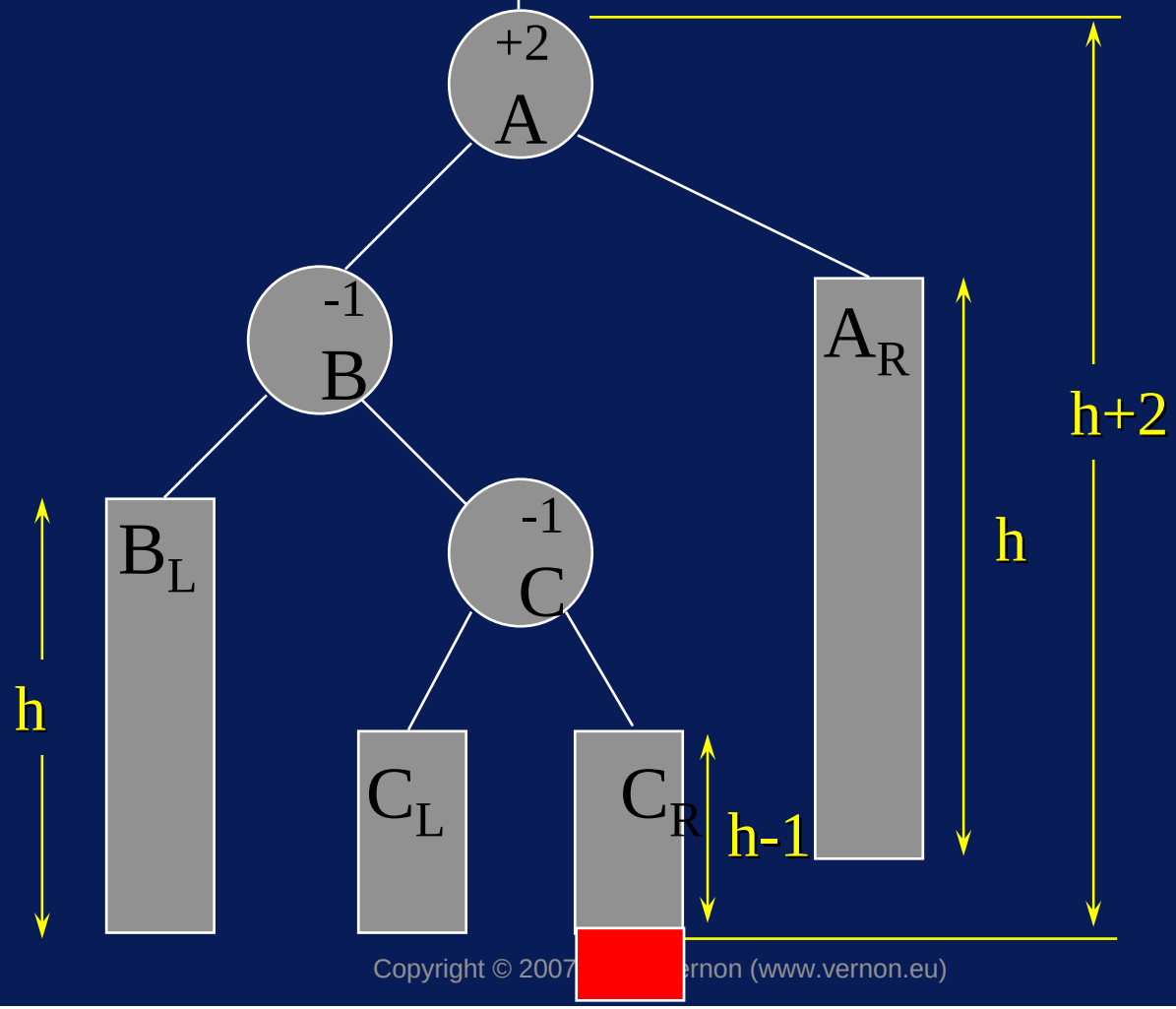
AVL Trees

Balanced Subtree

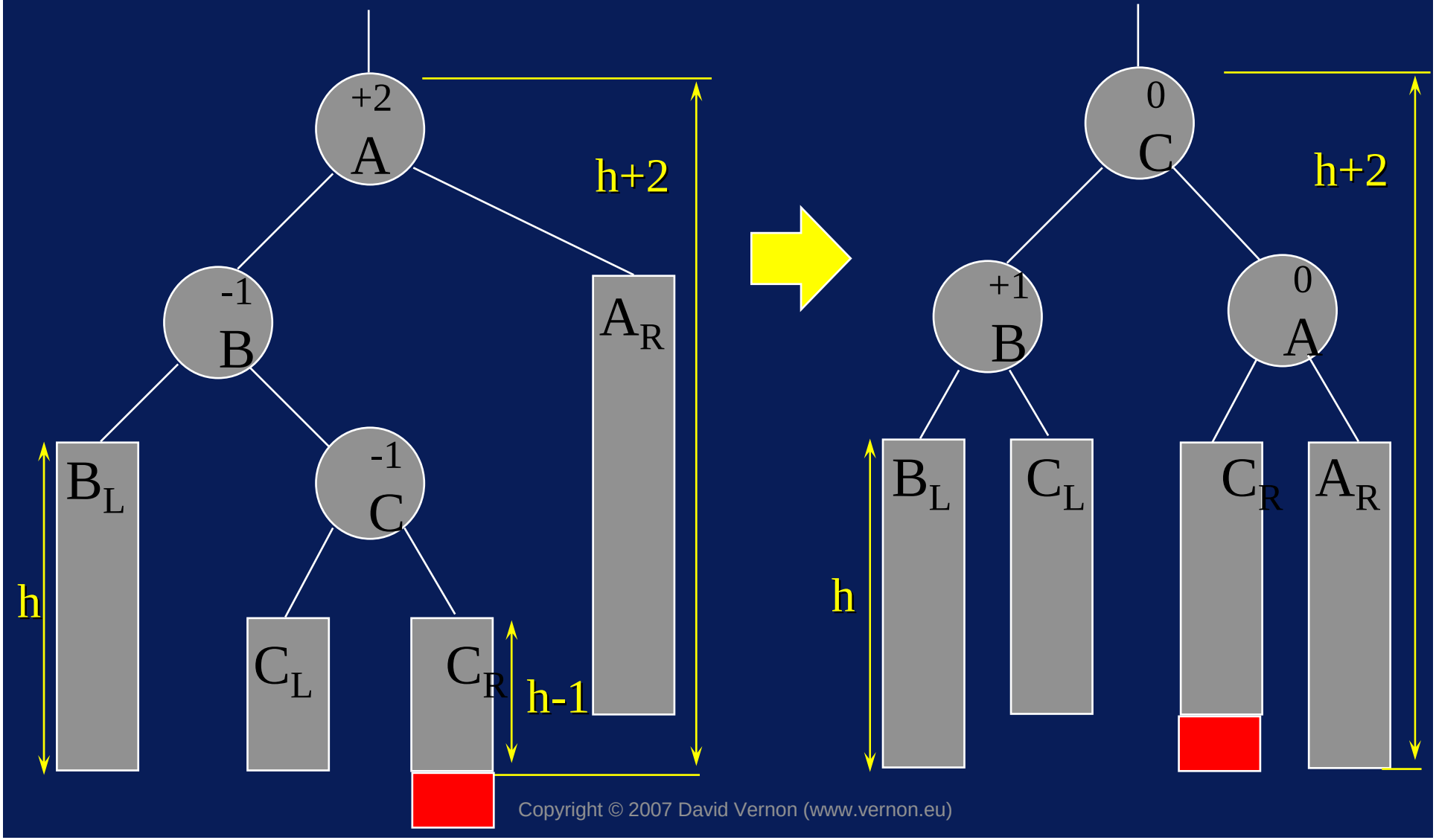


AVL Trees

Unbalanced following insertion

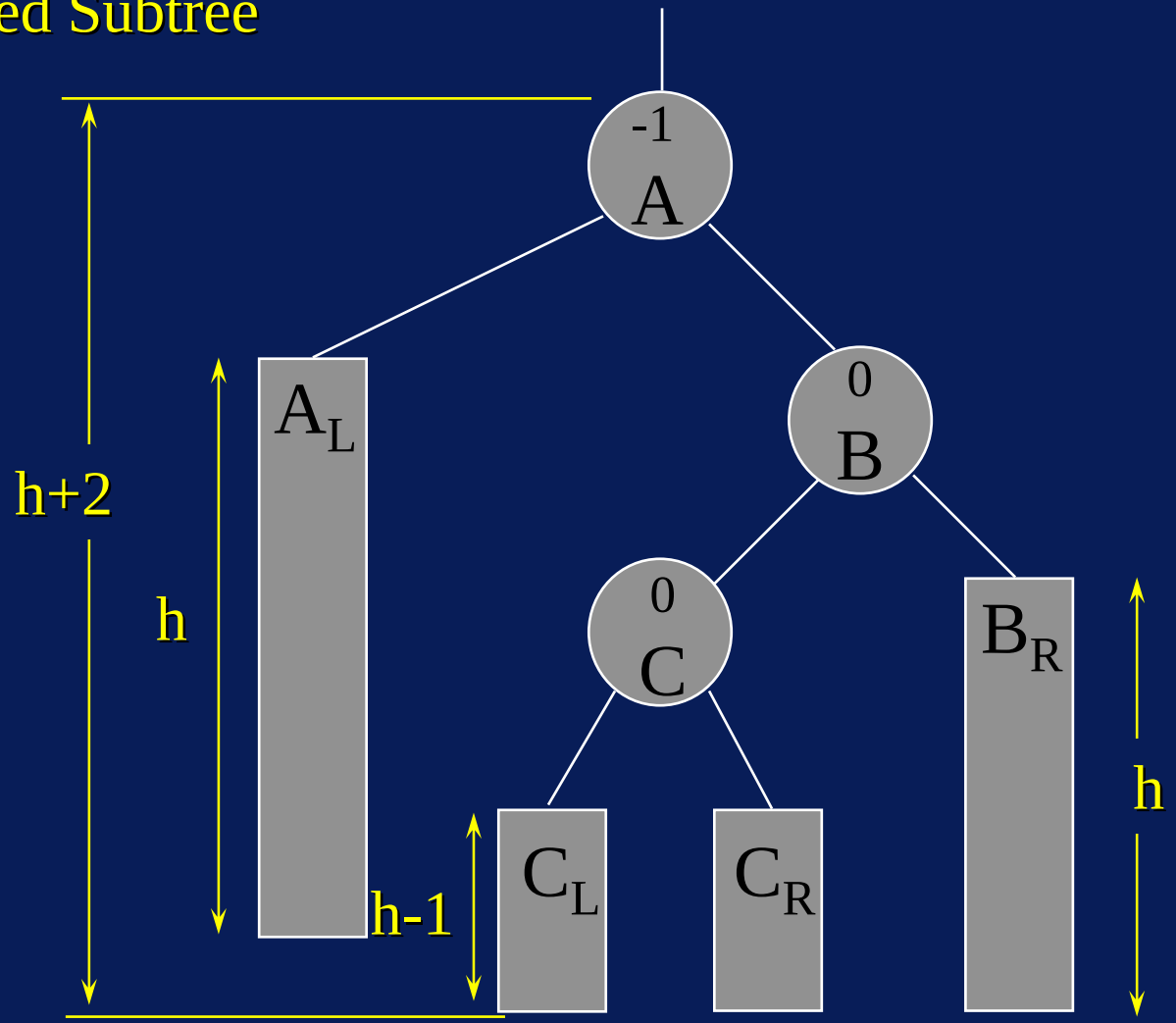


AVL Trees - LR rotation (c)



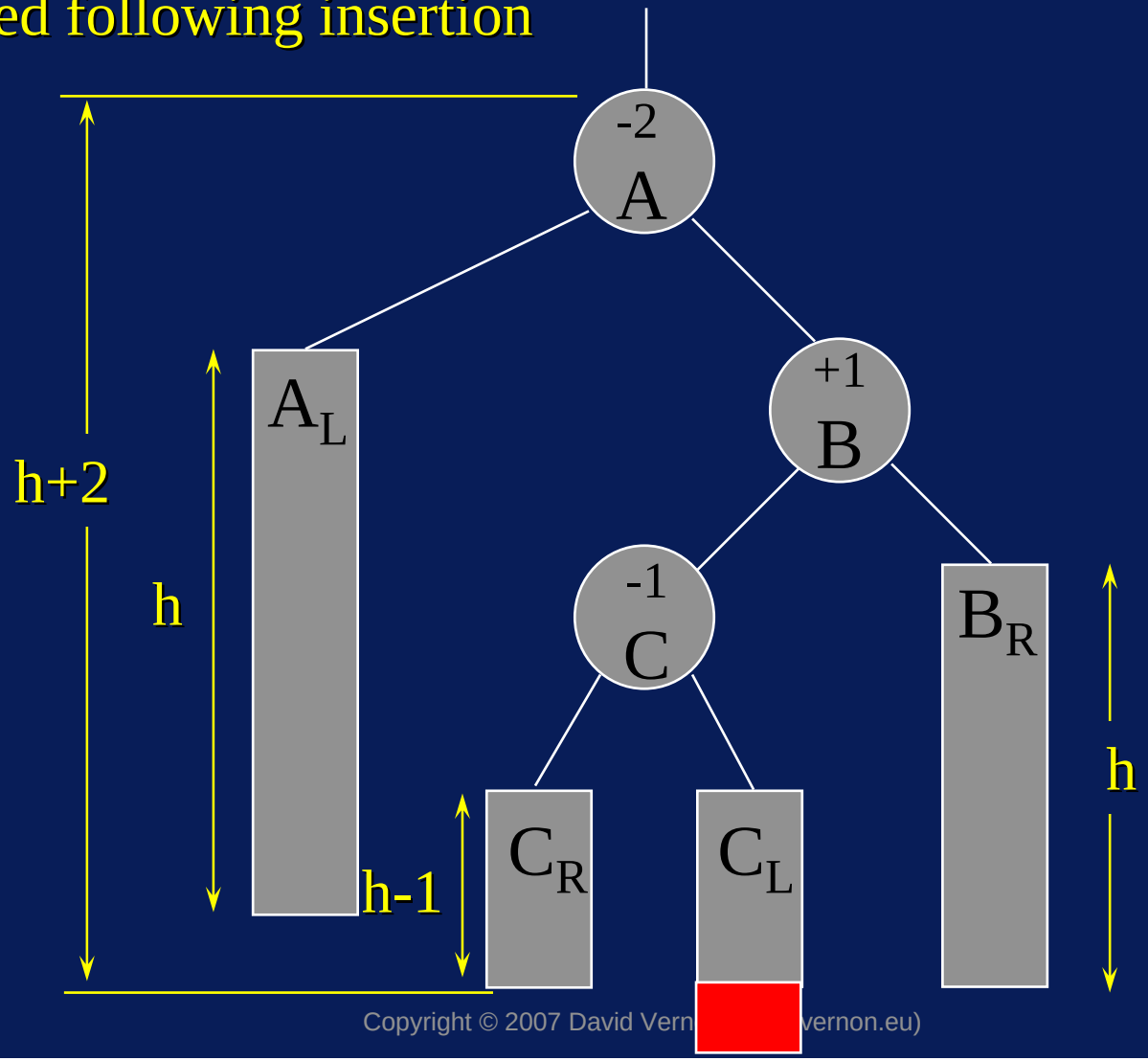
AVL Trees

Balanced Subtree

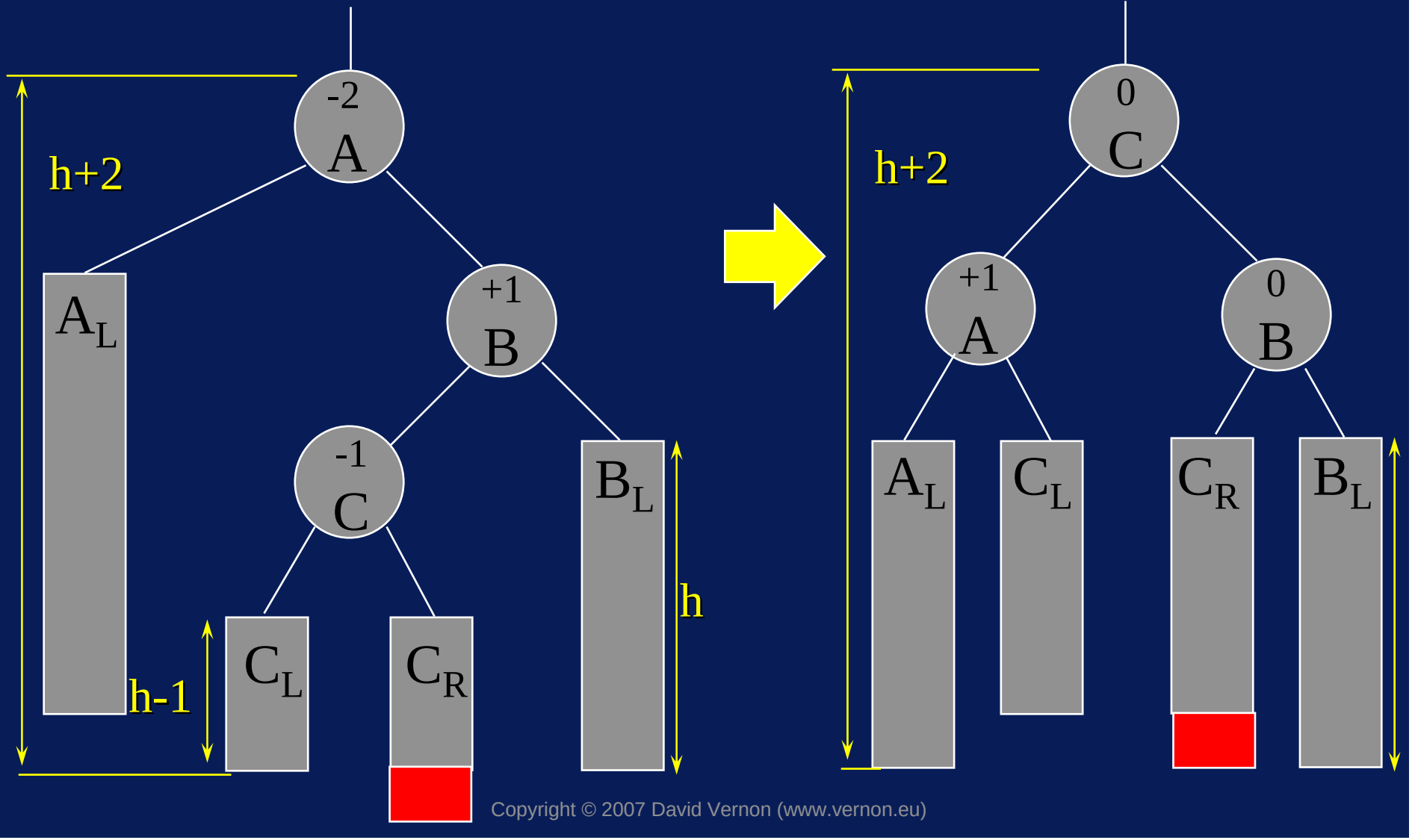


AVL Trees

Unbalanced following insertion



AVL Trees - RL rotation



AVL Trees

- To carry out this rebalancing we need to locate A, i.e. to window A
  - A is the nearest ancestor to Y whose balance factor becomes +2 or -2 following insertion
  - Equally, A is the nearest ancestor to Y whose balance factor was +1 or -1 before insertion
- We also need to locate F, the parent of A
  - This is where our complex window variable

AVL Trees

- Note in pasing that, since A is the nearest ancestor to Y whose balance factor was +1 or -1 before insertion, the balance factor of all other nodes on the part from A to Y must be 0
- When we re-balance the tree, the balance factors change (see diagrams above)
  - But changes only occur in subtree which is being rebalanced

AVL Trees

- The balance factors also change following an insertion which requires no rebalancing
- BF(A) is +1 or -1 before insertion
- Insertion causes height of one of A's subtrees to increase by 1
- Thus, BF(A) must be 0 after insertion (since, in this case, it's not +2 or -2)

Implementation of AVL_Insert()

```
PROCEDURE AVL_insert(e:elementtype; w:windowtype;
                     T: BINTREE);

(* We assume that variables of element type have two *)
(* data fields: the information field and a balance  *)
(* factor                                            *)
(* Assume also existence of two ADT functions to     *)
(* examine these fields:                             *)
(*                        Examine_BF(w, T)           *)
(*                        Examine_data(w, T)         *)
(* and one to modify the balance factor field        *)
(*                        Replace_BF(bf, w, T)       *)
var newnode: linktype;
begin
```

Implementation of AVL_Insert()

```
IF IsEmpty(T) (* special case *)
   THEN
      Insert(e, w, T);  (*insert as before *)
      Replace_BF(0, w, T)
   ELSE
      (* Phase 1: locate insertion point        *)
      (* A keeps track of most recent node with *)
      (* balance factor +1 or -1                *)
      A := w;
      WHILE ((NOT IsExternal(w, T)) AND
             (NOT (e.data = Examine_Data(w, T))) DO
         IF Examine_BF(w, T) <> 0 (* non-zero BF *)
            THEN
               A := w;
         ENDIF;
```

Implementation of AVL_Insert()

```
        IF (e.data < Examine_Data(w, T) )
           THEN
              Child(0, w, T)
           ELSE IF (e.data > Examine_Data(w, T) )
              Child(1, w, T)
           ENDIF
        ENDIF
     ENDWHILE
     (* If not found, then embark on Phase 2: *)
     (* insert & rebalance                    *)
     IF IsExternal(w, T)
        THEN
           Insert(e, w, T);  (*insert as before *)
           Replace_BF(0, w, T)
     ENDIF
```

Implementation of AVL_Insert()

```
(* adjust balance factors of nodes on path     *)
(* from A to parent of newly-inserted node     *)
(* By definition, they will have had BF=0      *)
(* and so must now change to +1 or -1          *)
(* Let d = this change,                        *)
(* d = +1 ... insertion in A's left subtree    *)
(* d = -1 ... insertion in A's right subtree   *)

IF (e.data < Examine_Data(A, T) )
   THEN
      v:= A;
      Child(0, v, T)
      B:= v;
      d := +1
   ELSE
```

Implementation of AVL_Insert()

```
      ELSE
         v:= A; Child(1, v, T)
         B:= v;
         d := -1
   ENDIF
   WHILE ((NOT IsEqual(w, v))) DO
      IF (e.data < Examine_Data(v, T) )
         THEN
            ReplaceBF(+1, v, T);
            Child(0, v, T) (* height of Left ^ *)
         ELSE
            ReplaceBF(-1, v, T);
            Child(1, v, T) (* height of Right ^ *)
      ENDIF
   ENDWHILE
```

Implementation of AVL_Insert()

```
(* check to see if tree is unbalanced *)

IF (ExamineBF(A, T) = 0 )
   THEN
      ReplaceBF(d, A, T) (* still balanced *)
   ELSE
      IF ((ExamineBF(A, T) + d) = 0)
         THEN
            ReplaceBF(0, A, T)(*still balanced*)
         ELSE

            (* Tree is unbalanced      *)
            (* determine rotation type *)
```

Implementation of AVL_Insert()

```
(* Tree is unbalanced       *)
(* determine rotation type *)

IF d = +1
   THEN  (* left imbalance *)
      IF ExamineBF(B) = +1
         THEN (* LL Rotation *)
            (* replace left subtree of A *)
            (* with right subtree of B   *)
            temp := B; Child(1, temp, T);
            ReplaceChild(0, A, T,  temp);

            (* replace right subtree of B with A *)
            ReplaceChild(1, B, T, A);
```

Implementation of AVL_Insert()

```
        (* replace right subtree of B with A *)
        ReplaceChild(1, B, T, A);

        ReplaceBF(0, A, T);
        ReplaceBF(0, B, T);
    ELSE (* LR Rotation *)
        C := B; Child(1, C, T);
        C_L := C; Child(0, C_L, T);
        C_R := C; Child(1, C_R, T);
        ReplaceChild(1, B, T, C_L);
        ReplaceChild(0, A, T, C_R);
        ReplaceChild(0, C, T, B);
        ReplaceChild(1, C, T, A);
```

Implementation of AVL_Insert()

```
IF ExamineBF(C) = +1 (* LR(b) *)
   THEN
      ReplaceBF(-1, A, T);
      ReplaceBF(0, B, T);
   ELSE
      IF ExamineBF(C) = -1 (* LR(c) *)
         THEN
            ReplaceBF(+1, B, T);
            ReplaceBF(0, A, T);
         ELSE                  (* LR(a) *)
            ReplaceBF(0, A, T);
            ReplaceBF(0, B, T);
      ENDIF
ENDIF
```

Implementation of AVL_Insert()

```
            (* B is new root *)
            ReplaceBF(0, C, T);
            B := C
        ENDIF (* LR rotation *)
    ELSE (* right imbalance *)

        (* this is symmetric to left imbalance *)
        (* and is left as an exercise!          *)

ENDIF (* d = +1 *)
```

Implementation of AVL_Insert()

```
                    (* the subtree with root B has been *)
                    (* rebalanced and it now replaces   *)
                    (* A as the root of the originally  *)
                    (* unbalanced tree                  *)

                    ReplaceTree(A, T, B)
                    (* Replace subtree A with B in T     *)
                    (* Note: this is a trivial operation *)
                    (* since we are using a complex      *)
                    (* window variable                   *)
                ENDIF
            ENDIF
        ENDIF
END (* AVL_Insert() *)
```

Red-Black Trees

Red-Black Trees

- The goal of height-balanced trees is to ensure that the tree is as complete as possible and that, consequently, it has minimal height for the number of nodes in the tree
- As a result, the number of probes it takes to search the tree (and the time it takes) is minimized.

Red-Black Trees

- A perfect or a complete tree with n nodes has height $O(\log_2 n)$
  - So the time it takes to search a perfect or a complete tree with n nodes is $O(\log_2 n)$
- A skinny tree could have height O(n)
  - So the time it takes to search a skinny tree can be O(n)
- Red-Black trees are similar to AVL trees in that they allow us to construct trees which have a guaranteed search

Red-Black Trees

- A red-black tree is a binary tree whose nodes can be coloured either red or black to satisfy the following conditions:
  - Black condition: Each root-to-frontier path contains exactly the same number of black nodes
  - Red condition: Each red node that is not the root has a black parent
  - Each external node is black

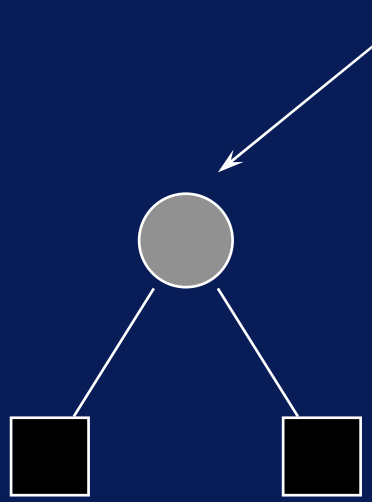
Red-Black Trees

- A red-black search tree is a red-black tree that is also a binary search tree
- For all n>= 1, ever red-black tree of size n has height $O(\log_2 n)$
  - Thus, red-black trees provide a guaranteed worst-case search time of $O(\log_2 n)$

Red-Black Trees

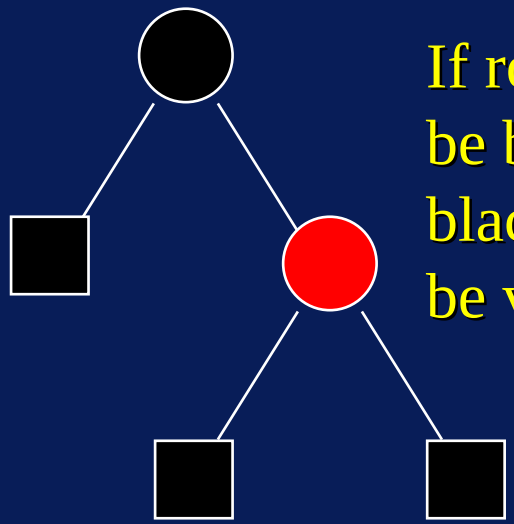
- Red-black tree (condition 3)

Red-Black Trees



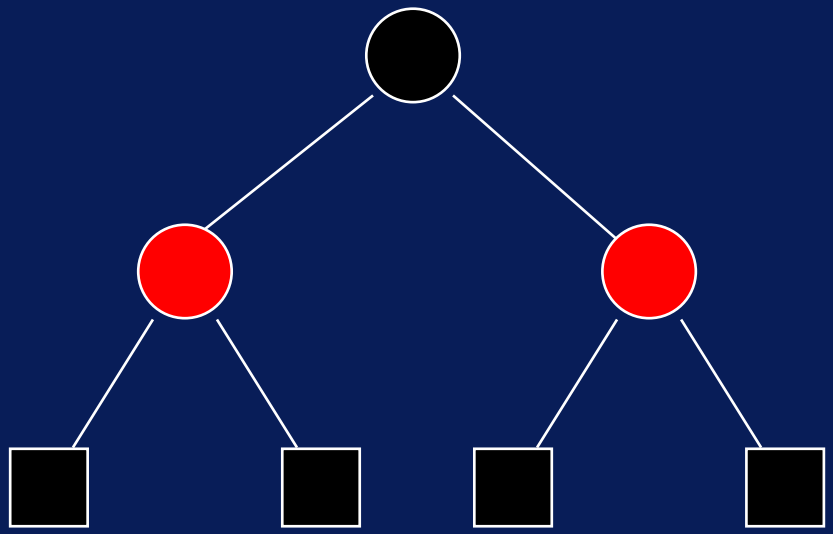
Red-black tree (condition 3)

Red-Black Trees

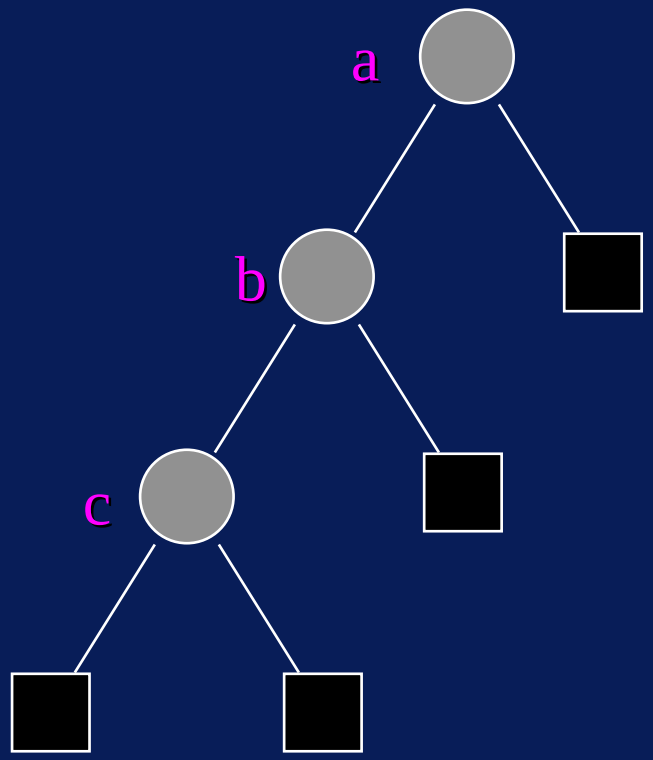


If root was red, then right child would have to be black (if it was red, it would have to have a black parent) but then the black condition would be violated.

Red-Black Trees



Red-Black Trees



To satisfy black condition, either

(1) node a is black and nodes b and c are red, or
(2) nodes a, b, and c are red.

In both cases, a red condition is violated.

Therefore, this is not a red-black tree

Red-Black Trees

- For all n >= 1, every red-black tree of size n has height $O(\log_2 n)$
- Thus, red-black trees provide a guaranteed worst-case search time of $O(\log_2 n)$

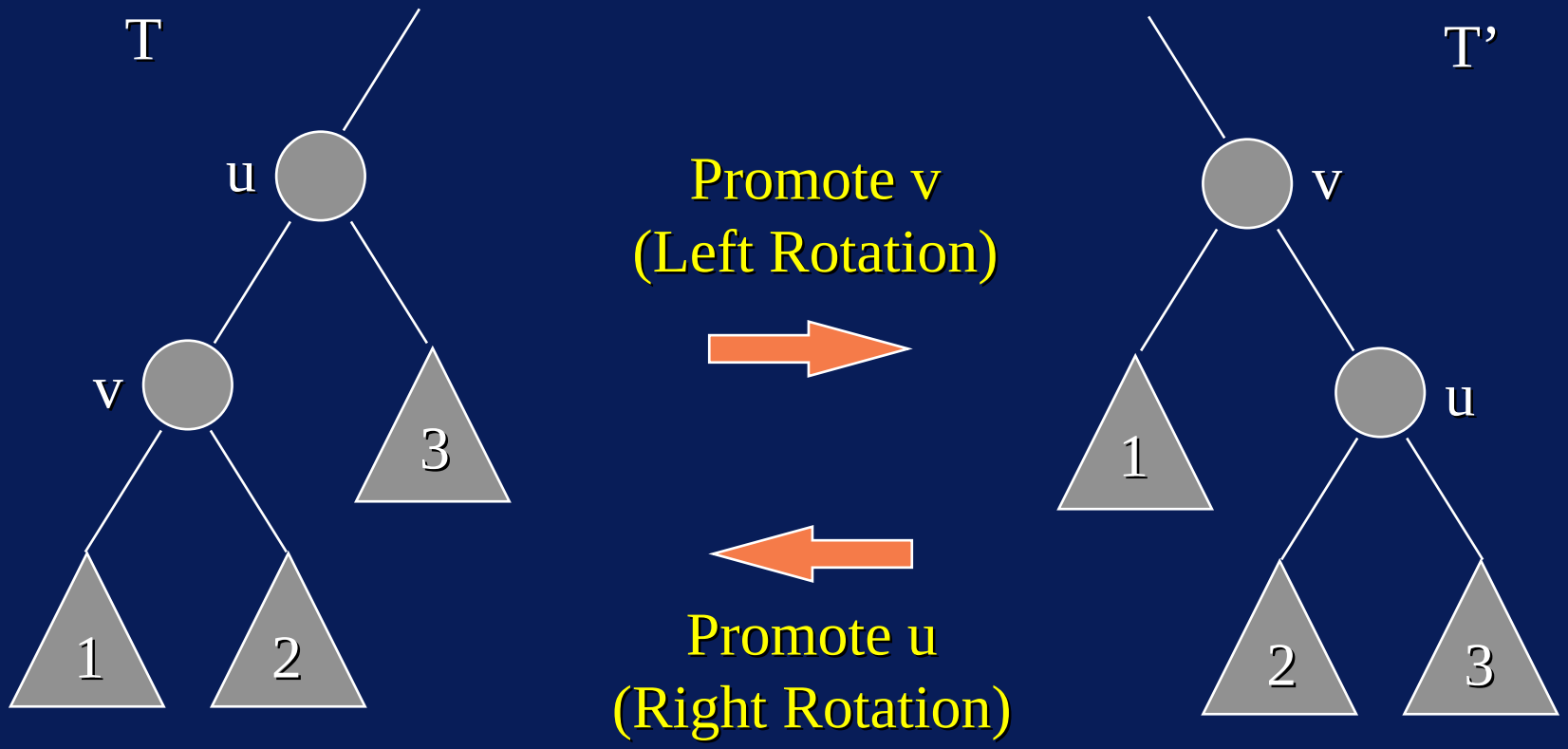
Red-Black Trees

- Insertions and deletions can cause red and black conditions to be violated
- Trees then have to be restructured
- Restructuring called a promotion (or rotation)
  - Single promotion
  - 2 promotion

Red-Black Trees

- Single promotion
- Also referred to as
  - single (left) rotation
  - single (right) rotation
- Promotes a node one level

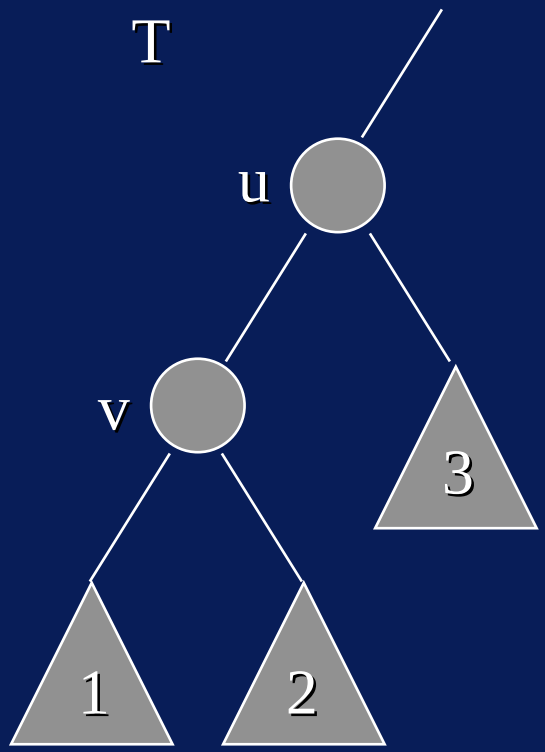
Red-Black Trees



Red-Black Trees

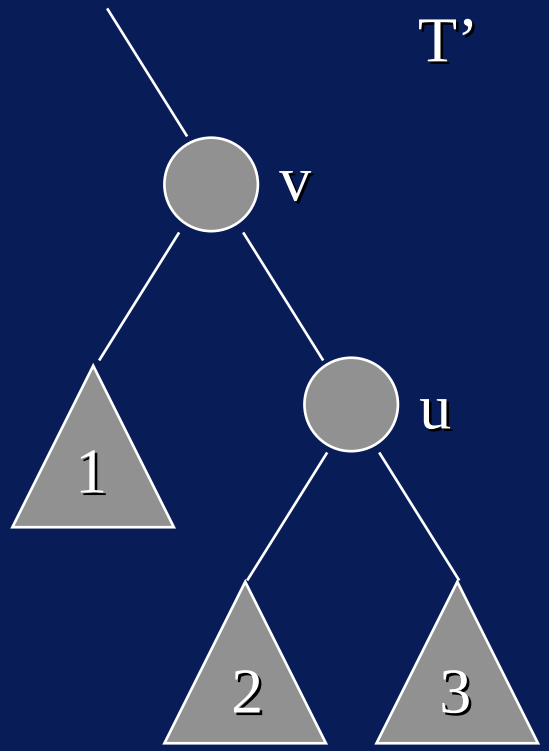
- A single promotion (Left Rotation or Right Rotation) preserves the binary-search condition
- Same manner as an AVL rotation

Red-Black Trees



Promote v
(Left Rotation)

Promote u
(Right Rotation)



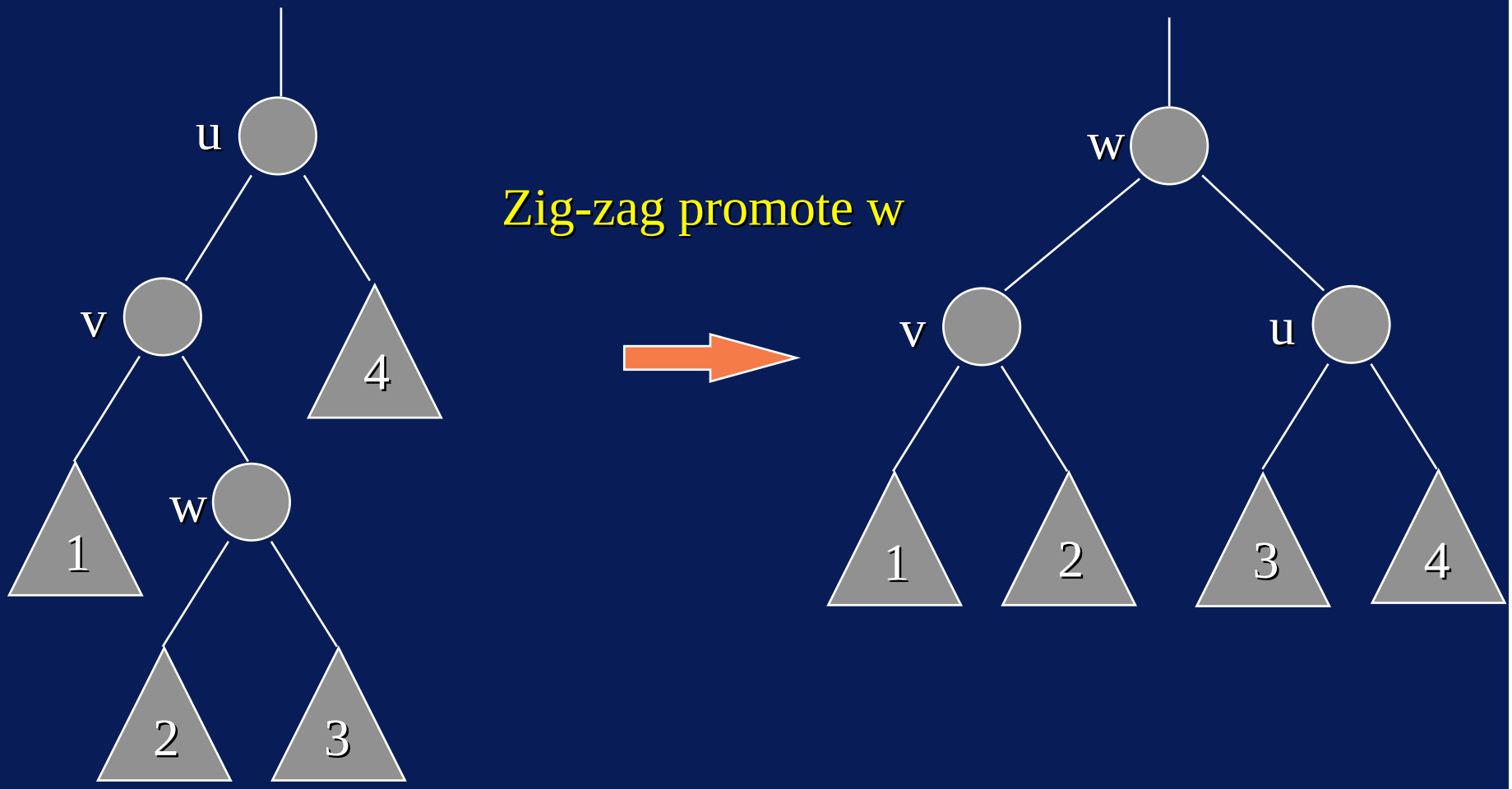
keys(1) < key(v) < key(u)
key(v) < keys(2) < key(u)
key(u) < keys(3)

keys(1) < key(v)
key(v) < keys(2) < key(u)
key(v) < key(u) < keys(3)

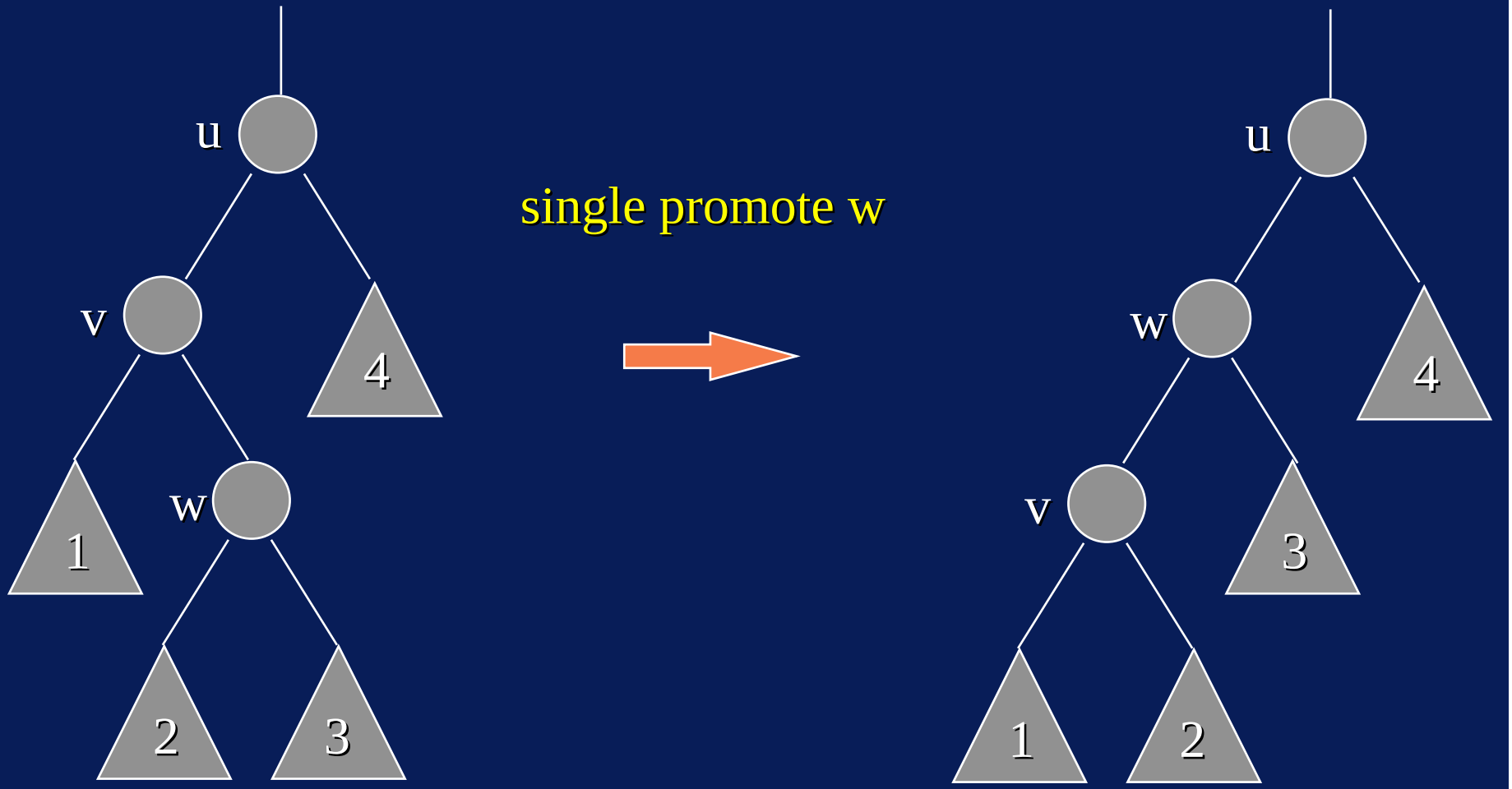
Red-Black Trees

- 2-Promotion
- Zig-zag promotion
- Composed of two single promotions
- And hence preserves the binary-search condition

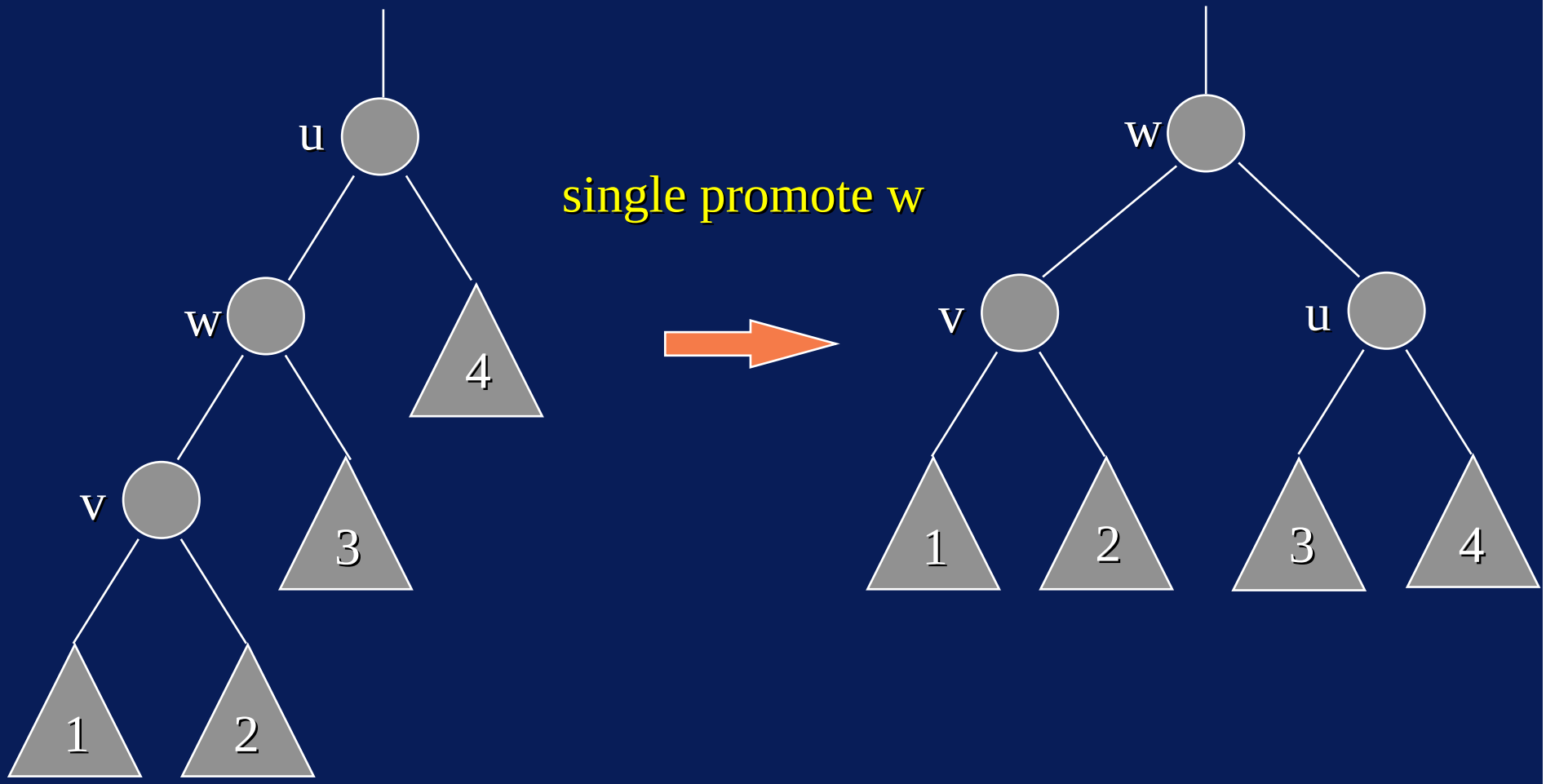
Red-Black Trees



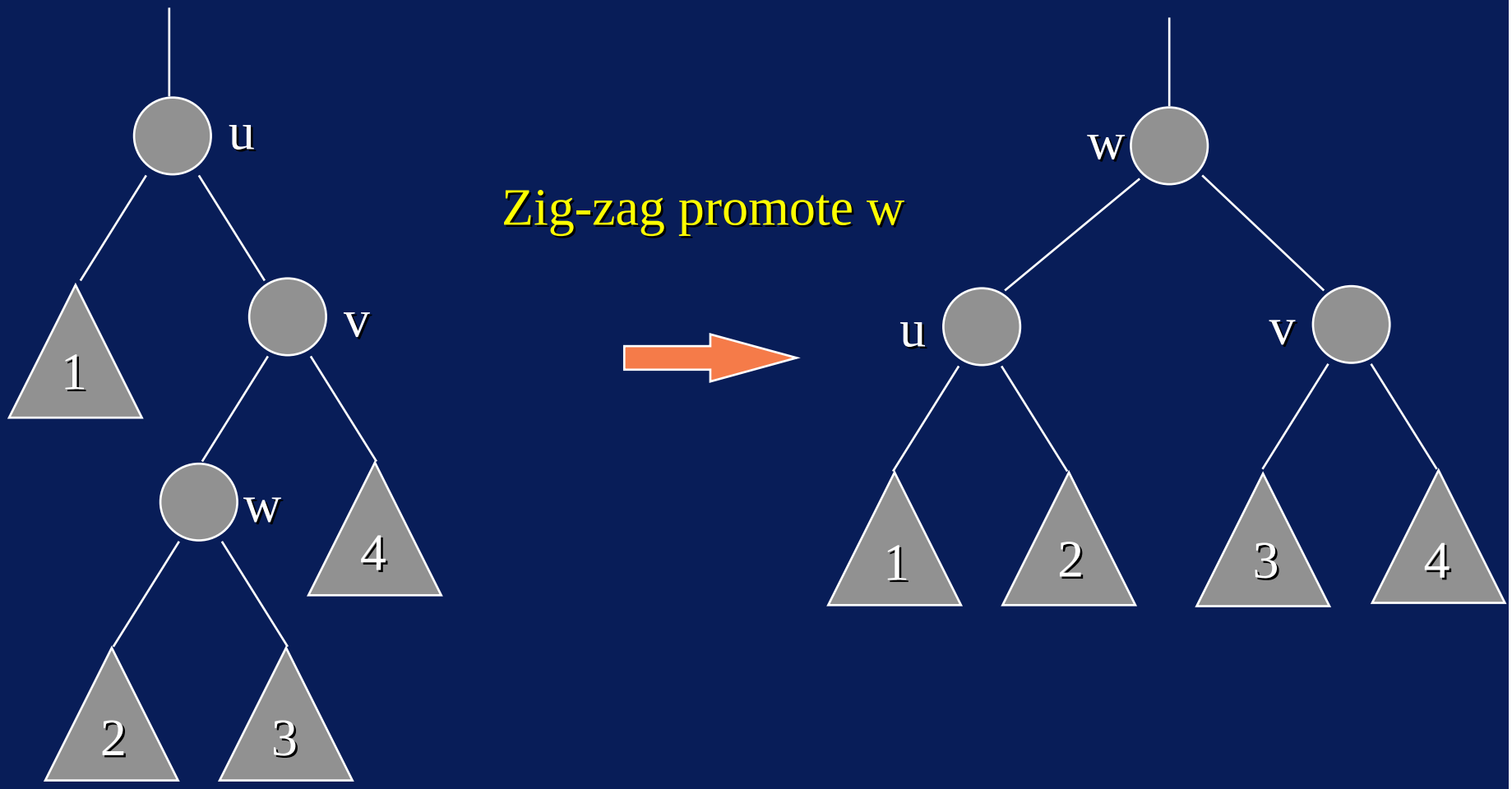
Red-Black Trees



Red-Black Trees



Red-Black Trees



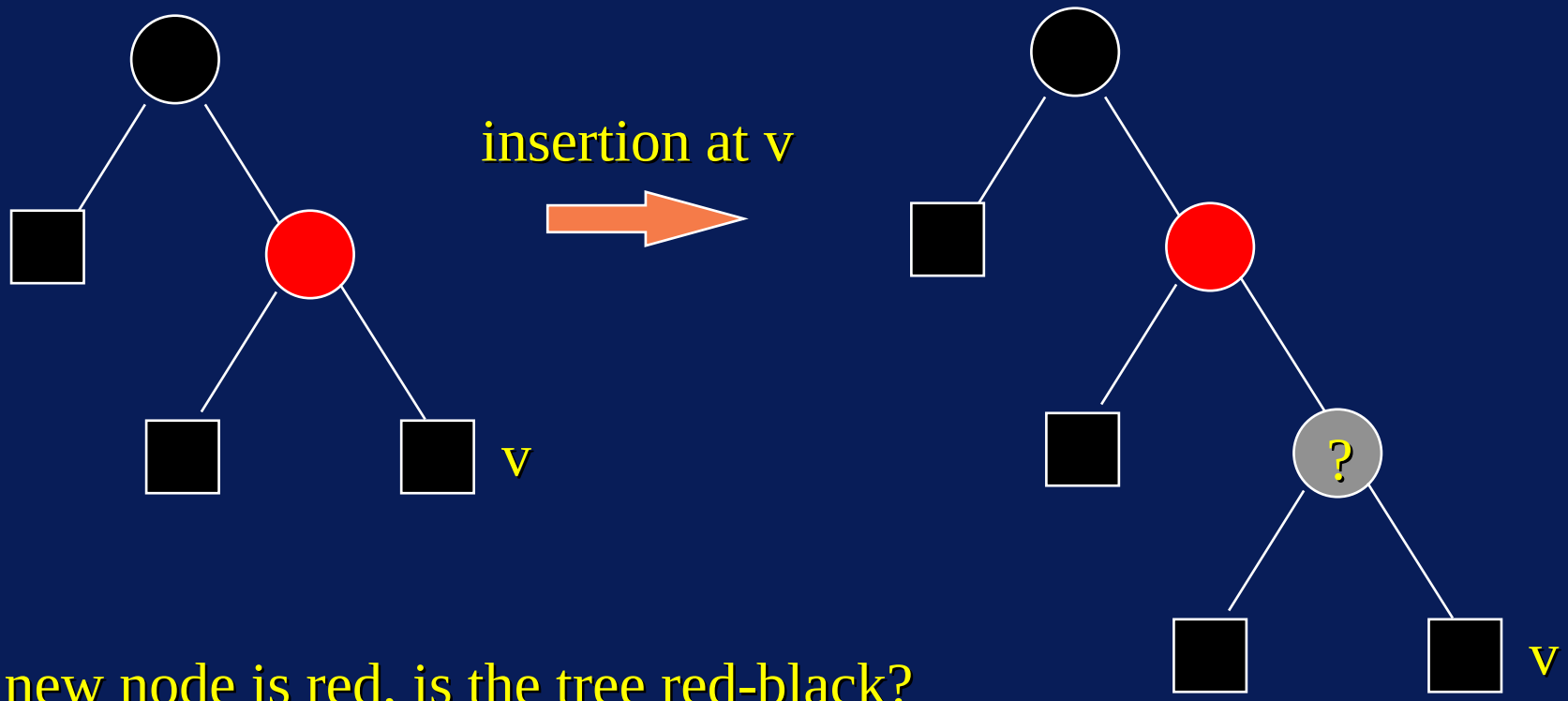
Red-Black Trees

- Insertions
- A red-black tree can be searched in logarithmic time, worst case
- Insertions may violate the red-black conditions necessitating restructuring
- This restructuring can also be effected in logarithmic time
- Thus, an insertion (or a deletion) can be effected in logarithmic time

Red-Black Trees

- Just as with AVL trees, we perform the insertion by
  - first searching the tree until an external node is reached (if the key is not already in the tree)
  - then inserting the new (internal) node
- We then have to recolour and restructure, if necessary

Red-Black Trees

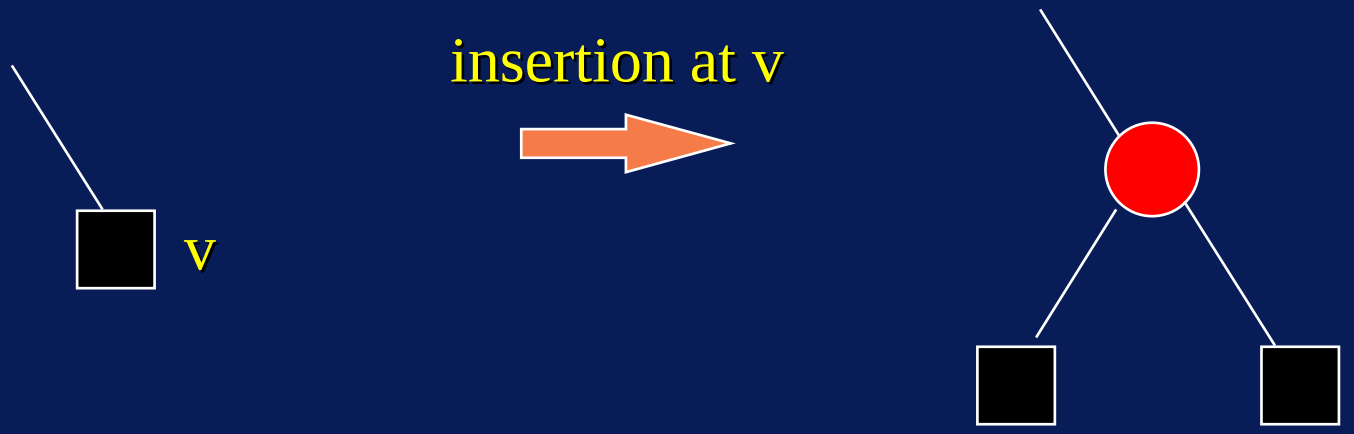


If new node is red, is the tree red-black?
If the new node is black, is the tree red-black?

Red-Black Trees

- Recolouring:
  - Colour new node red
  - This preserves the black condition
  - but may violate the red condition
- Red condition can be violated only if the parent of an internal node is also red
- Must transform this ‘almost red-black tree’ into a red-black tree

Red-Black Trees

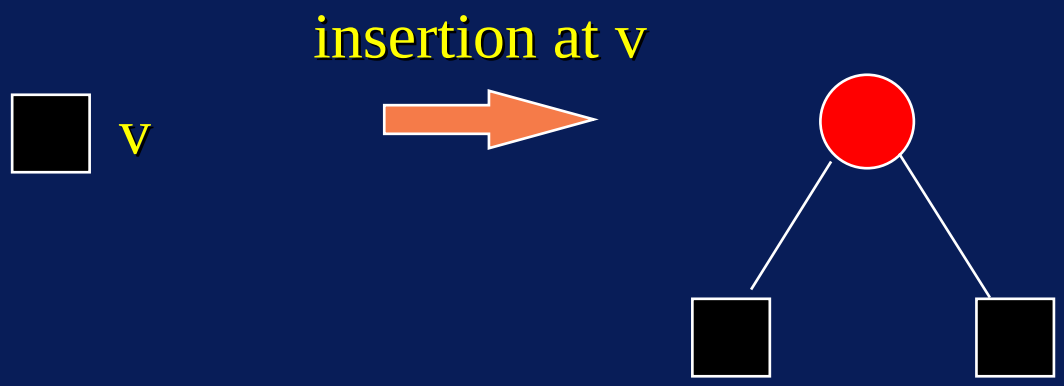


Red-Black Trees

- Recolouring and restructuring algorithm
  - The node u is a red node in a BST, T
  - u is the only candidate violating node
  - Apart from u, the tree T is red-black

Red-Black Trees

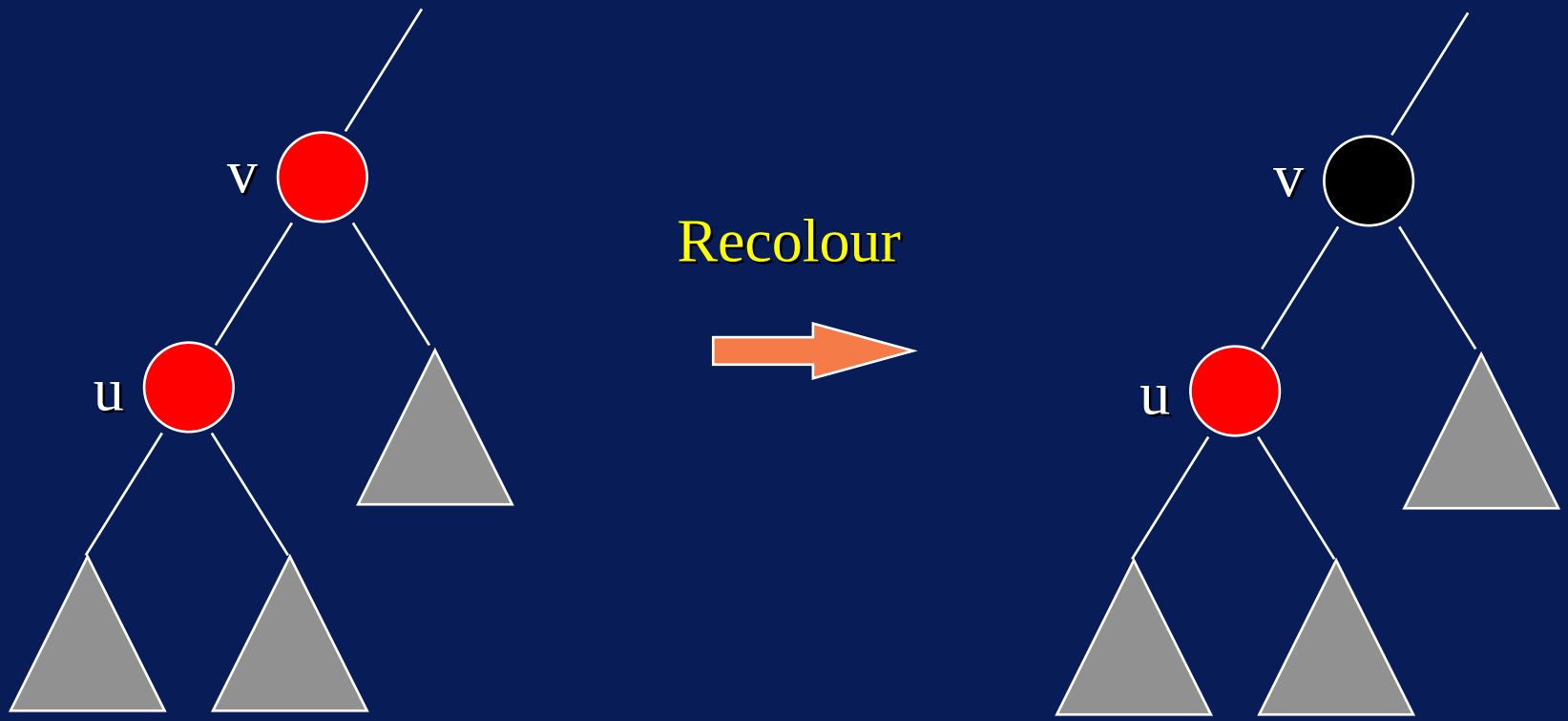
- Case 1:
  - u is the root
  - T is red-black



Red-Black Trees

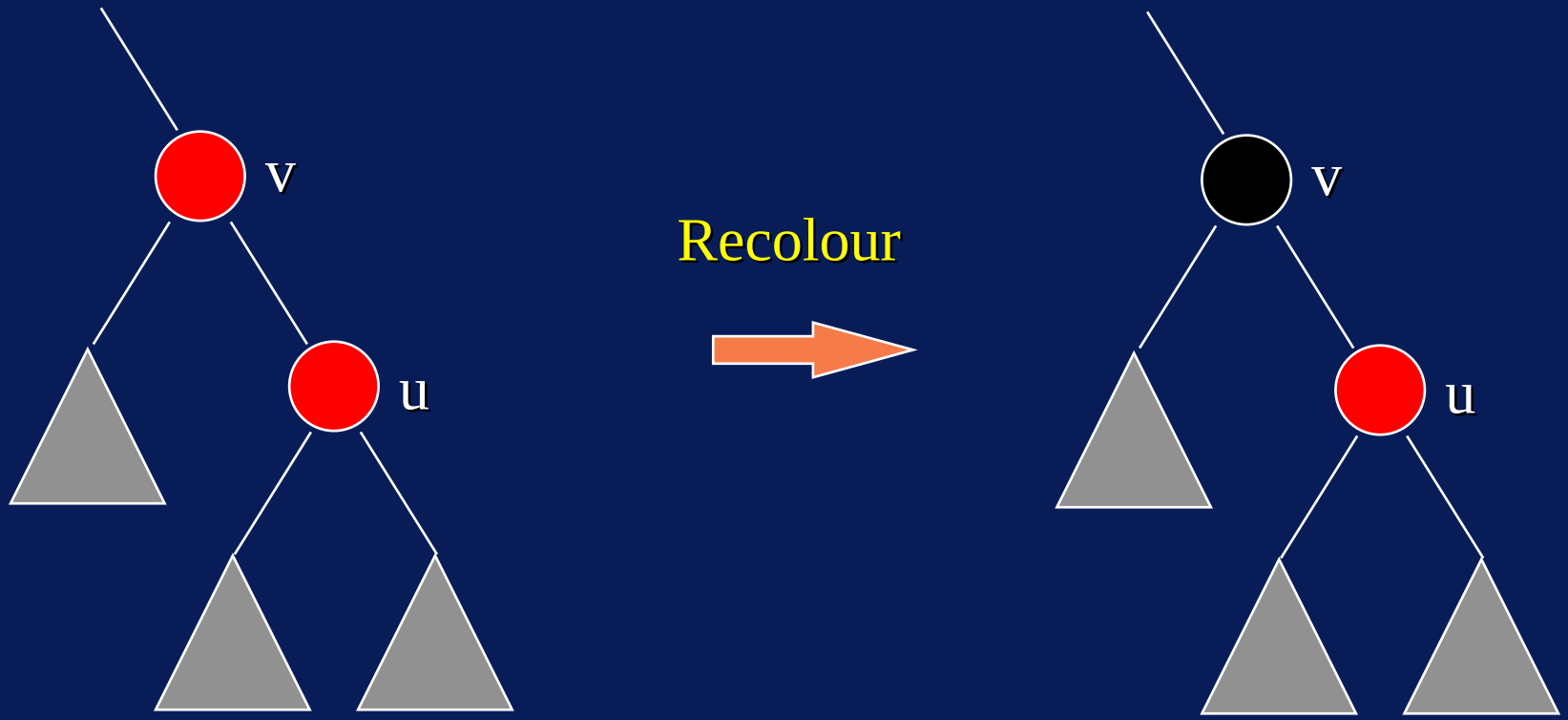
- Case 2:
  - u is not the root
  - its parent v is the root
  - Colour v black

Red-Black Trees



Is there anything unexpected about this figure?

Red-Black Trees



Is there anything unexpected about this figure?

Red-Black Trees

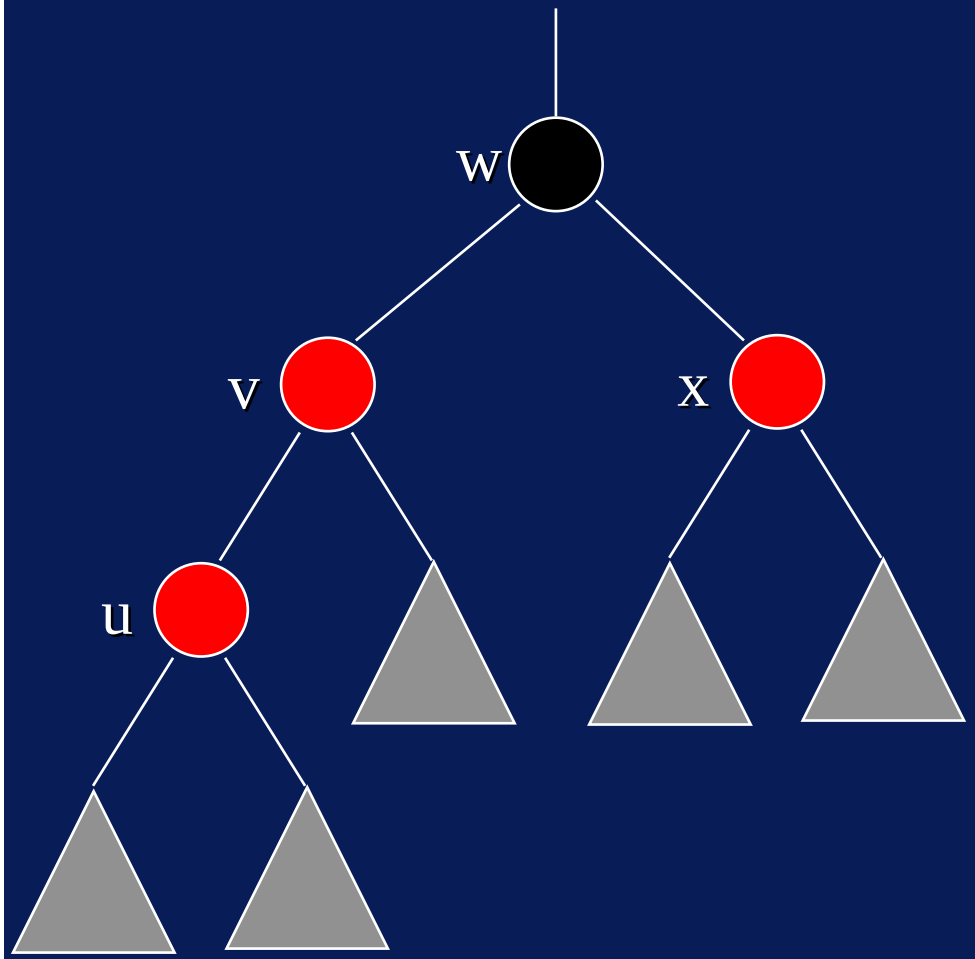
- Case 3:
  - u is not the root,
  - its parent v is not the root,
  - v is the left child of its parent w
  - (x is the right child of w, i.e. x is v's sibling)

Red-Black Trees

- Case 3.1:
  - x is red
  - Colour v and x black and w red
  - *Repeat the restructuring with u := w*

    (since the recolouring of w to red may cause a red violation)

Red-Black Trees



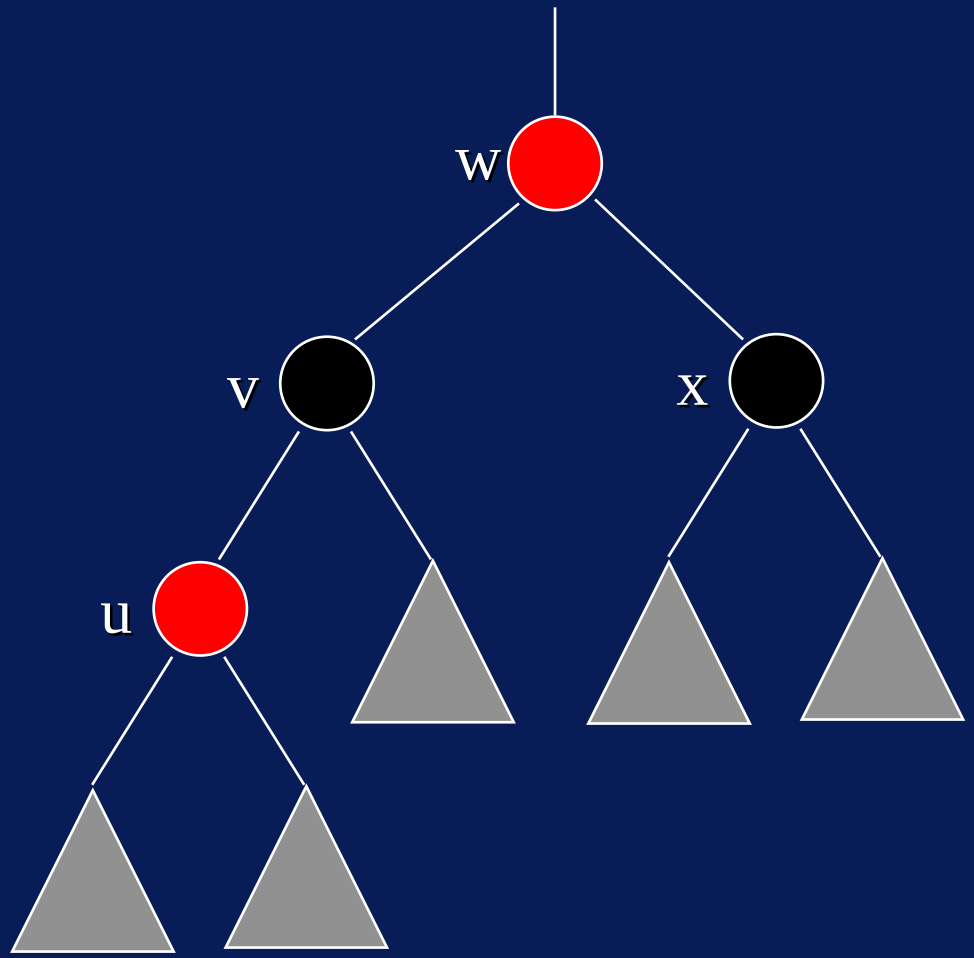
Note:
w must be black,
v must be red,
u must be red.
Why?

Recolour

Red-Black Trees

- u must be red because we colour new nodes that way by convention (to preserve the black condition)
- v must be red because otherwise it would be black and then we wouldn’t have violated the red condition and we wouldn’t be restructuring anything!
- w must be black because every red node (that isn’t the root) has a black parent

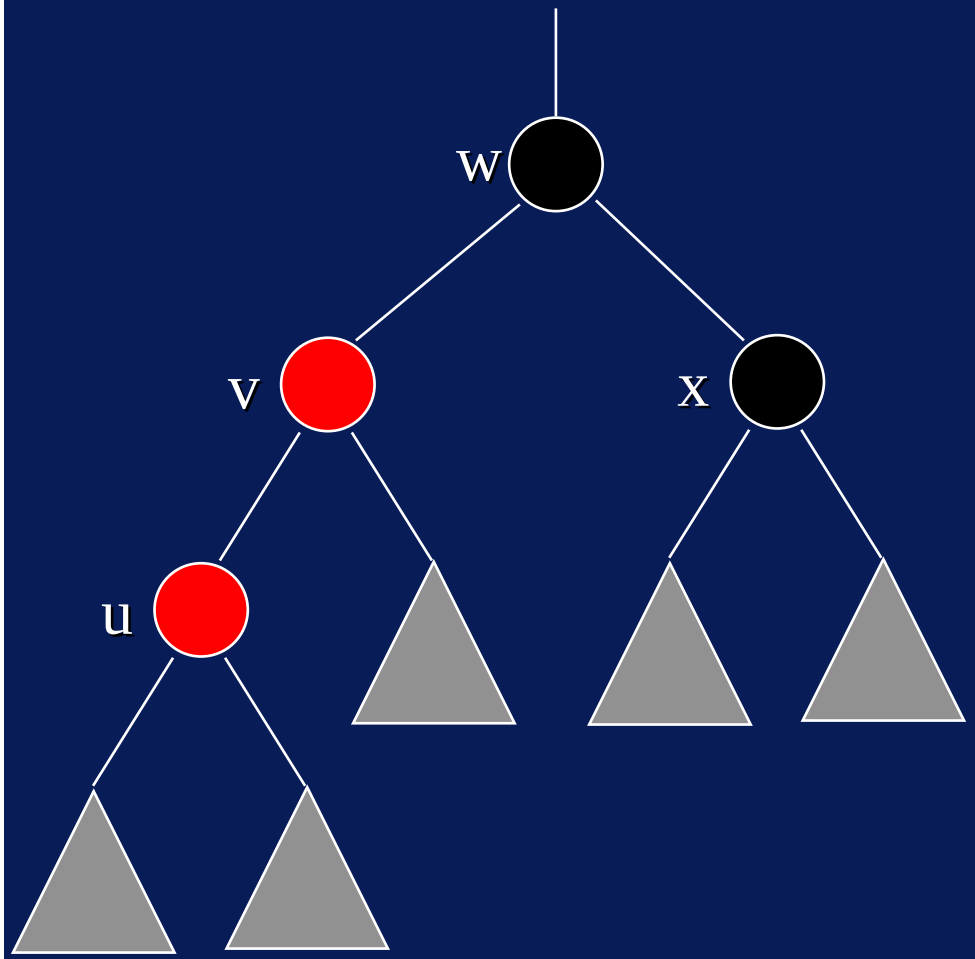
Red-Black Trees



Red-Black Trees

- Case 3.2:
  - x is black
  - u is the left child of v
  - Promote v
  - Colour v black
  - Colour w red

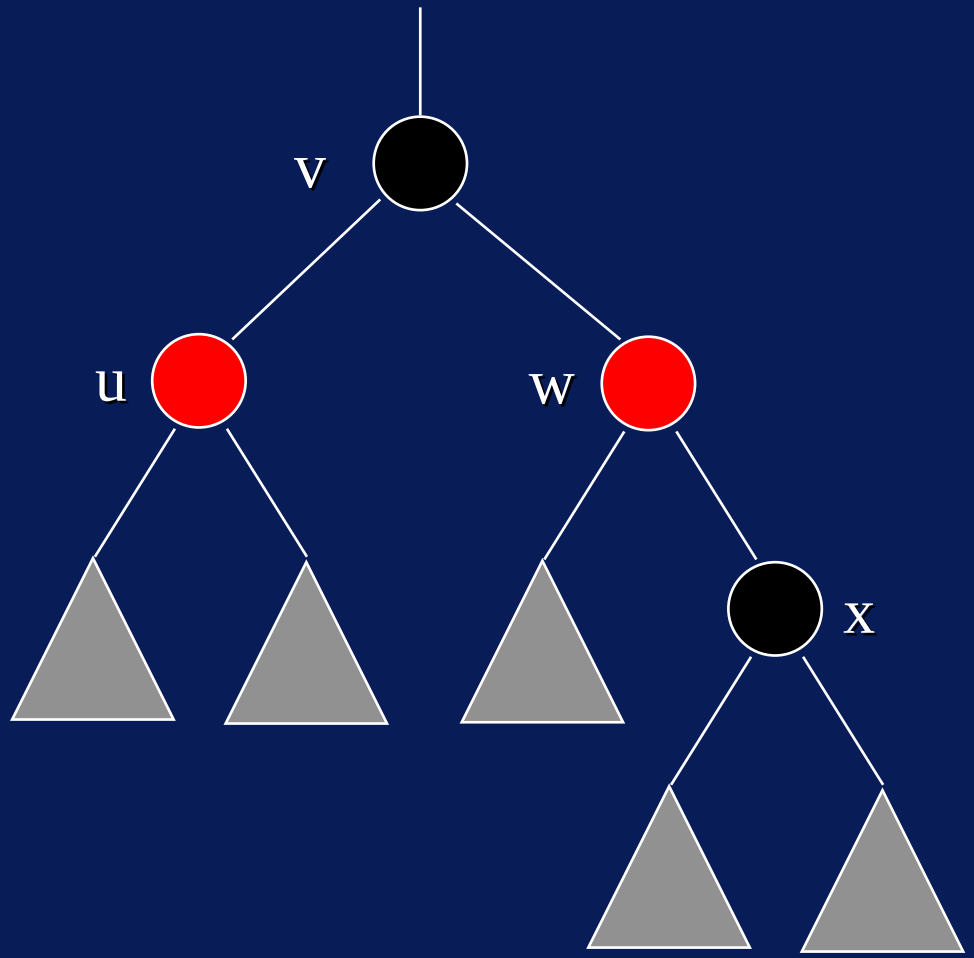
Red-Black Trees



Restructure and recolour

Promote v;
colour v black;
colour w red

Red-Black Trees



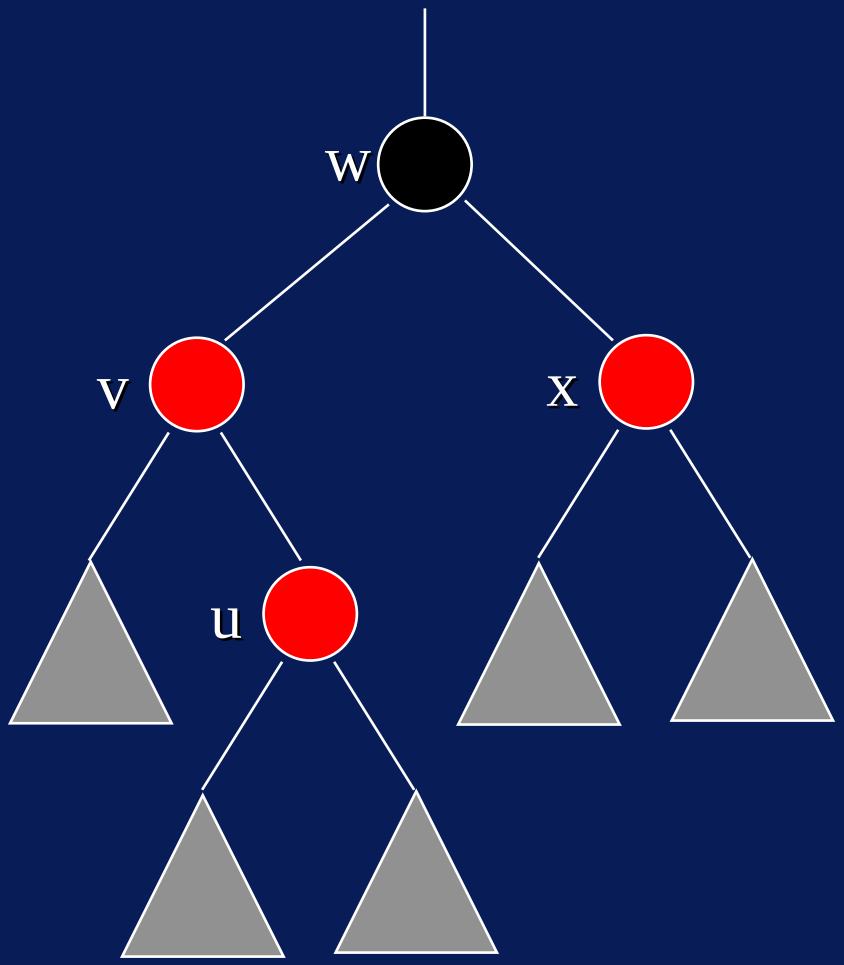
Red-Black Trees

- Case 3.3:
  - x is red
  - u is the right child of v
  - Colour v and x black
  - Colour w red

  - *Repeat the restructuring with u := w*

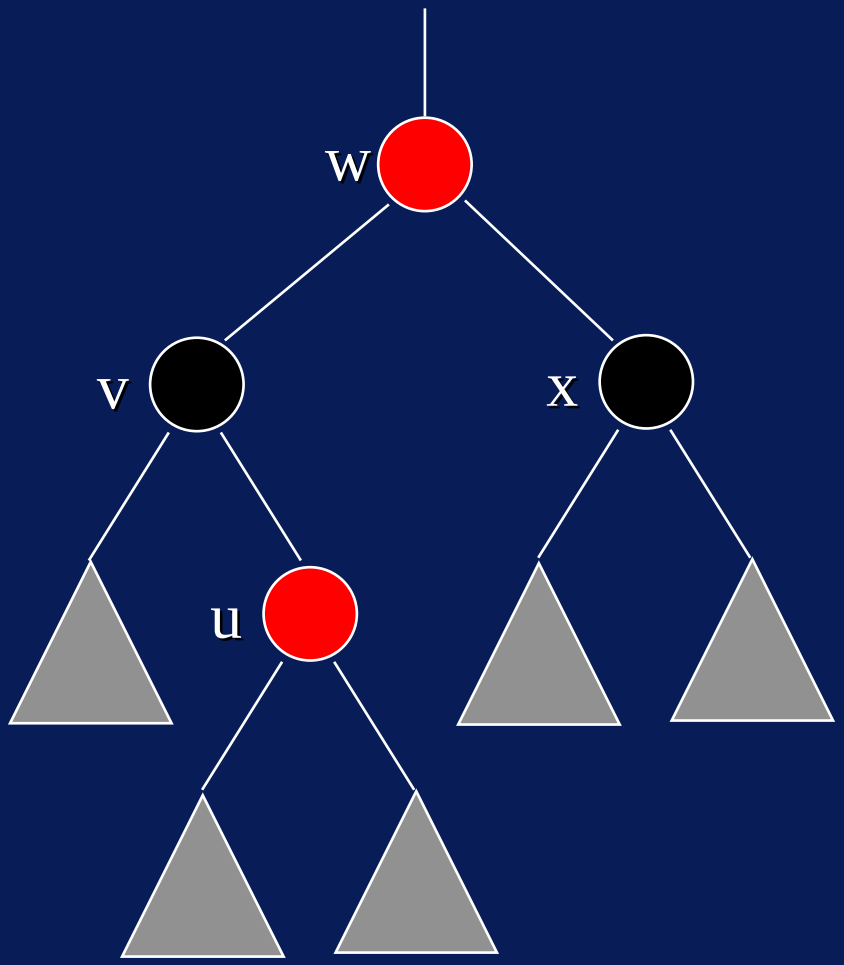
    (since the recolouring of w to red may cause a red violation)

Red-Black Trees



Recolour

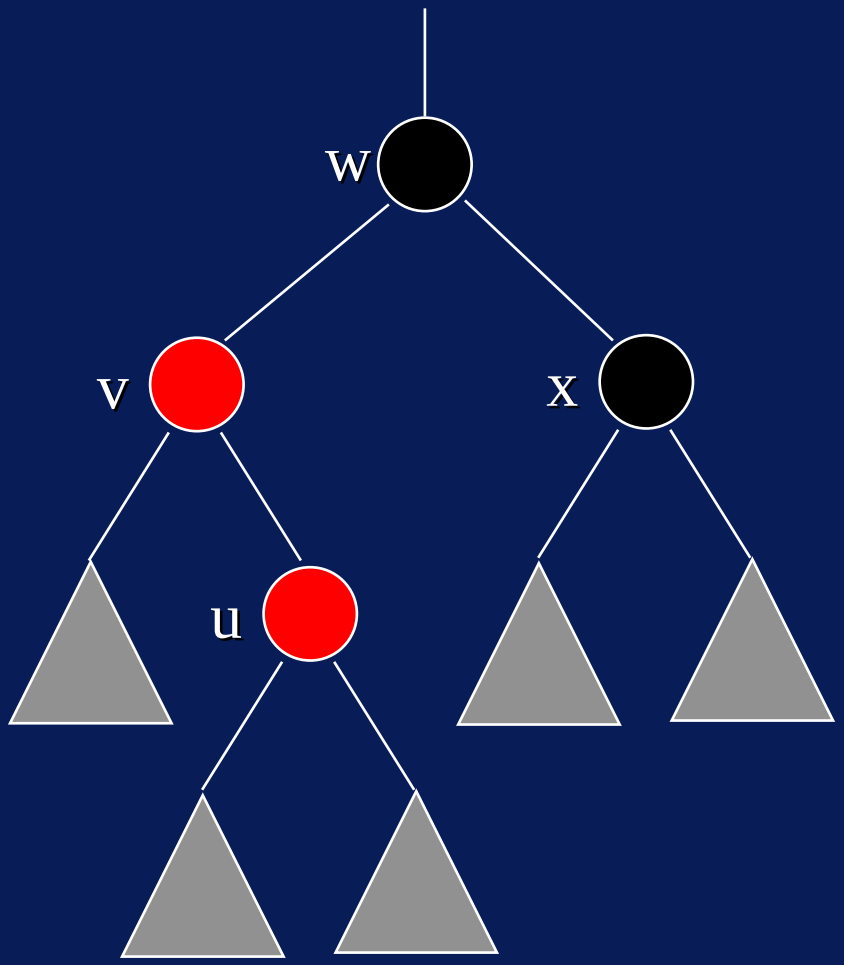
Red-Black Trees



Red-Black Trees

- Case 3.4:
  - x is black
  - u is the right child of v
  - Zig-zag promote u
  - Colour u black
  - Colour w red

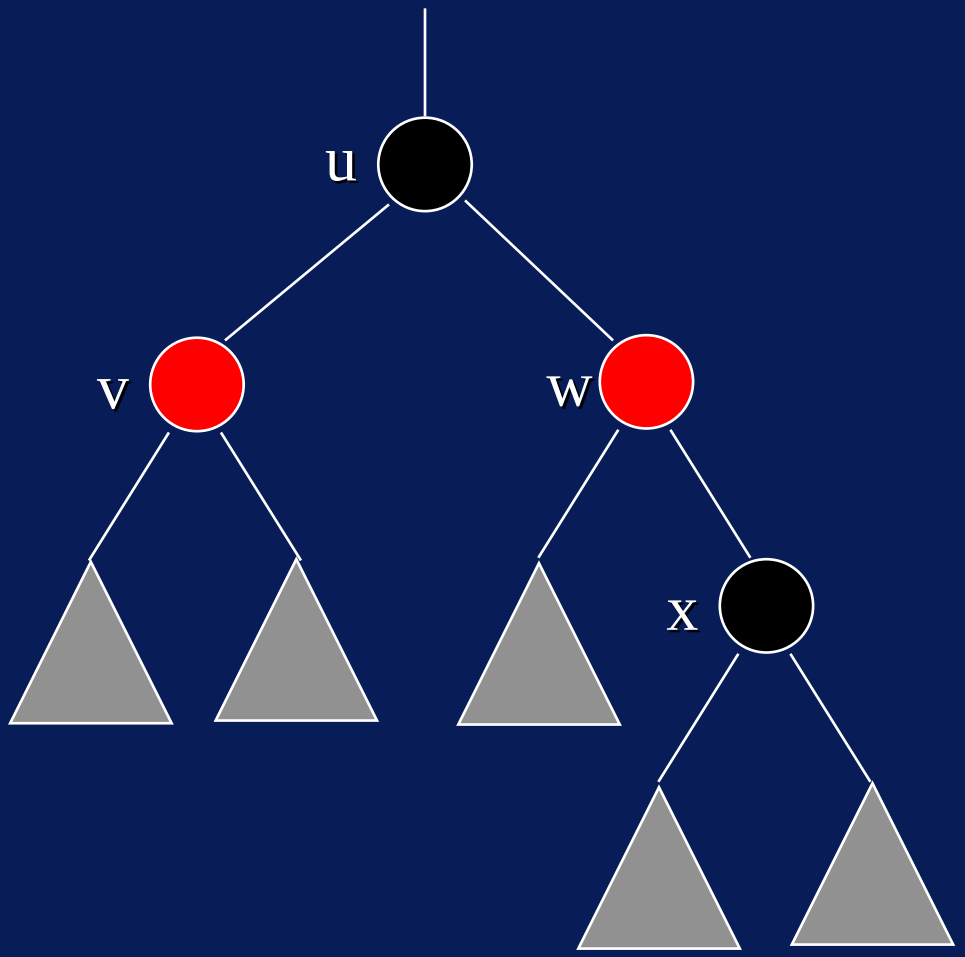
Red-Black Trees



Recolour & restructure

Zig-zag promote u;
colour u black;
colour w red

Red-Black Trees



Red-Black Trees

- Case 4:
  - u is not the root,
  - its parent v is not the root,
  - v is the right child of its parent w
  - (x is the left child of w, i.e. x is v's sibling)
- This case is symmetric to case 3.