Advanced Algorithms

Course Information
Review of the Basics

Professor Siu-Wing Cheng
HKUST

CSIT5500E (L1) — Advanced Algorithms

Instructor: Prof. Siu-Wing Cheng, scheng@cse.ust.hk

Lecture Time: Tuesday 19:30 - 20:20

Assessment:

- (30%) Written assignments
- (30%) In-class midterm
- (40%) Final examination

Midterm date: March 27, 2018

Lecture Notes:
Go to https://canvas.ust.hk/
Look for the files at the CSIT5500 home page.

Tentative list of topics (may be updated later):

- Basics, sorting, search tree.
- Divide and conquer.
- String matching and suffix array.
- Dyanmic programming.
- Graph search, minimum spanning tree, shortest path.
- Stable marriage.
- Maximum flow.
- Stable marriage.
- Streaming and sketching.

Algorithm

bsearch(x, A, i, j):

1. if $i > j$ then return -1
2. $m := \lfloor (i + j)/2 \rfloor$
3. If $A[m] = x$, then return m
4. If $A[m] > x$, then return bsearch$(x, A, i, m - 1)$
5. If $A[m] < x$, then return bsearch$(x, A, m + 1, j)$

A description in pseudocode.

Compute the middle index m. Compare the input key with the element $A[m]$. If they are equal, output m and return. If the input key is smaller, recurse on the left subarray delimited by the first element and $A[m-1]$. If the input key is greater, recurse on the right subarray delimited by $A[m+1]$ and the last element. The recursion stops when the input subarray is empty.

A recipe for performing a computation. In this class, we often refer to the single processor model with infinite memory. We assume that any number can be stored in a computer word, and the following arithmetic operations can be performed in constant time each: $\{+, -, \times, \div, \sqrt[k]{\cdot}, \lfloor \cdot \rfloor, \lceil \cdot \rceil$, trigonometric functions$\}$, where k is a constant.

Correctness Proof

A lot of algorithms compute straightforwardly according to the mathematical definitions of the object to be computed. We call this the brute-force method.

A smart algorithm may do things in a more obscure way. In that case, a rigorous correctness proof is required; otherwise, we are not sure what we are computing.

Complexity Analysis

We are interested in the efficiency of an algorithm in terms of its running time and its storage requirement.

However, it consumes time and resources to conduct experiments and take measurements. Thus, it is highly desirable to have an easier way to perform a first-cut analysis to eliminate the inefficient algorithms without any implementation.

A solution is to count the number of steps taken by the algorithm, assuming each step takes the same constant amount of time.

We are not even interested in the exact count because it is a first-cut comparison. We are only interested in how the running time grows when the input size grows. The algorithm whose running time grows slower is better.

The same consideration applies to the storage requirement of the algorithm.

Input Size

The input size indicates how large the input size. Since we assume that any number can be stored in a computer word and each arithmetic operation takes constant time, we take the number of items in the input as the input size (instead of bits).

By an item, we mean something that can be represented by a number and so it can be stored using one computer word.

Examples:

Sorting	Size of the list or array
Graph problems	Numbers of vertices and edges
Searching	Number of input keys

Asymptotics

Consider the following linear search algorithm:

1. Let $A[0..n-1]$ be an input array of numbers. Let q be an input number. We want to find the location of q in A. Initialize $loc := -1$.
2. For $i := 0$ to $n-1$, if $A[i] = q$, then $loc := i$.
3. Output *loc*

The steps to be executed in an iteration depends on the outcome of the checking of the if-condition.

So counting the number of steps exactly is cumbersome. The number of steps to be executed in an iteration is no more than some constant c, irrespective of the evaluation of if-condition.

Therefore, the total number of steps is proportional to the number of iterations, which is n. The running time or time complexity is proportional to the input size n, and we say that the running time or time complexity is $O(n)$.

We want to express the running time in the form $O(f(n))$, where $f(n)$ is a function in n. It means that there exists some constant $c > 0$ such that the growth of the running time as a function in n is not faster than $c \cdot f(n)$.

For example, the linear search executes at most cn steps for some constant $c \geq 1$. So the number of steps is not exactly n. But it is fine for us to say that the running time of linear search is $O(n)$.

There are usually two requirements on $f(n)$:

- $f(n)$ should be kept as simple as possible. For example, for linear search, there is no point in saying that the running time is $O(\sqrt{n^2})$ or $O(\frac{n^{3/2}}{n} \cdot \sqrt{n})$.
- Note that $f(n)$ is an upper bound on the growth rate of the running time. So it is technically correct to say that the running time of linear search is $O(n^2)$. Obviously, for comparison sake, this expression is not useful. Thus, we would like to make $f(n)$ as tight as possible.

Let A and B be two algorithms for the same task (e.g. searching). If their time complexities are $O(f(n))$ and $O(g(n))$, respectively, such that $f(n) < g(n)$, then we say that A is faster.

For example, if both are searching algorithms and $f(n) = \log_2 n$ and $g(n) = n$, then we say that A is faster, which is always true for a large enough n.

Let A and B be two algorithms for the same task (e.g. searching). If their time complexities are $O(f(n))$ and $O(g(n))$, respectively, such that $f(n) < g(n)$, then we say that A is faster.

For example, if both are searching algorithms and $f(n) = \log_2 n$ and $g(n) = n$, then we say that A is faster, which is always true for a large enough n.

n	$\log_2 n$
2	1
32	5
1,024	10
1,048,576	20
1,073,741,824	30

n^2	n
100	10
10,000	100
1,000,000	1,000
100,000,000	10,000

n^2	n
100	10
10,000	100
1,000,000	1,000
100,000,000	10,000

$n \log_2 n$	n
160	32
10,240	1,024
491,520	32,768
20,971,520	1,048,576

Since $\log_a n = \log_b n/(\log_b a)$ for any constants a and b, $\log_a n$ and $\log_b n$ have the same asymptotic growth rate. Thus, the convention is to write $\log n$ without indicating the base.

In addition to ignoring leading constant factors, lower order terms should also be ignored. For example, if $f(n) = 10n^2 + 1000n + 10000 \log_2 n$, we can simply write $f(n) = O(n^2)$.

We use $O(1)$ to denote a number that is at most some constant. For example, $O(1)$ time means constant time.

You may have some concern. If the number of steps taken by an algorithm A is $10n$ and another algorithm B is $1000n$. Then, algorithm A is superior, but both have $O(n)$ running time according to our discussion!

You may have some concern. If the number of steps taken by an algorithm A is $10n$ and another algorithm B is $1000n$. Then, algorithm A is superior, but both have $O(n)$ running time according to our discussion!

Right. Our first-cut comparison does not distinguish between A and B. In this case, implementation and experimentation may become necessary because the kind of steps taken by A and B may now matter a lot.

Assume a 2GHz computer

	Insertion sort n^2	Merge sort $n \log_2 n$
Macau (0.5M)	~2min	0.0048s

Assume a 2GHz computer

	Insertion sort n^2	Merge sort $n \log_2 n$
Macau (0.5M)	~2min	0.0048s
Hong Kong (8M)	~9hr	0.1s

Assume a 2GHz computer

	Insertion sort n^2	Merge sort $n \log_2 n$
Macau (0.5M)	~2min	0.0048s
Hong Kong (8M)	~9hr	0.1s
China (1357M)		

Assume a 2GHz computer

	Insertion sort n^2	Merge sort $n \log_2 n$
Macau (0.5M)	~2min	0.0048s
Hong Kong (8M)	~9hr	0.1s
China (1357M)	~29yr	~21s

Worst-Case Analysis

Suppose that we modify the previous linear search algorithm as follows:

1. Let $A[0..n-1]$ be an input array of numbers. Let q be an input number. We want to find the location of q in A.
2. For $i := 0$ to $n-1$, if $A[i] = q$, then output i and halt.
3. Output "Not found".

The number of steps in an iteration is still at most some constant. But the number of iterations is no longer always equal to n because the algorithm stops as soon as q is located.

If we are lucky, we may find q at $A[0]$. Only $O(1)$ steps are executed in this case. If q is not in the array A, we need n iterations and $O(n)$ steps are executed.

The running time depends on both the input size and the input content. Input content is much harder to work with. An easy way to work around it is to conduct a worst-case analysis.

That is, we are only interested in the running time of an algorithm under the worst possible input. The running time bound obtained is called the worst-case running time or worst-case time complexity of the algorithm. The advantage is that the running time is guaranteed not to exceed the worst-case bound.

Worst-case analysis sounds quite difficult for the following reasons:

- We need to come up with the worst possible input.
- We need to show that it is indeed the worst possible input.

Worst-case analysis sounds quite difficult for the following reasons:

- We need to come up with the worst possible input.
- We need to show that it is indeed the worst possible input.

Luckily, it usually suffices by just looking at the steps of the algorithm.

In the case of the modified linear search, since the for-loop will iterate at most n times, we know that the worst case running time is $O(1)$ times n, which is $O(n)$. Note that I do not need to explain how to force the algorithm to use n iterations.

Worst-Case Analysis of Binary Search

Let $T(n)$ denotes the worst-case running time of binary search on a sorted array of n numbers.

In each recursive call, we perform a constant number of steps and then make a recursive call on the left or right subarray of length at most $n/2$. Therefore,

$$T(n) \leq T(n/2) + O(1)$$

The boundary condition is step 1. When we are down to probing a subarray of only one number, either we find x or the algorithm will make a recursive call on an empty subarray and it will return immediately. Therefore, $T(1) = O(1)$.

$$\begin{aligned} T(n) &\leq T(n/2) + O(1) \\ &\leq T(n/4) + O(1) + O(1) \\ &\leq T(n/2^k) + k \cdot O(1) \end{aligned}$$

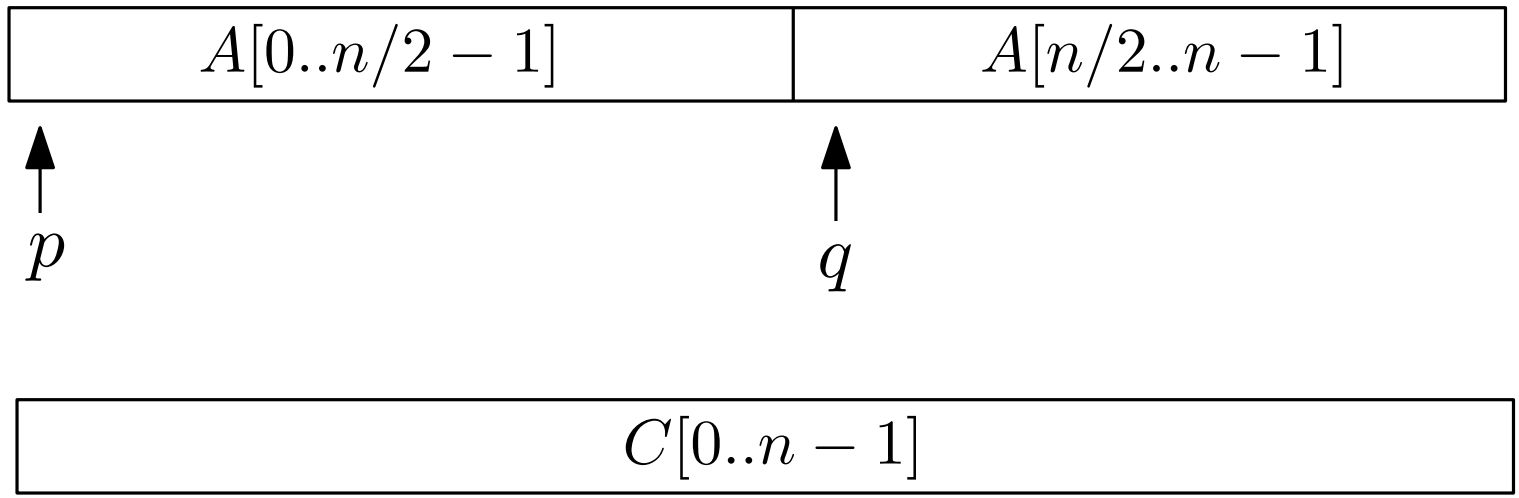
We are interested in $k = \log_2 n$ when we hit the boundary condition:

$$\begin{aligned} T(n) &\leq T(1) + \log_2 n \cdot O(1) \\ &= O(1) + O(\log n) \\ &= O(\log n) \end{aligned}$$

Mergesort

Suppose that you are given an unorderd array $A[0..n-1]$ of n numbers. Mergesort is supposed to return the array A in sorted order.

The idea is to apply divide-and-conquer. We first recursively sort $A[0..n/2-1]$ and $A[n/2..n]$ independently. This is done by two recursive calls of mergesort on these two subarrays. Afterwards, we merge these two sorted subarrays into one sorted array.



We maintain two indices p and q equal to 0 and $n/2$ initially, respectively. We also have a temporary array $C[0..n-1]$.

We compare $A[p]$ and $A[q]$. If $A[p] \leq A[q]$, we append $A[p]$ to C and increment p. If $A[q] < A[p]$, we append $A[q]$ to C and increment q. If $p = n/2$, we append the remaining entries $A[q..n-1]$ to C. If $q = n$, we append the remaining entries $A[p..n/2-1]$ to C.

Animation of mergesort: http://www.algoanim.ide.sk/

Worst-Case Analysis of Merge Sort

Let $T(n)$ denotes the worst-case running time of mergesort on an array of n numbers.

In each recursive call, we perform a constant number of steps and then make recursive calls on the left and right subarrays of length at most $n/2$.

However, unlike binary search, we need to merge two sorted subarrays after returning from the two recursive calls. So what is the worst-case running time of this merging step?

The merging step consists of two phases. In Phase 1, we copy elements from the two subarrays to C and increment p or q. In Phase 2, we copy the remaining elements in one of the two subarrays to C.

The merging step consists of two phases. In Phase 1, we copy elements from the two subarrays to C and increment p or q. In Phase 2, we copy the remaining elements in one of the two subarrays to C.

In Phase 1, an increment of p is accompanied by a constant number of operations for copying one element. The same holds for an increment of q. Therefore, it suffices to count the increments of p and q in Phase 1. Since neither p nor q is decremented in Phase 1, p can only be incremented $n/2$ times before reaching $n/2$. Similarly, q can only be incremented $n/2$ times before reaching n. And either p or q must be incremented. Therefore, there at most n increments of p and q in Phase 1, meaning that Phase 1 runs in $O(n)$ time.

In Phase 2, we copy the remaining entries in one of the subarrays to C. Since each subarray has $n/2$ entries to begin with, there are no more than $n/2$ entries to be copied in the end. So Phase 2 runs in $O(n)$ time.

Hence, the merging step runs in $O(n)$ time.

In Phase 2, we copy the remaining entries in one of the subarrays to C. Since each subarray has $n/2$ entries to begin with, there are no more than $n/2$ entries to be copied in the end. So Phase 2 runs in $O(n)$ time.

Hence, the merging step runs in $O(n)$ time.

We have the following recurrence:

$$T(n) \leq 2T(n/2) + O(n)$$

The boundary condition is $T(1) = O(1)$.

$$\begin{aligned} T(n) &\leq 2T(n/2) + O(n) \\ &\leq 2(2T(n/4) + O(n/2)) + O(n) \\ &\leq 4T(n/4) + O(n) + O(n) \\ &\leq 2^k T(n/2^k) + k \cdot O(n) \end{aligned}$$

We are interested in hitting the boundary condition when $k = \log_2 n$. Then:

$$\begin{aligned} T(n) &\leq 2^{\log_2 n} T(1) + \log_2 n \cdot O(n) \\ &\leq O(n) + O(n \log n) \\ &\leq O(n \log n) \end{aligned}$$

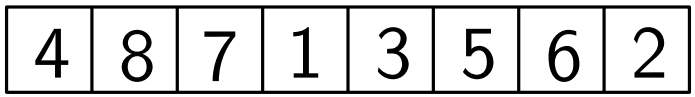
Quicksort

The idea is to use an element x from $A[0..n-1]$ to partition $A[0..n-1]$ into two parts $A[0..j]$ and $A[j+1, n-1]$ such that:

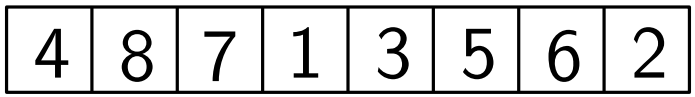
- Each element in $A[0..j]$ is at most x.
- Each element in $A[j+1..n-1]$ is at least x.

Afterwards, apply quicksort recursively to sort $A[0..j]$ and $A[j+1..n-1]$.

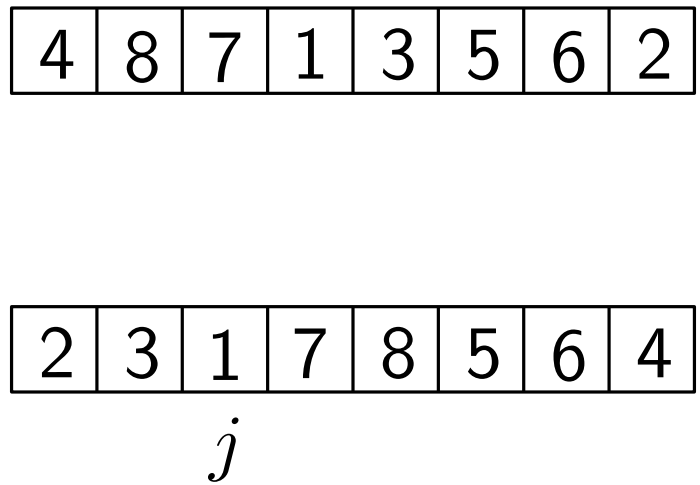
Nothing else to do after the two recursions return!


4 8 7 1 3 5 6 2

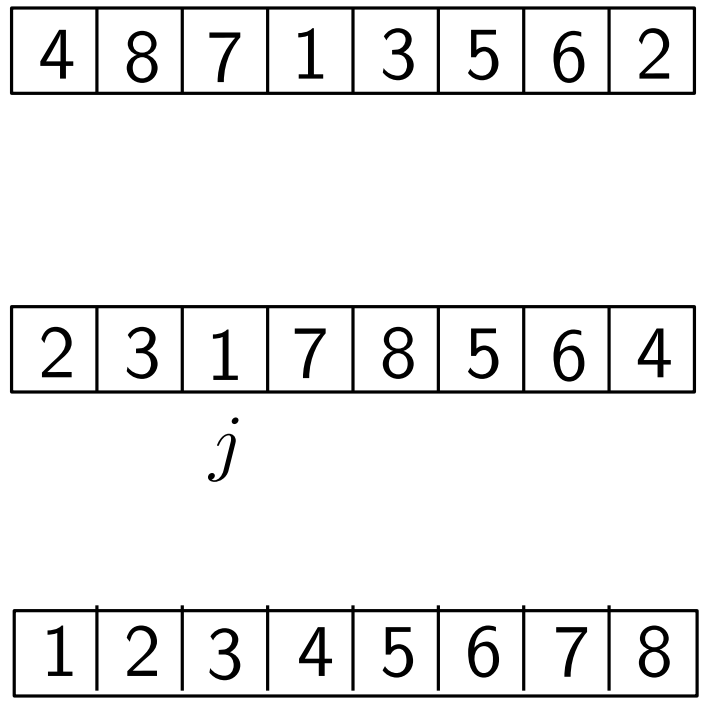
pivot $x = 4$



pivot $x = 4$



pivot $x = 4$



Quicksort(A, p, r) // Sort the subarray $A[p..r]$.

1. if $r \leq p$, then return.
2. Pick a pivot from $A[p..r]$. // usually a random choice
3. $q :=$ Partition(A, p, r)
4. Quicksort(A, p, q)
5. Quicksort$(A, q + 1, r)$

Quicksort(A, p, r) // Sort the subarray $A[p..r]$.

1. if $r \leq p$, then return.
2. Pick a pivot from $A[p..r]$. // usually a random choice
3. $q :=$ Partition(A, p, r)
4. Quicksort(A, p, q)
5. Quicksort$(A, q + 1, r)$

$A[p..r]$ is divided by Partition into two parts $A[p..q]$ and $A[q + 1..r]$.

Quicksort(A, p, r) // Sort the subarray $A[p..r]$.

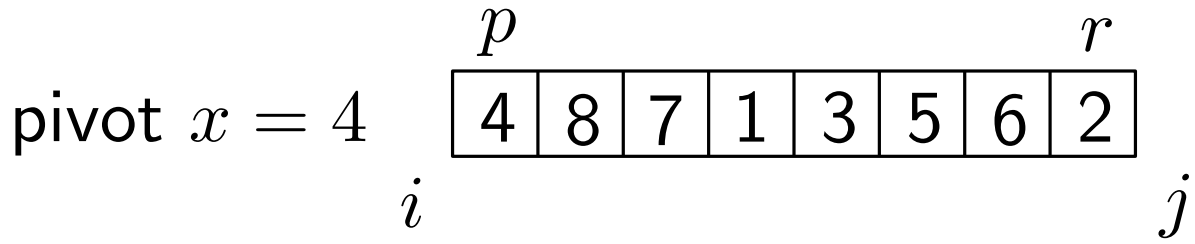
1. if $r \leq p$, then return.
2. Pick a pivot from $A[p..r]$. // usually a random choice
3. $q :=$ Partition(A, p, r)
4. Quicksort(A, p, q)
5. Quicksort$(A, q + 1, r)$

$A[p..r]$ is divided by Partition into two parts $A[p..q]$ and $A[q + 1..r]$.

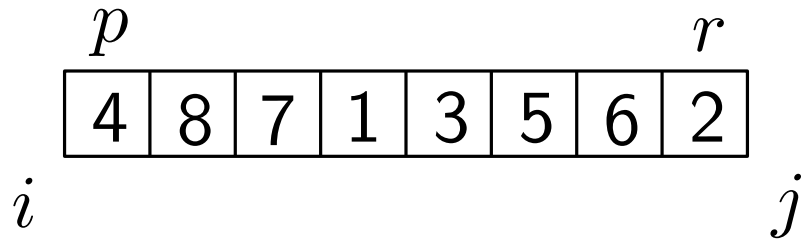
The first call is Quicksort$(A, 0, n - 1)$.

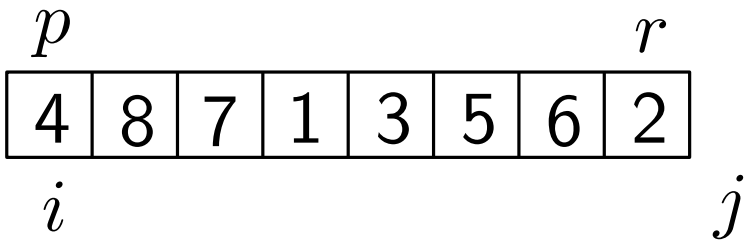
Partition(A, p, r).

1. Suppose that $i \in [p, r]$ is picked to be the pivot.
2. $x := A[i]$; exchange $A[p]$ and $A[i]$; $i := p - 1$; $j := r + 1$
3. Repeat the following:
    (a) Repeat $\{\ i := i + 1\ \}$ until $A[i] \geq x$
    (b) Repeat $\{\ j := j - 1\ \}$ until $A[j] \leq x$
    (c) If $i \geq j$, then return j
    (d) Swap $A[i]$ and $A[j]$

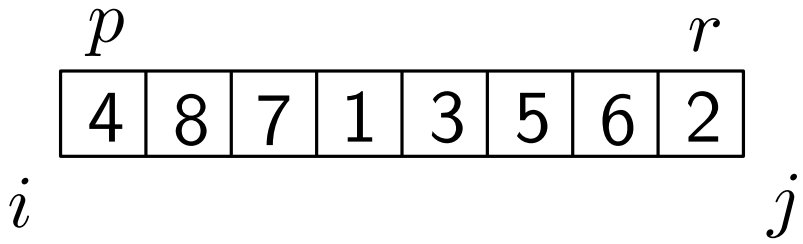

pivot $x = 4$
p
r
4
8
7
1
3
5
6
2
i
j

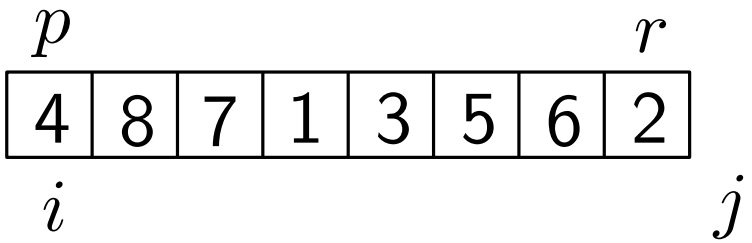
pivot $x = 4$

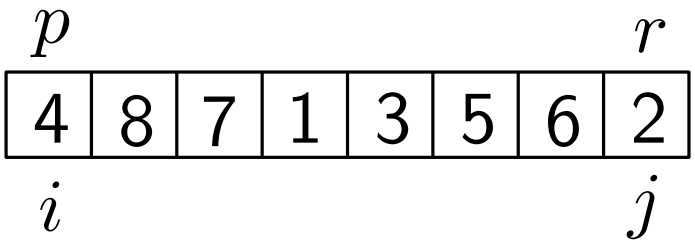




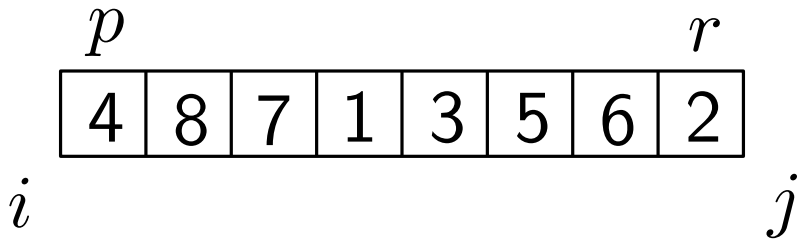
pivot $x = 4$

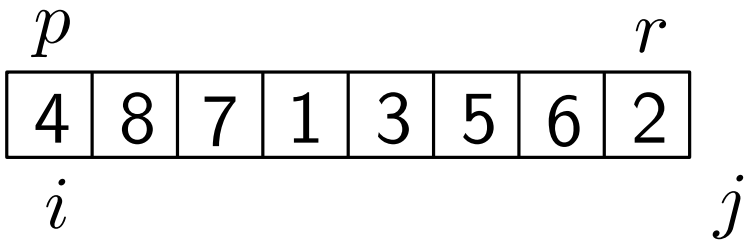


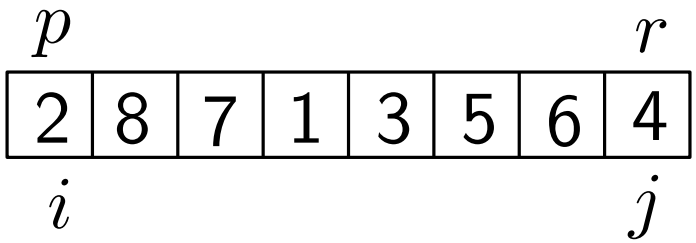




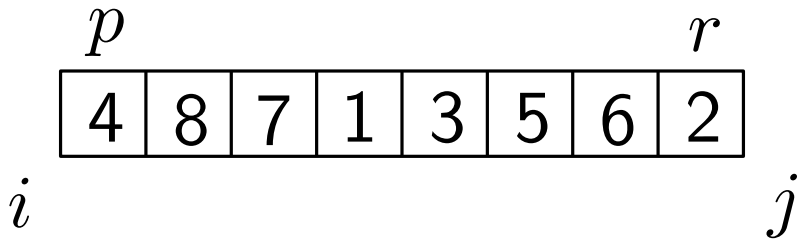
pivot $x = 4$

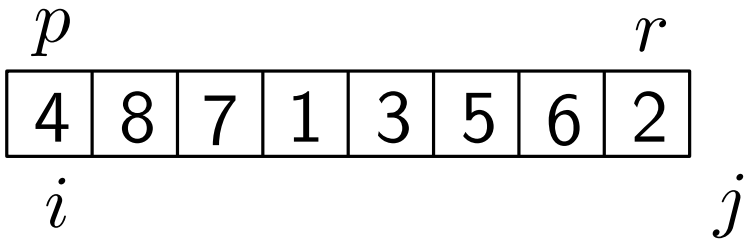


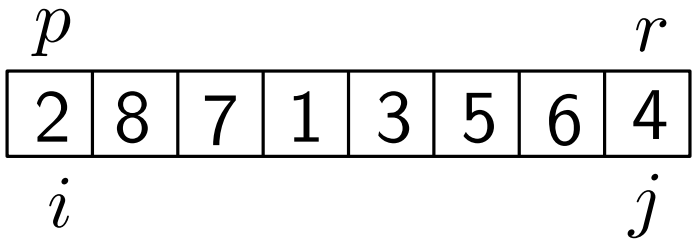


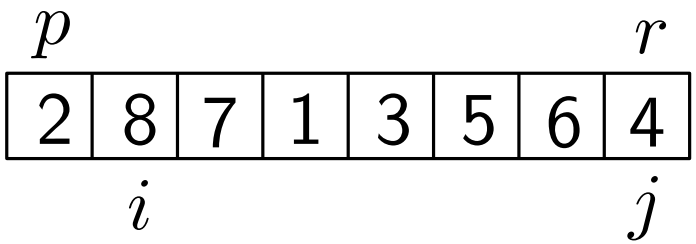


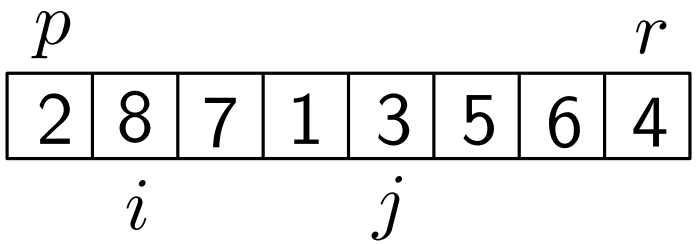
pivot $x = 4$


p
r
4 8 7 1 3 5 6 2
i
j

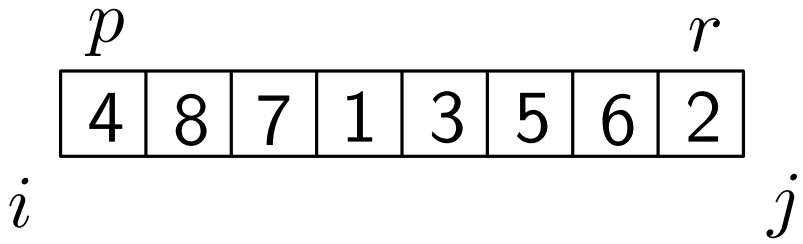

p
r
4 8 7 1 3 5 6 2
i
j

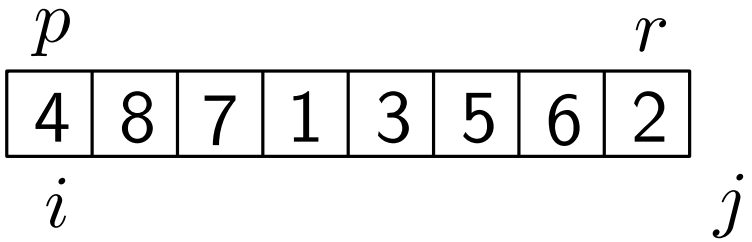

p
r
2 8 7 1 3 5 6 4
i
j

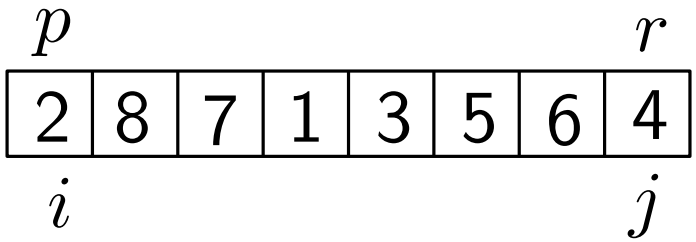

p
r
2 8 7 1 3 5 6 4
i
j

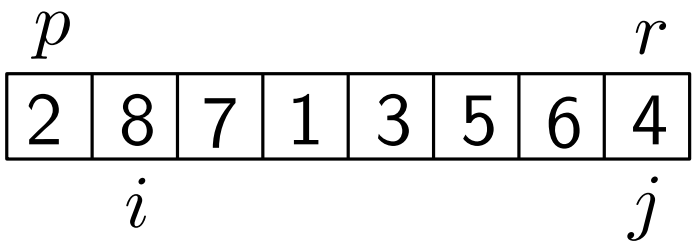

p
r
2 8 7 1 3 5 6 4
i
j

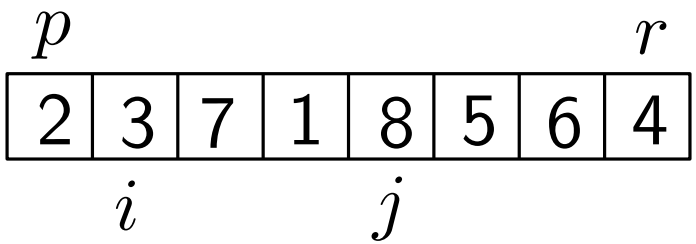
pivot $x = 4$


p
r
4 8 7 1 3 5 6 2
i
j

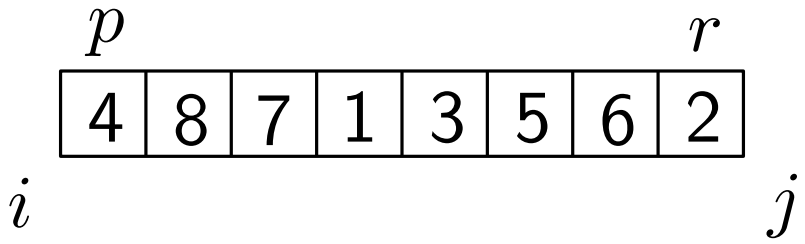

p
r
4 8 7 1 3 5 6 2
i
j

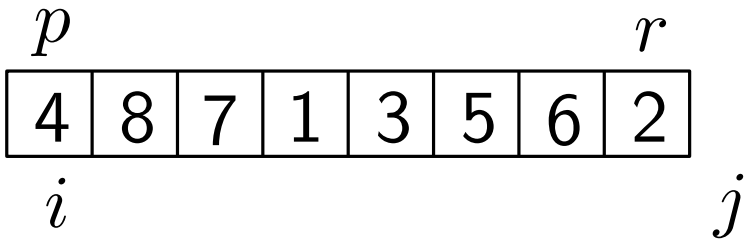

p
r
2 8 7 1 3 5 6 4
i
j

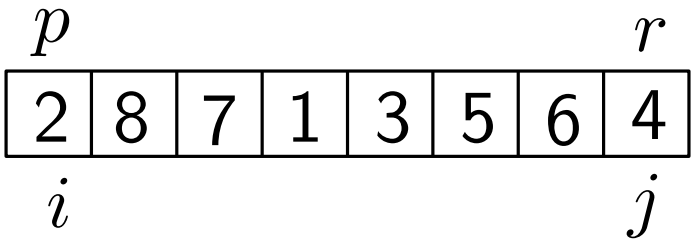

p
r
2 8 7 1 3 5 6 4
i
j

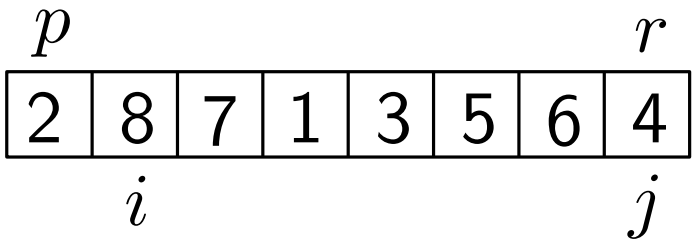

p
r
2 3 7 1 8 5 6 4
i
j

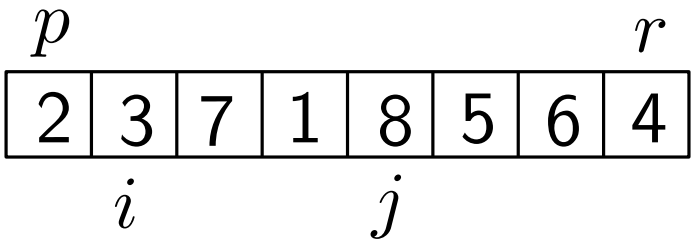
pivot $x = 4$

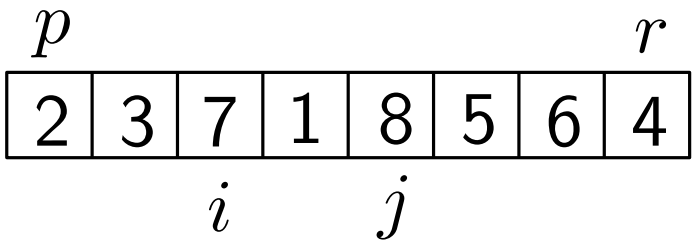

p
r
4 8 7 1 3 5 6 2
i
j


p
r
4 8 7 1 3 5 6 2
i
j

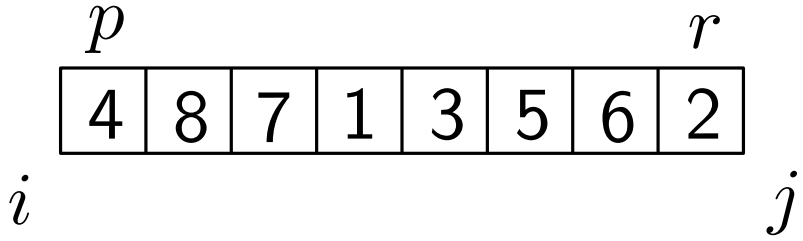

p
r
2 8 7 1 3 5 6 4
i
j

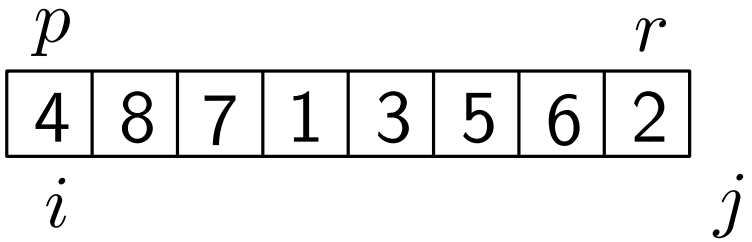

p
r
2 8 7 1 3 5 6 4
i
j

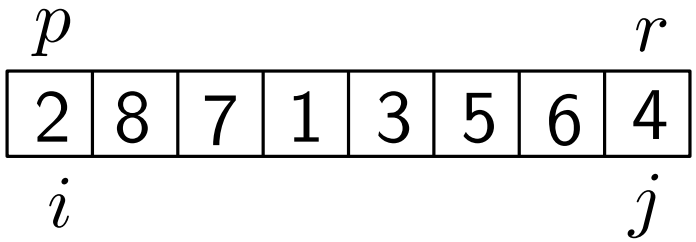

p
r
2 3 7 1 8 5 6 4
i
j

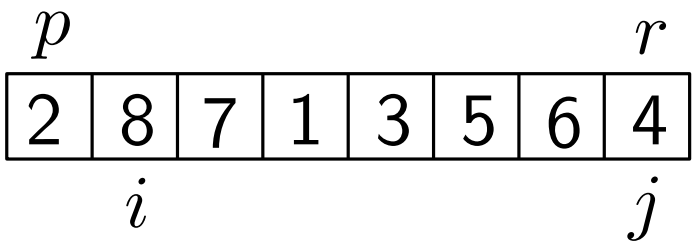

p
r
2 3 7 1 8 5 6 4
i
j

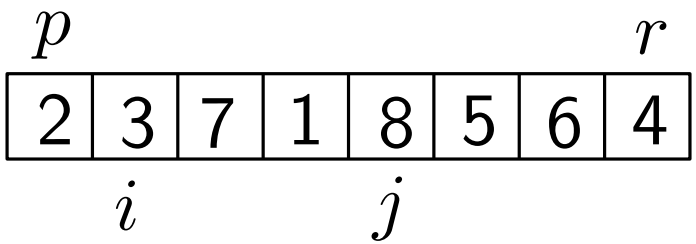
pivot $x = 4$

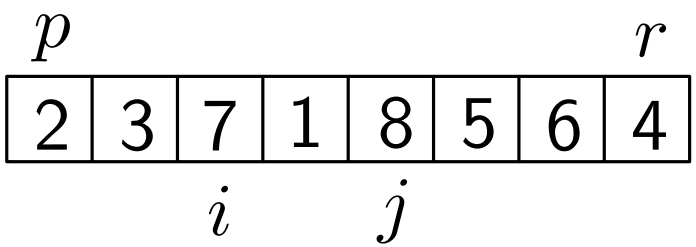

p
r
4 8 7 1 3 5 6 2
i
j

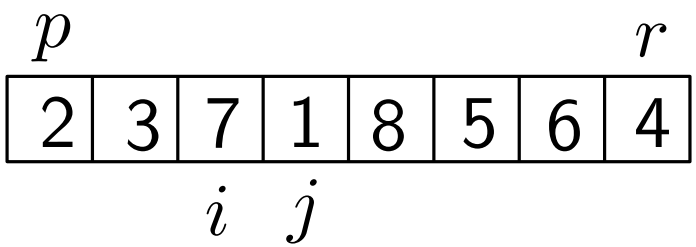

p
r
4 8 7 1 3 5 6 2
i
j


p
r
2 8 7 1 3 5 6 4
i
j

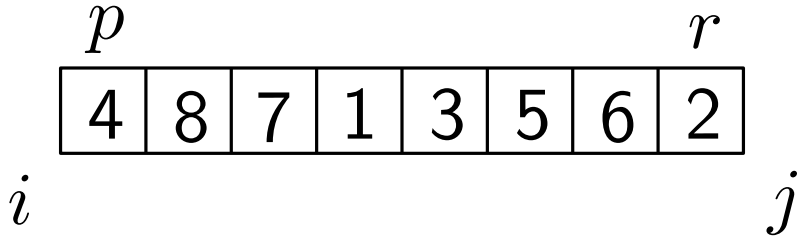

p
r
2 8 7 1 3 5 6 4
i
j

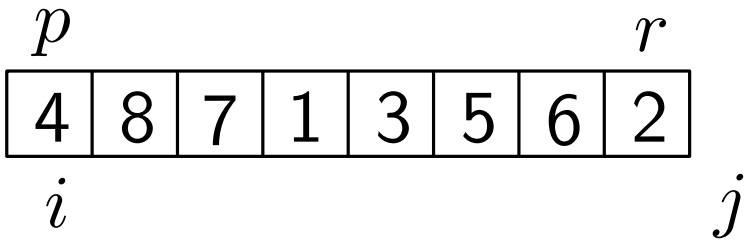

p
r
2 3 7 1 8 5 6 4
i
j

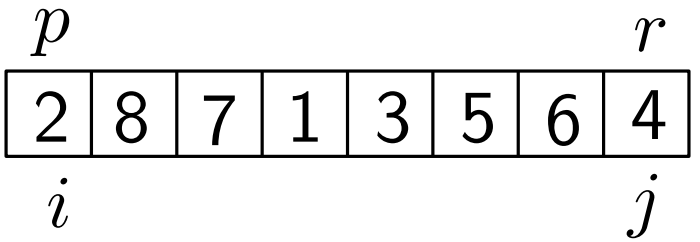

p
r
2 3 7 1 8 5 6 4
i
j

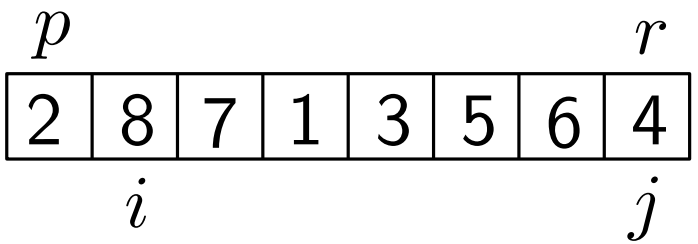

p
r
2 3 7 1 8 5 6 4
i
j

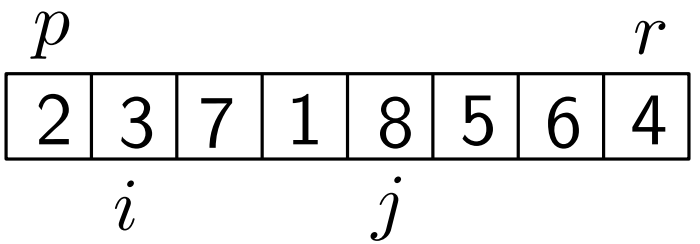
pivot $x = 4$

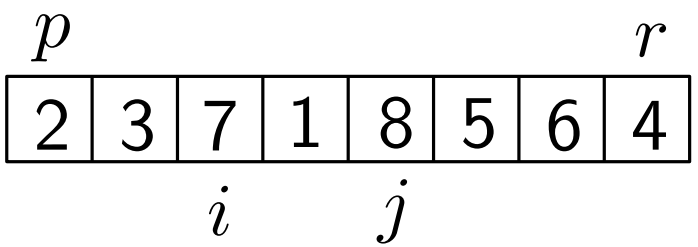

p
r
4 8 7 1 3 5 6 2
i
j

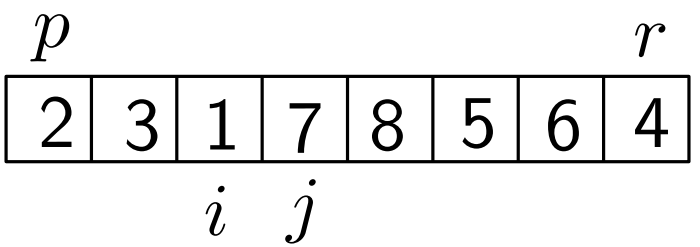

p
r
4 8 7 1 3 5 6 2
i
j


p
r
2 8 7 1 3 5 6 4
i
j

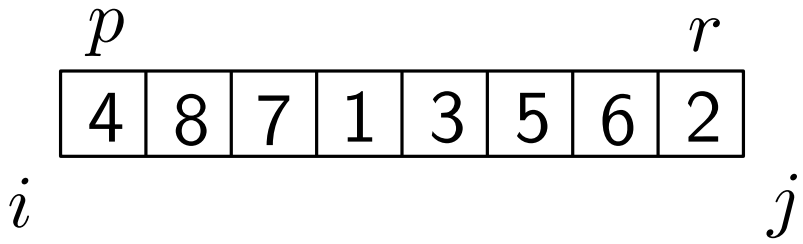

p
r
2 8 7 1 3 5 6 4
i
j

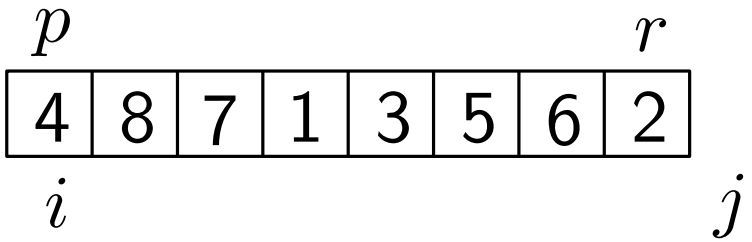

p
r
2 3 7 1 8 5 6 4
i
j

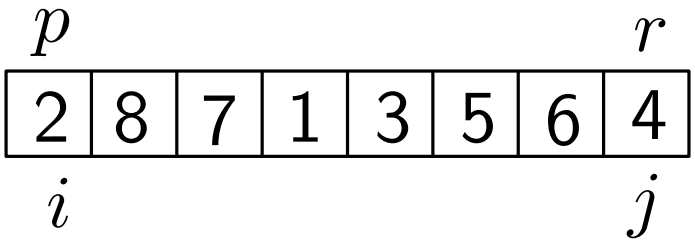

p
r
2 3 7 1 8 5 6 4
i
j

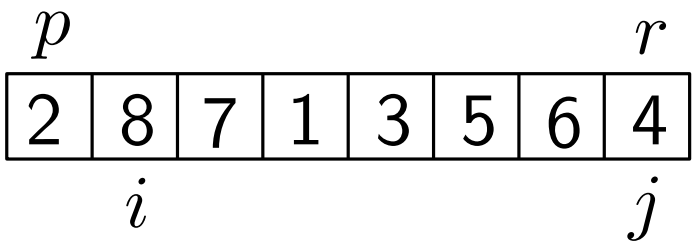

p
r
2 3 1 7 8 5 6 4
i
j

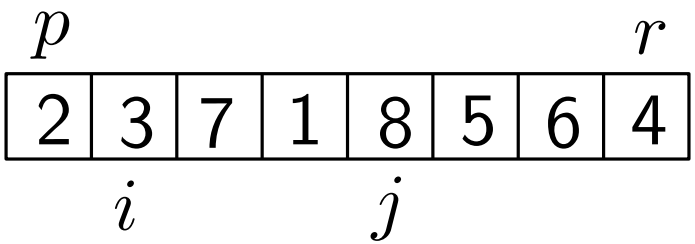
pivot $x = 4$

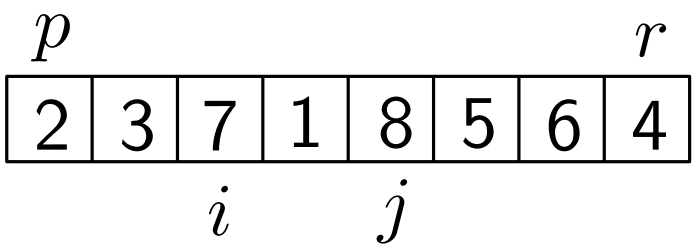

p
r
4 8 7 1 3 5 6 2
i
j

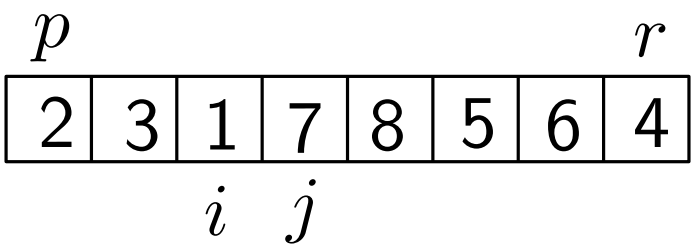

p
r
4 8 7 1 3 5 6 2
i
j

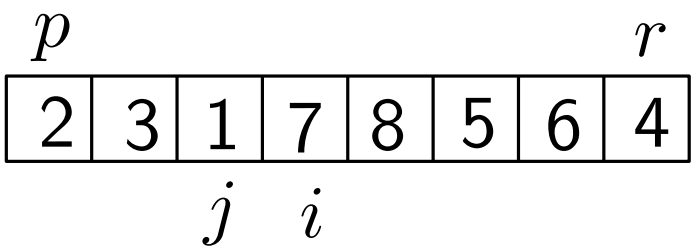

p
r
2 8 7 1 3 5 6 4
i
j


p
r
2 8 7 1 3 5 6 4
i
j


p
r
2 3 7 1 8 5 6 4
i
j


p
r
2 3 7 1 8 5 6 4
i
j


p
r
2 3 1 7 8 5 6 4
i
j


p
r
2 3 1 7 8 5 6 4
j
i

Worst-Case Analysis of Quicksort

Let $T(n)$ denotes the worst-case running time of quicksort on an array of n numbers.

In each recursive call, we call Partition and then make recursive calls on the left and right subarrays of length at most $n/2$.

Over all iterations of Partition, the indices i and j are always incremented and decremented respectively for at most $r - p$ times. That is, Partition runs in time linear in the length of the subarray $A[p..r]$.

However, the following recurrence does not hold in the worst case:

$$T(n) \leq 2T(n/2) + O(n)$$

If the pivot is chosen uniformly at random, it can be shown that the expected running time is $O(n \log n)$.

Heapsort

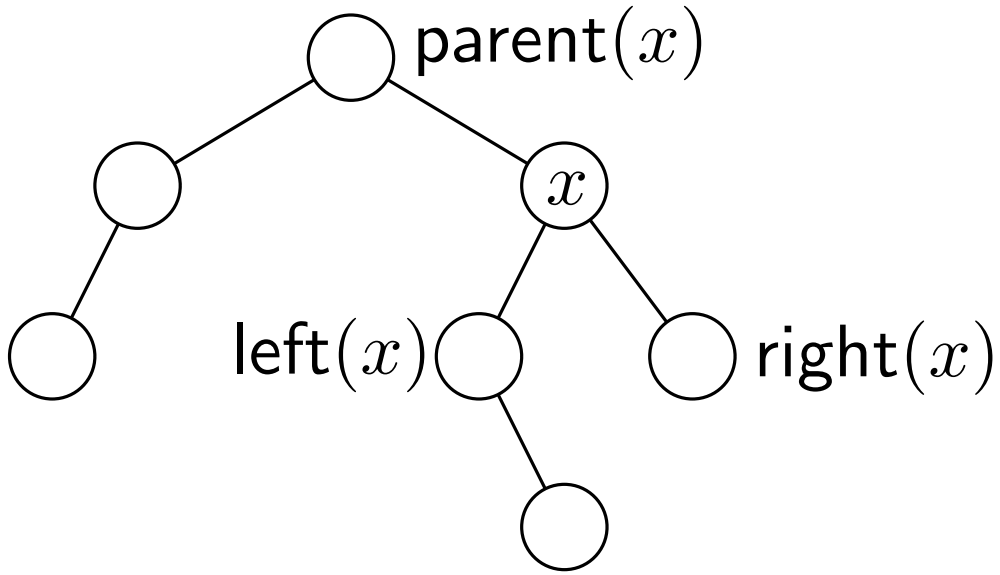
1. Select the current minimum from the remaining numbers.
2. Output the current minimum and remove it.
3. Repeat the above two steps.

A new data structure heap:

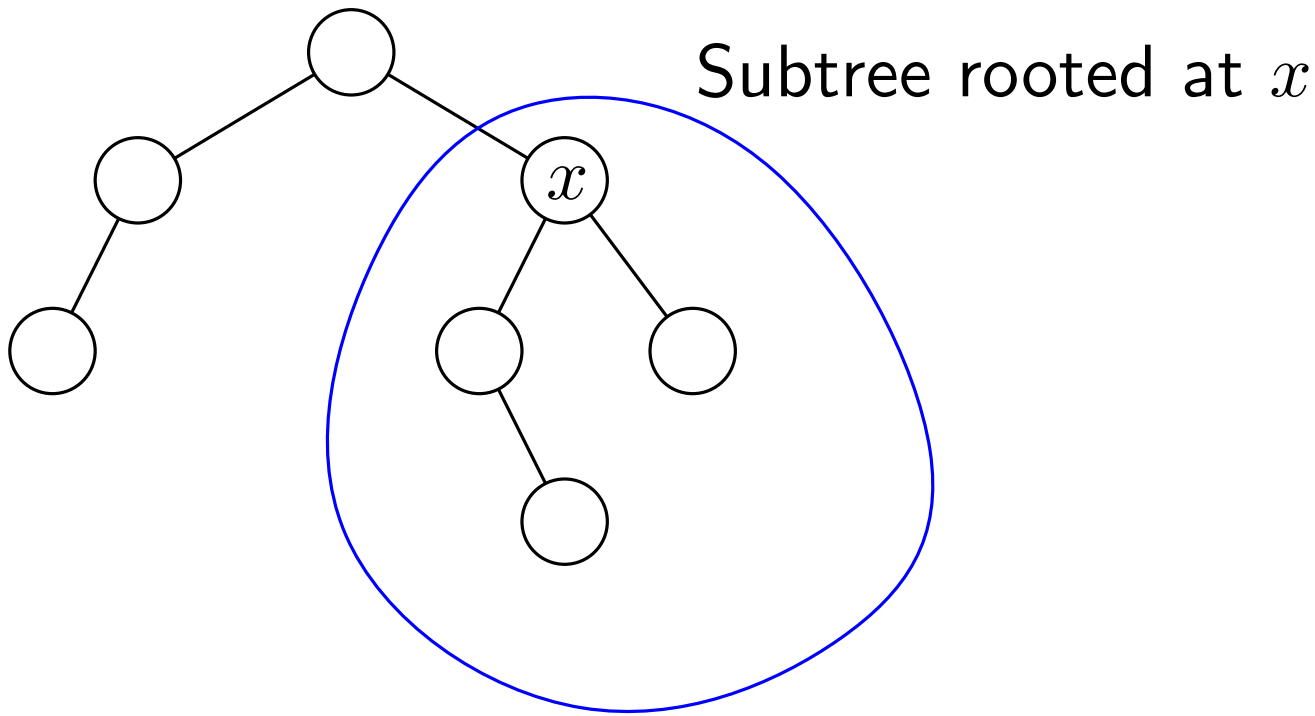
- Report the current minimum in $O(1)$ time.
- Delete the current minimum in $O(\log n)$ time.
- Insert a new number in $O(\log n)$ time.

Binary Tree

- A root at the topmost level.
- Each node has zero, one or two children.
- A node with no children is a leaf.
- For any node x, the parent, left and right children of x are denoted by parent(x), left(x) and right(x), respectively.

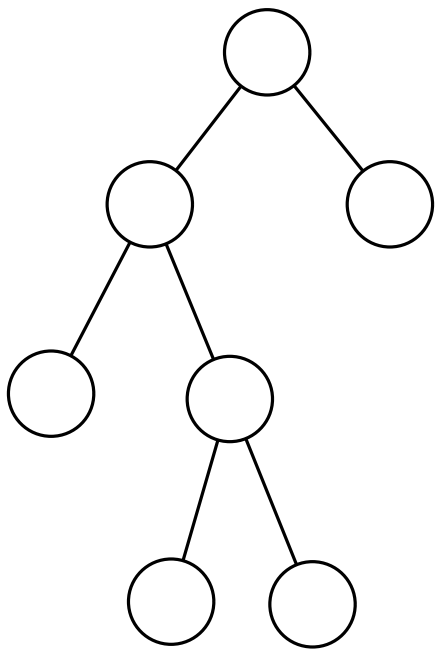


- A non-leaf node is an internal node.
- The children of a node are siblings.
- The subtree rooted at a node x consists of x and the descendants of x.
- The tree height is the number of edges on the longest root-to-leaf path.

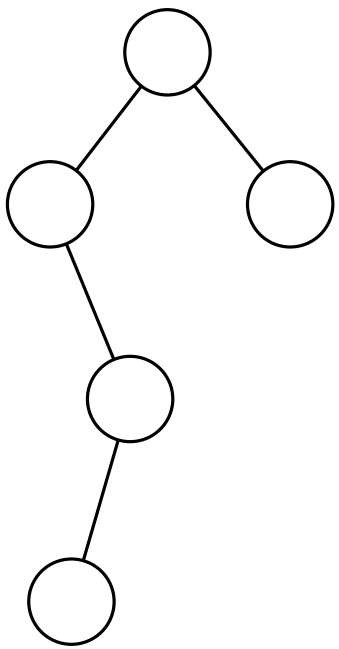


Special Binary Trees

Full binary tree: every internal node has exactly two children.

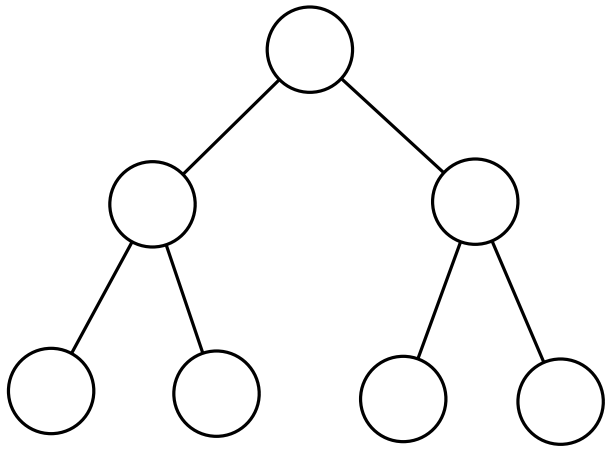


YES

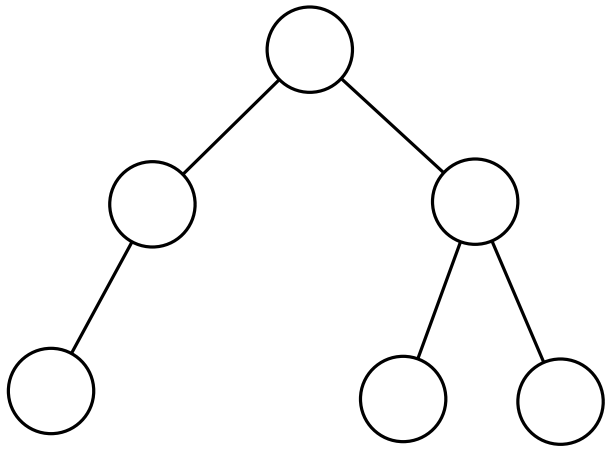


NO

Perfect binary tree: Evey level is full.

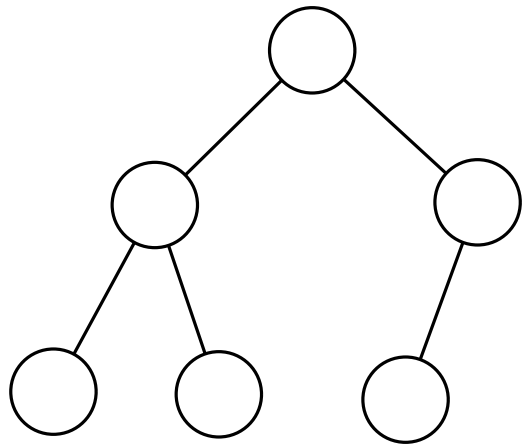


YES

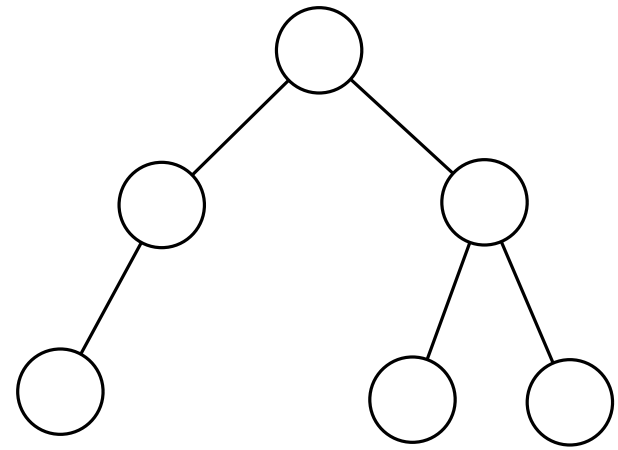


NO

Complete binary tree: Every level is full except possibly the bottommost level. If the bottommost level is not full, all nodes there are packed to the left.



YES

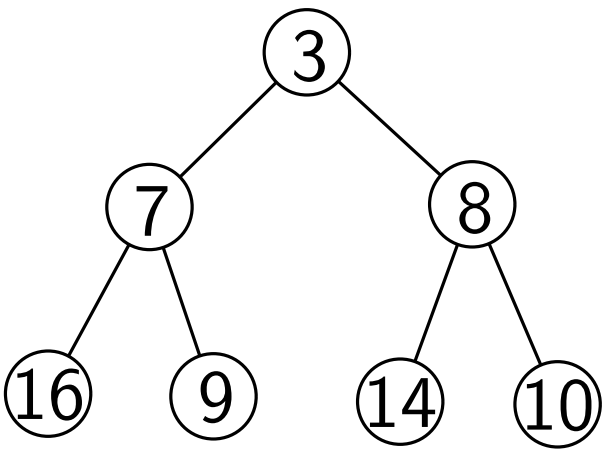


NO

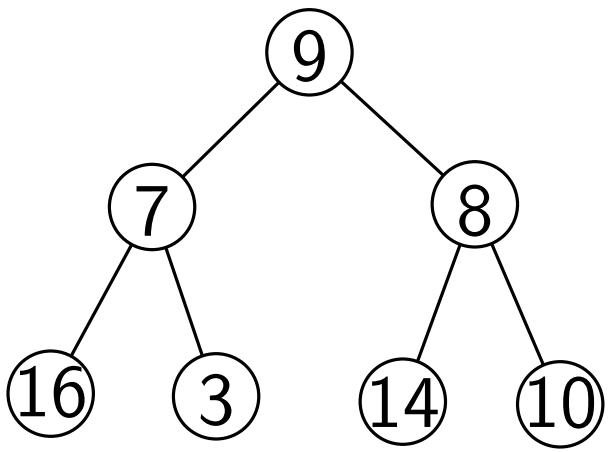
Claim: A complete binary tree of n nodes has $O(\log n)$ height.

Heap again

- A complete binary tree. Values are stored at the nodes.
- Min-heap property: value at a node is no more than the values at its children.



YES



NO

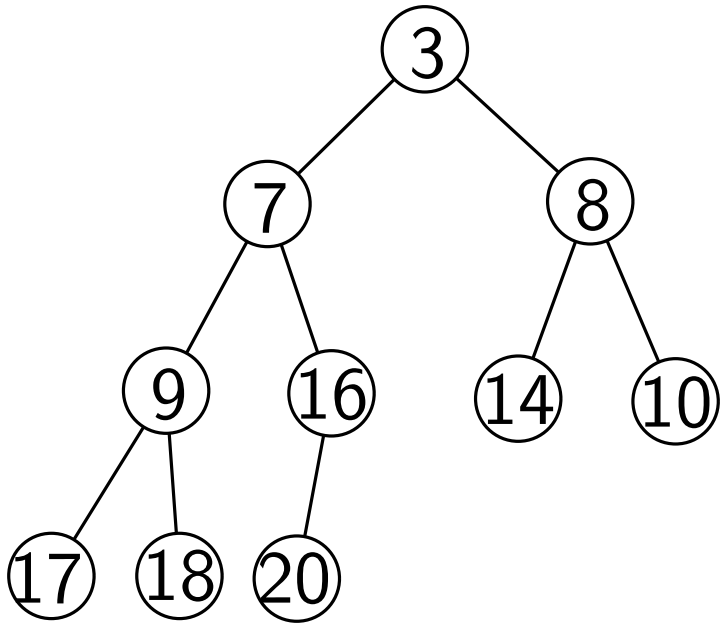
Minimum is stored at the root. Can be accessed in $O(1)$ time.

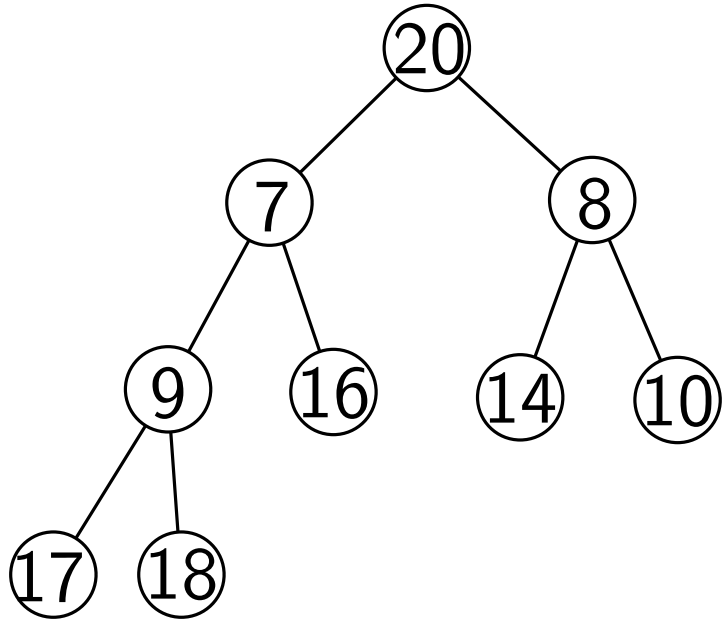
Delete the minimum:

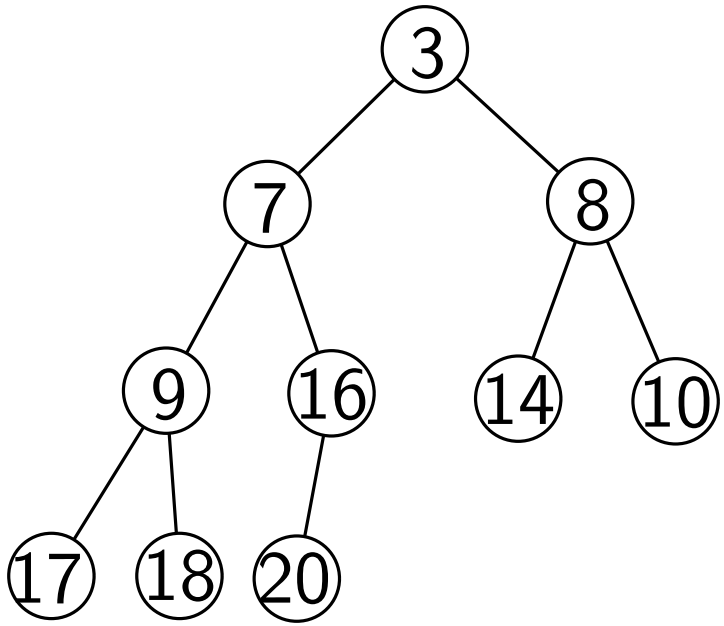
1. Copy the last value to the root.
2. Restore the min-heap property.

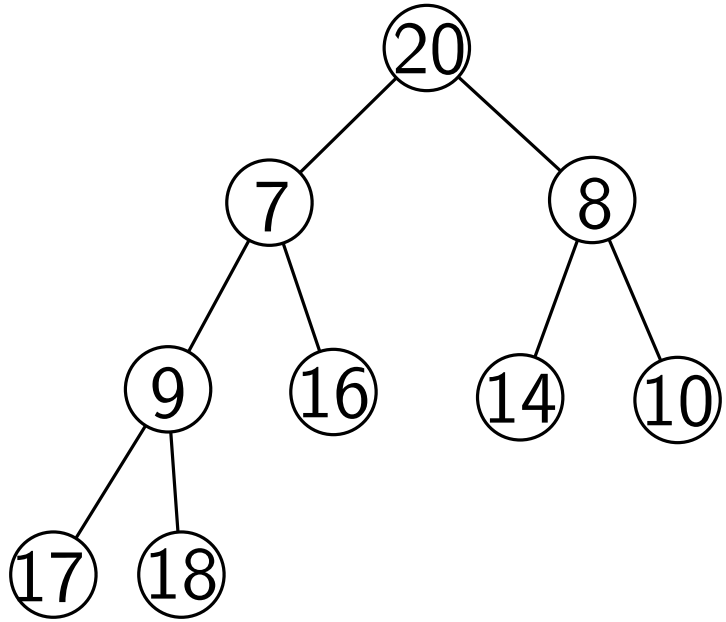
Insert a new value:

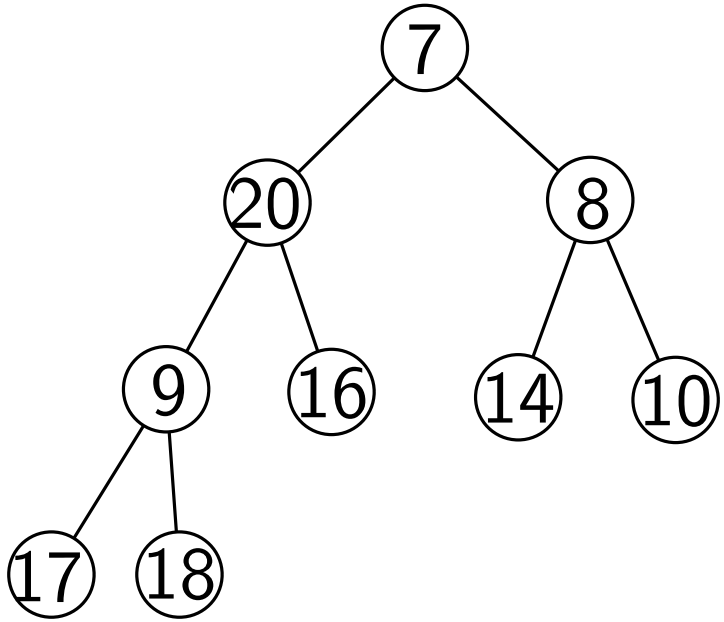
1. Add the new value to the bottommost level, or if that level is full, create a new level.
2. Retore the min-heap property.

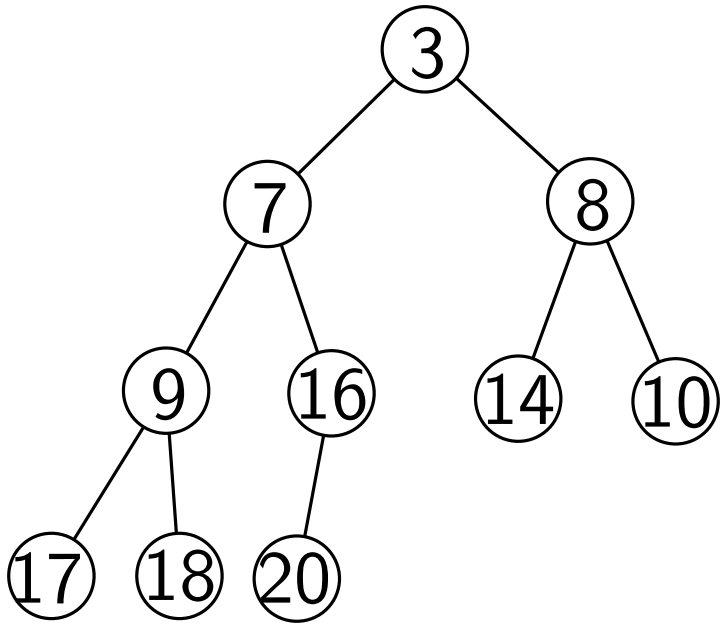

3
7
8
9
16
14
10
17
18
20

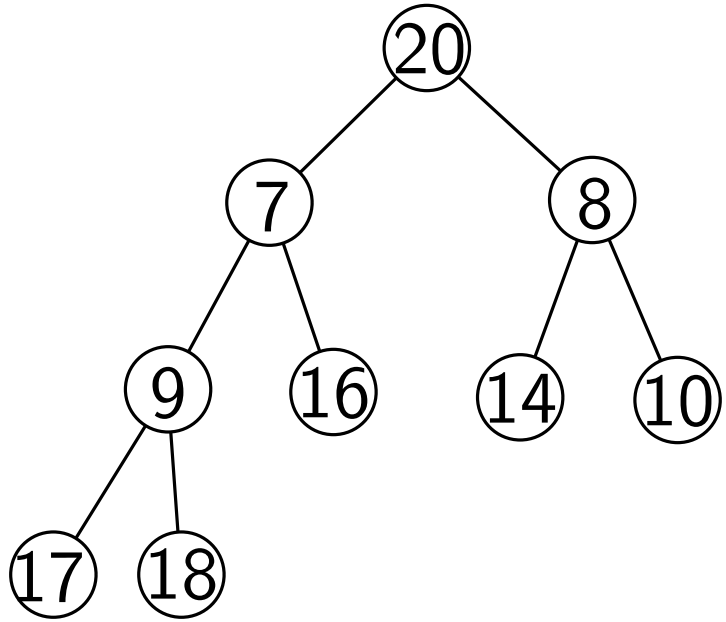

20
7
8
9
16
14
10
17
18

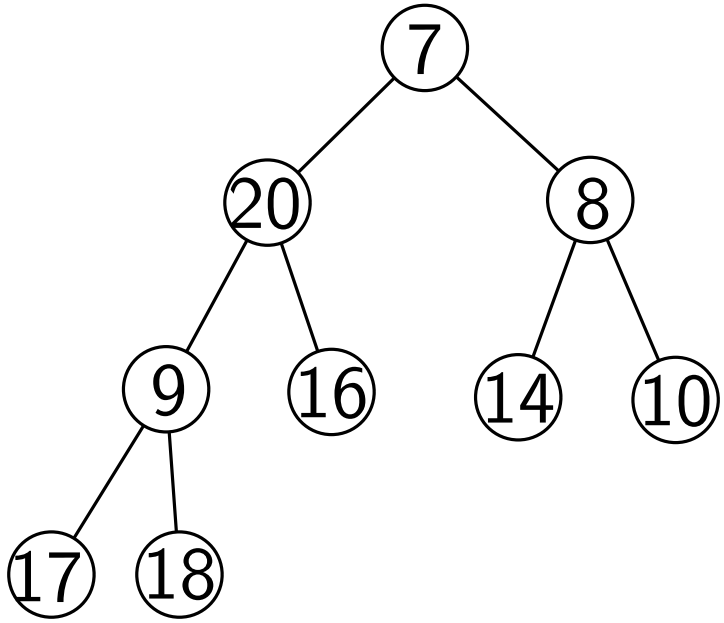

3
7
8
9
16
14
10
17
18
20

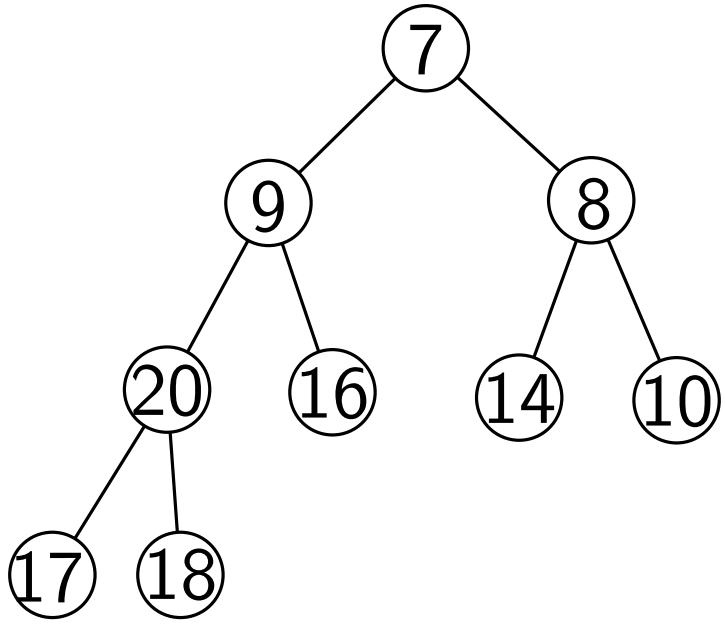

20
7
8
9
16
14
10
17
18

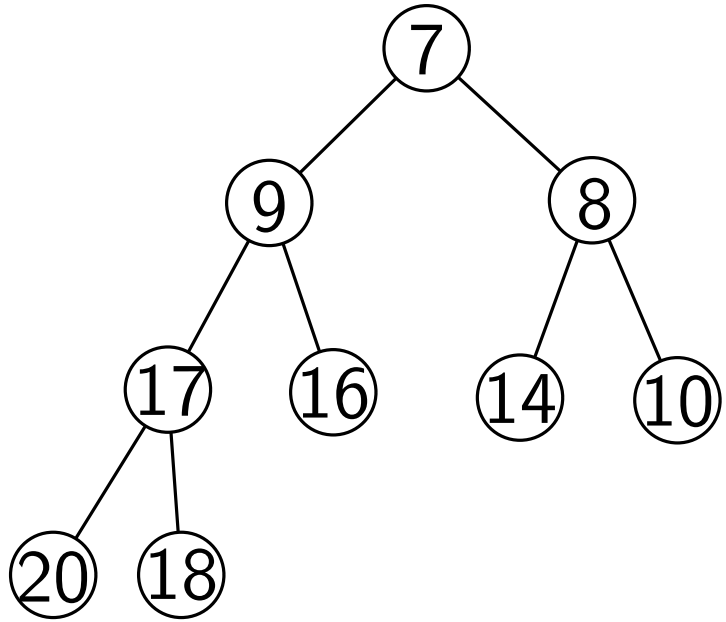

7
20
8
9
16
14
10
17
18

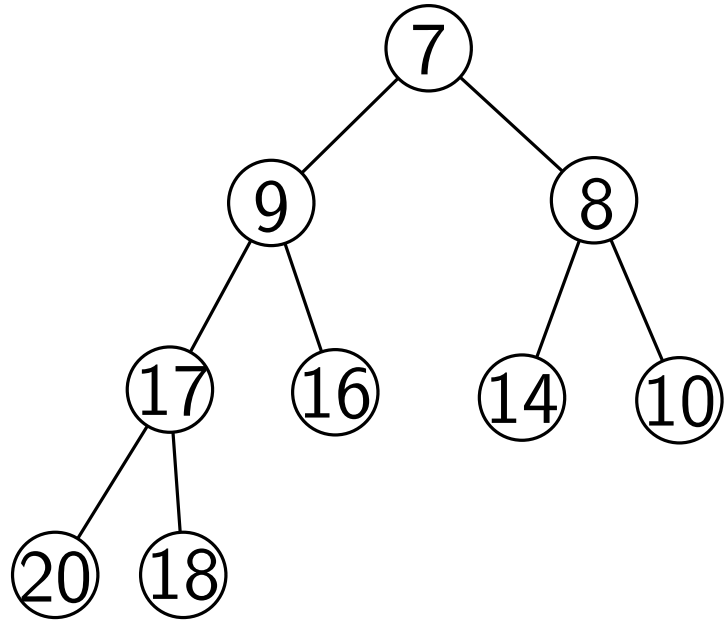

3
7
8
9
16
14
10
17
18
20


20
7
8
9
16
14
10
17
18

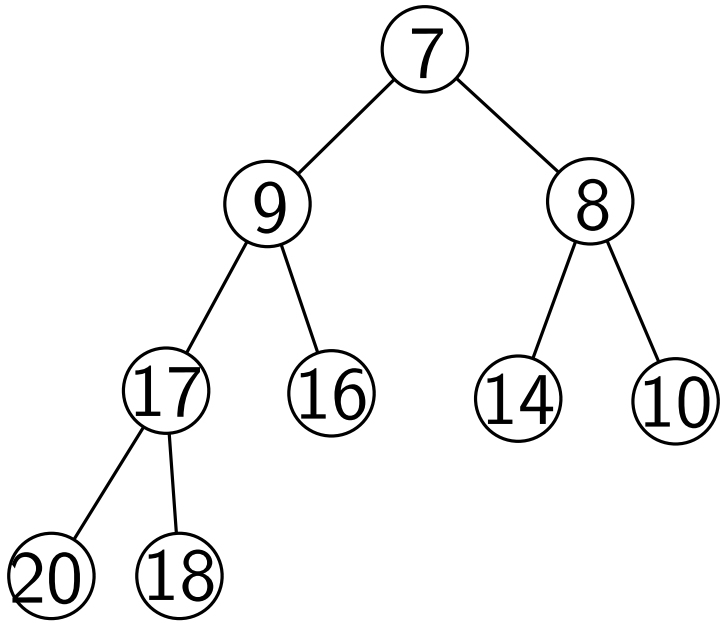

7
20
8
9
16
14
10
17
18

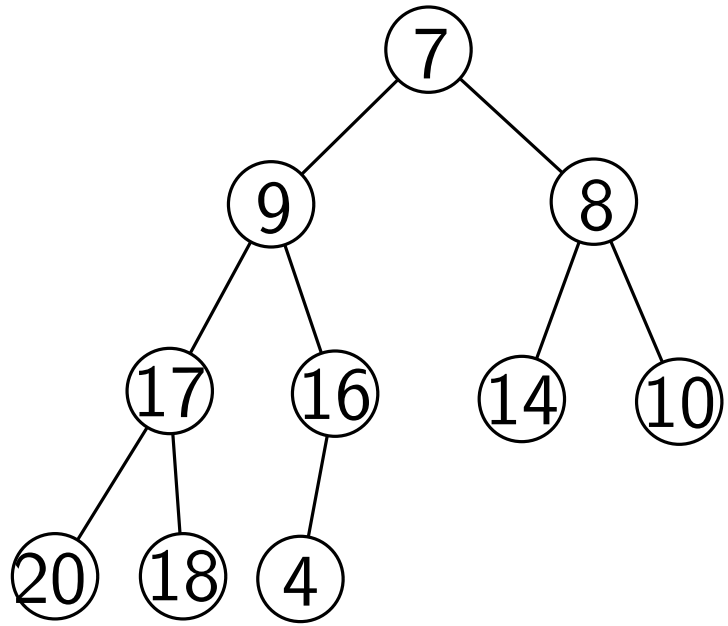

7
9
8
20
16
14
10
17
18

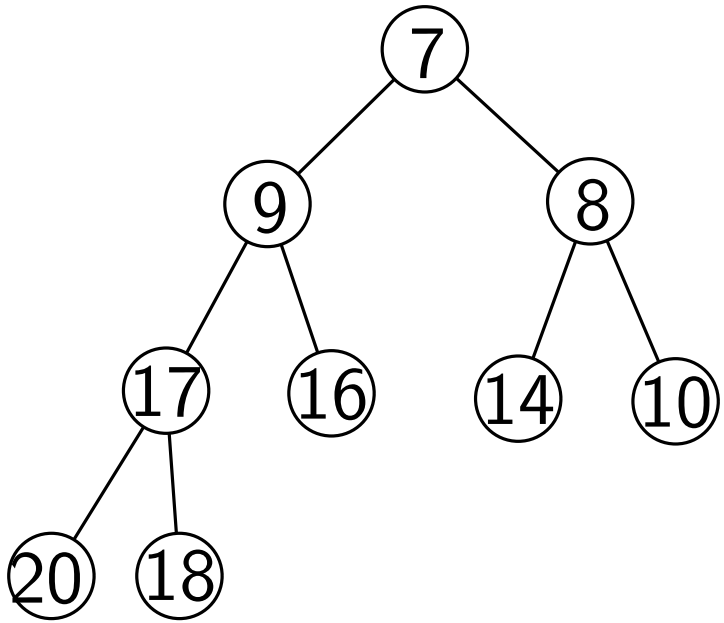

7
9
8
17
16
14
10
20
18

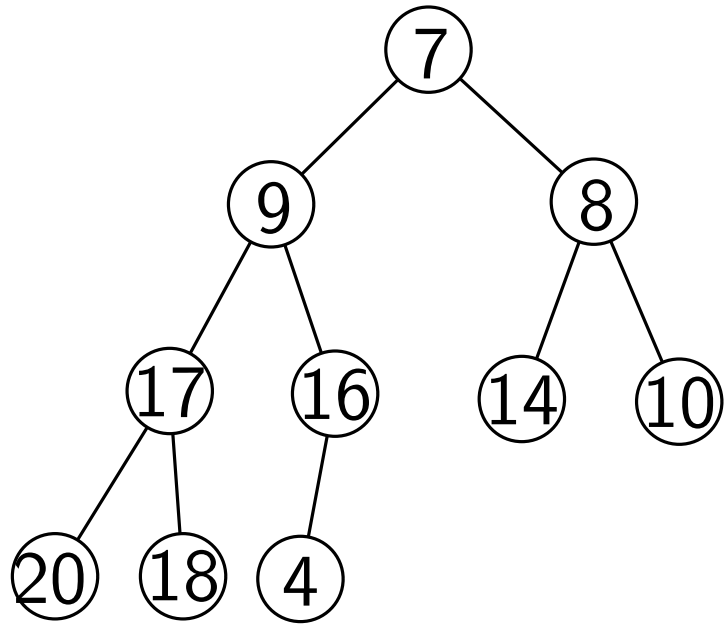


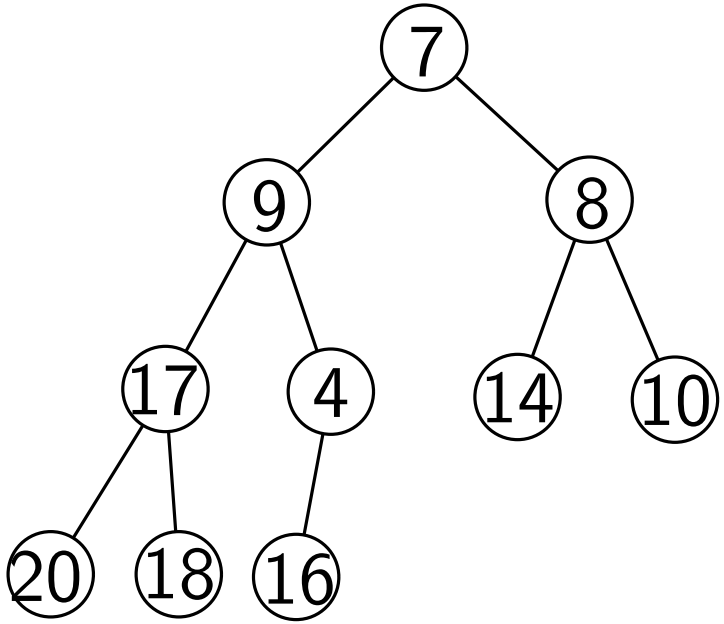
Running time is $O(\text{height}) = O(\log n)$.

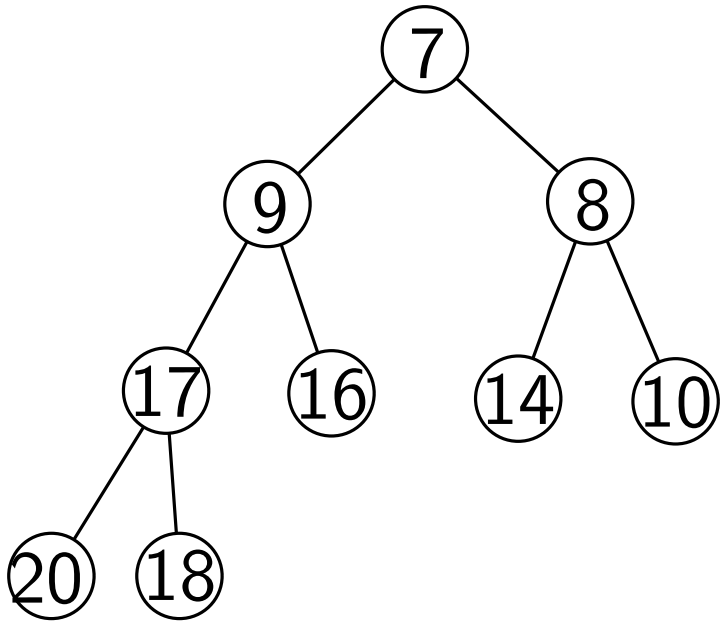

7
9
8
17
16
14
10
20
18

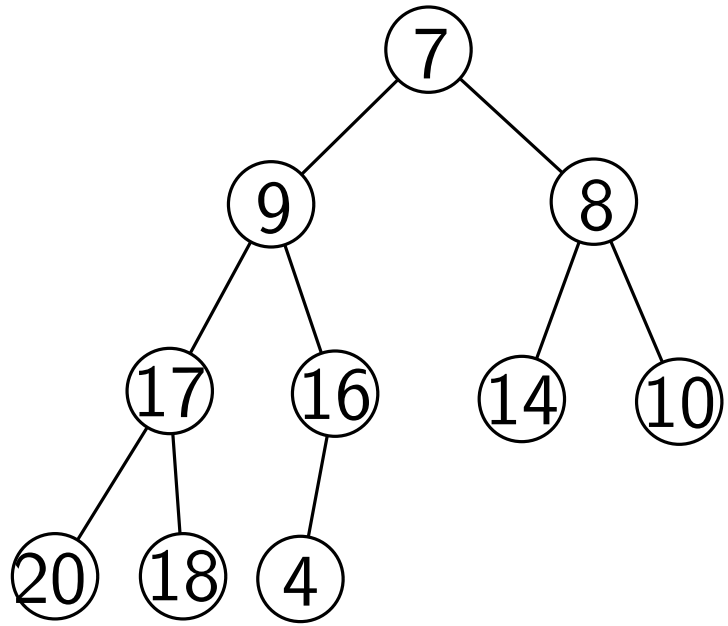

7
9
8
17
16
14
10
20
18
4

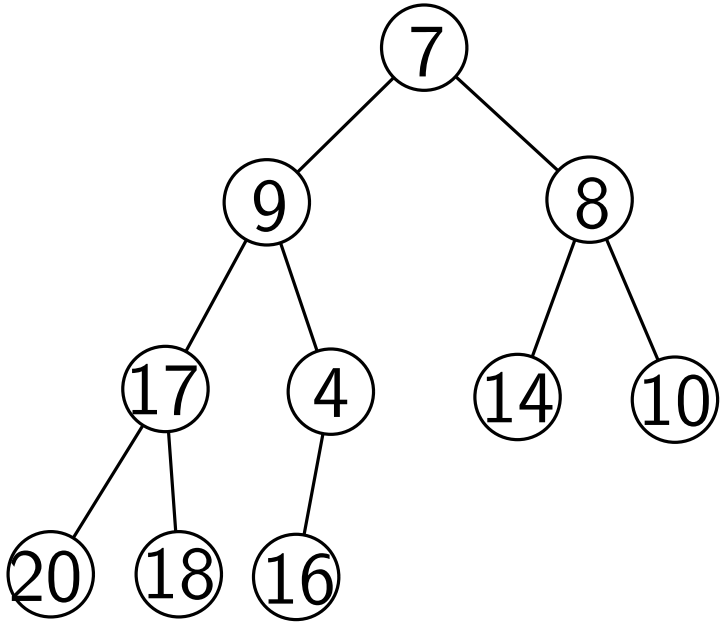

7
9
8
17
16
14
10
20
18

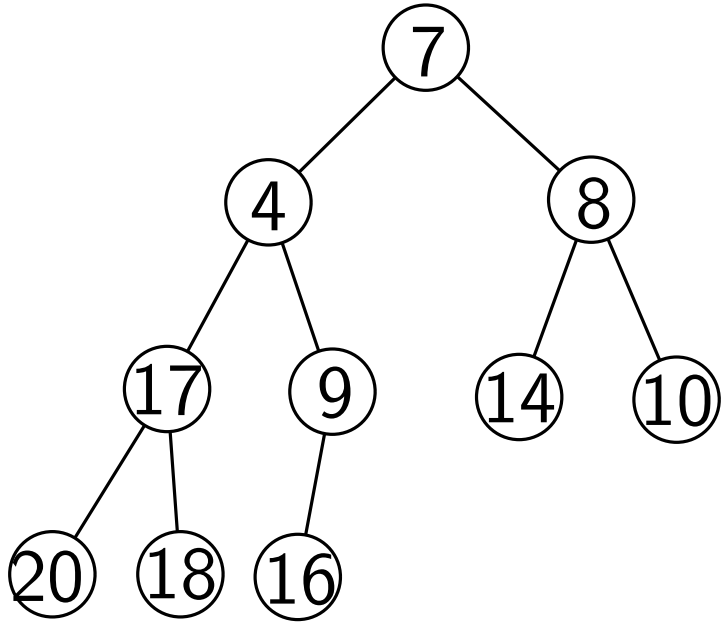

7
9
8
17
16
14
10
20
18
4

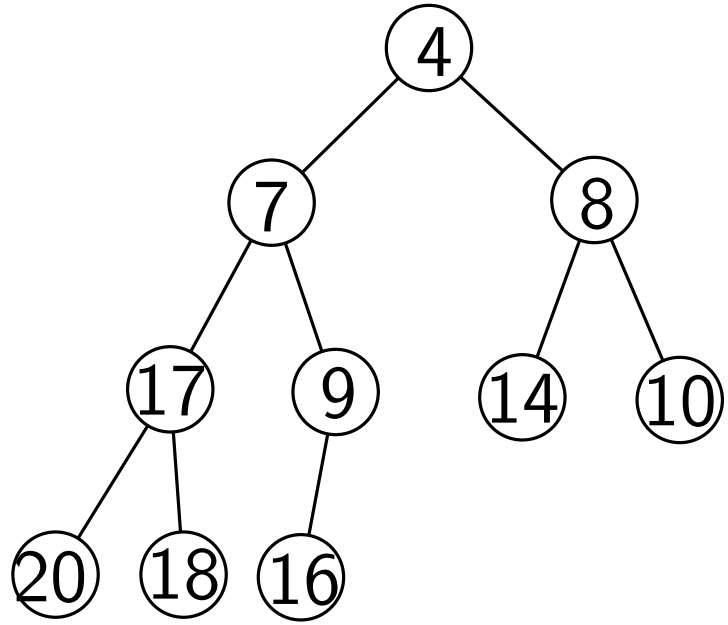

7
9
8
17
4
14
10
20
18
16


7
9
8
17
16
14
10
20
18

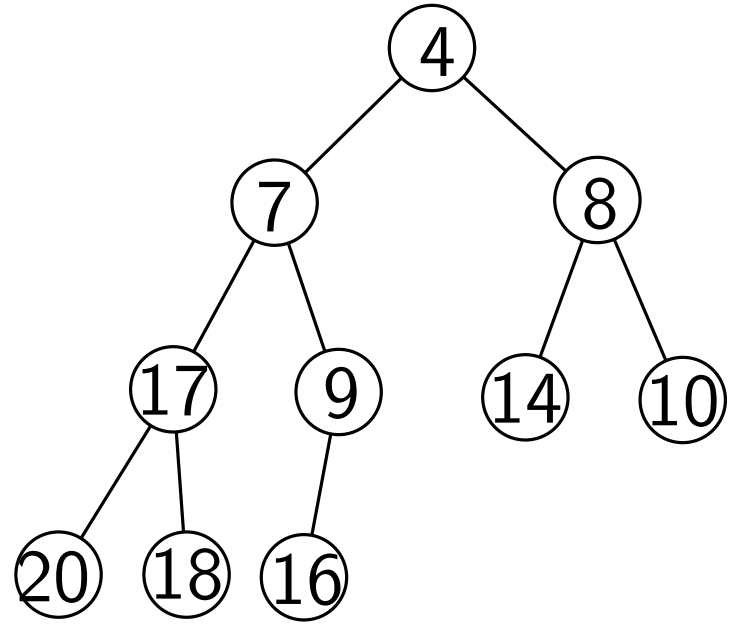

7
9
8
17
16
14
10
20
18
4


7
9
8
17
4
14
10
20
18
16


7
4
8
17
9
14
10
20
18
16



Running time is $O(\text{height}) = O(\log n)$.

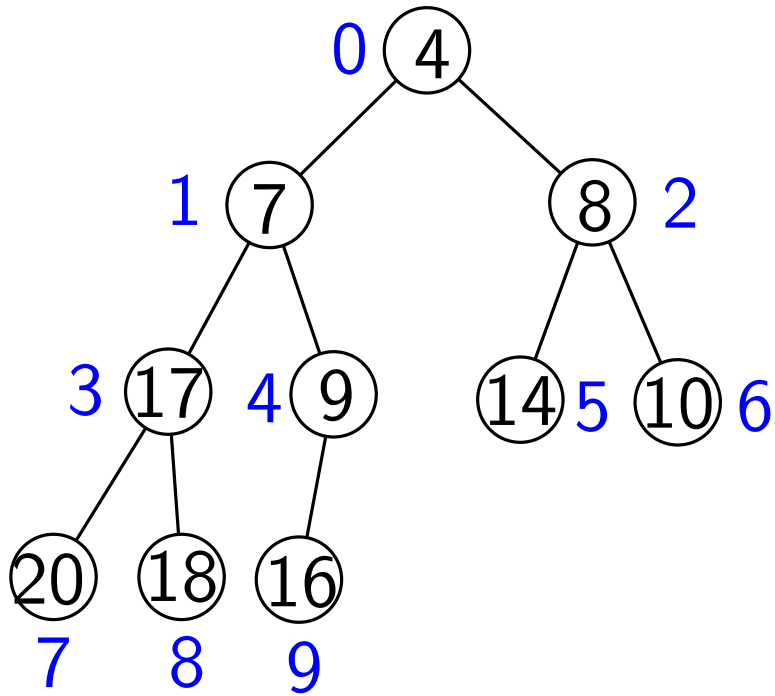


Running time is $O(\text{height}) = O(\log n)$.

So building a heap takes $O(n \log n)$ time. And sorting by repeated deleting the minimum takes $O(n \log n)$ time.

Array representation of a heap

1. Given a node at location i, its children are at locations $2i + 1$ and $2i + 2$.
2. Given a node at location i, its parent is at location $\lfloor (i - 1)/2 \rfloor$.
3. If n is the current number of values in the heap, then $A[n - 1]$ stores the rightmost node at the bottommost level.

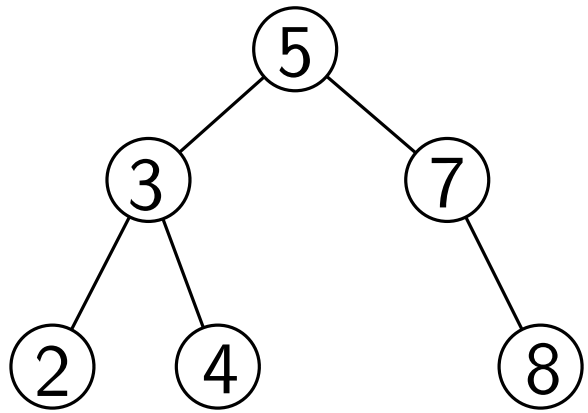


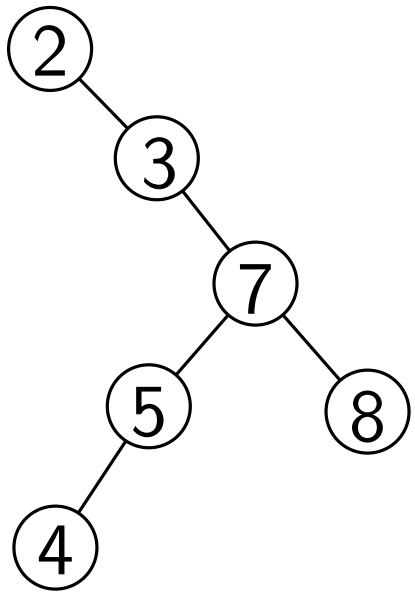
Binary Search Tree

- Store a set of values in the nodes of a binary tree.
- Binary search tree property: for any node x, the values in the left subtree of x are less than the value at x, and values in the right subtree of x are greater than the value at x.

Binary Search Tree

- Store a set of values in the nodes of a binary tree.
- Binary search tree property: for any node x, the values in the left subtree of x are less than the value at x, and values in the right subtree of x are greater than the value at x.





Not unique!

Searching

Suppose that we want to check if an input value k is stored in the tree. We start at the root x:

1. If root x is undefined, then return a null pointer.
2. If $k = \text{value}(x)$, then return a pointer to x.
3. If $k < \text{value}(x)$, then recurse on $\text{left}(x)$; otherwise, recurse on $\text{right}(x)$.

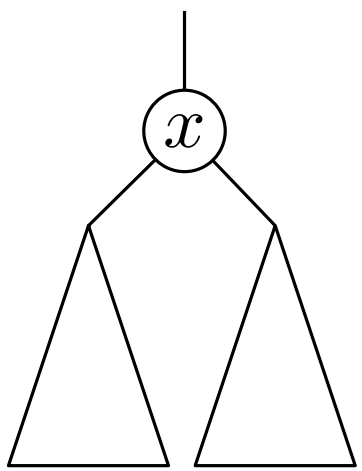
Finding minimum: Keep following left child pointer until the left child pointer is null. Then the current node stores the minimum.

Finding the maximum: Keep following the right child pointer until the right child pointer is null. Then the current node stores the maximum.

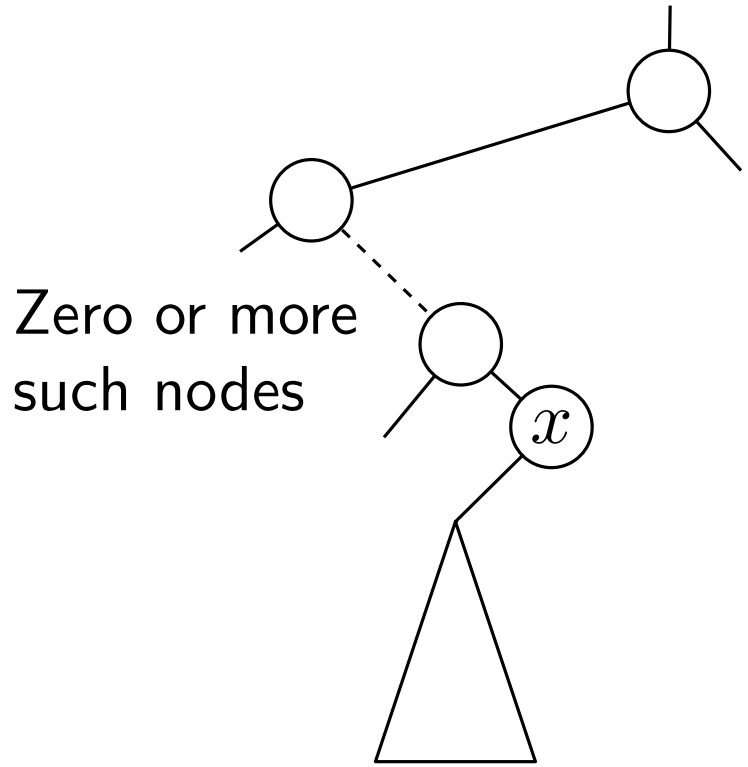
Running time $= O(\text{height})$.

Successor

The successor of a node x is the node that stores the value that immediately follows value(x) in the total order.



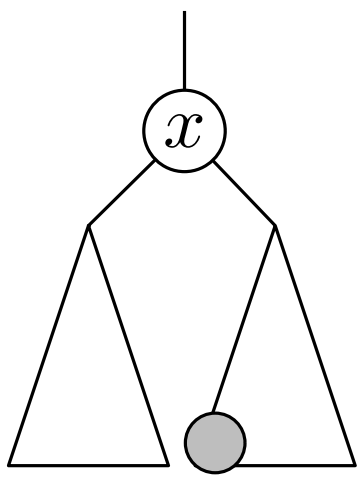
Case 1



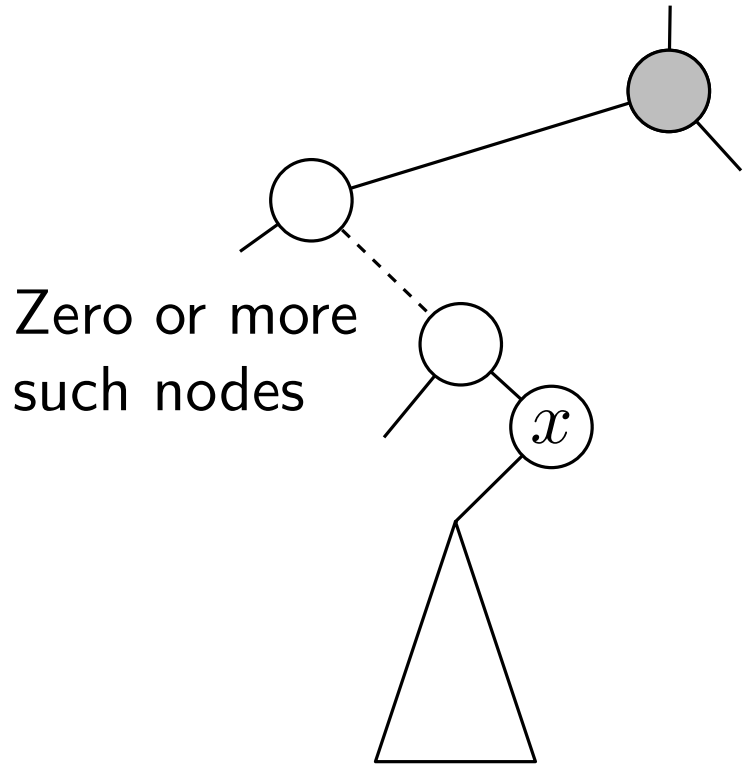
Case 2

Successor

The successor of a node x is the node that stores the value that immediately follows value(x) in the total order.



Case 1



Case 2

Succesor (x):

1. If right(x) is non-null then return the minimum in the subtree rooted at right(x).
2. Otherwise,
   (a) $y :=$ parent(x)
   (b) while y is not null and $x =$ right(y) do
      (i) $x := y$
      (ii) $y :=$ parent(y)
3. Return y

Succesor (x):

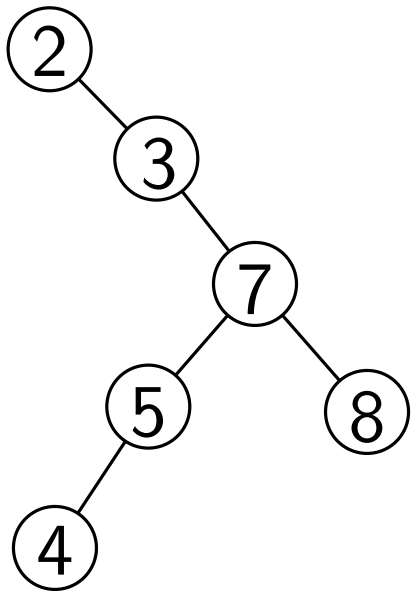
1. If right(x) is non-null then return the minimum in the subtree rooted at right(x).
2. Otherwise,
   (a) $y :=$ parent(x)
   (b) while y is not null and $x =$ right(y) do
      (i) $x := y$
      (ii) $y :=$ parent(y)
3. Return y

Predecessor can be symmetrically defined and handled.

Insertion

Suppose that we want to insert a new value k. Search for k in the tree. We will fall off a leaf x eventually. Create a new node to store k and make this new node an appropriate child of x.

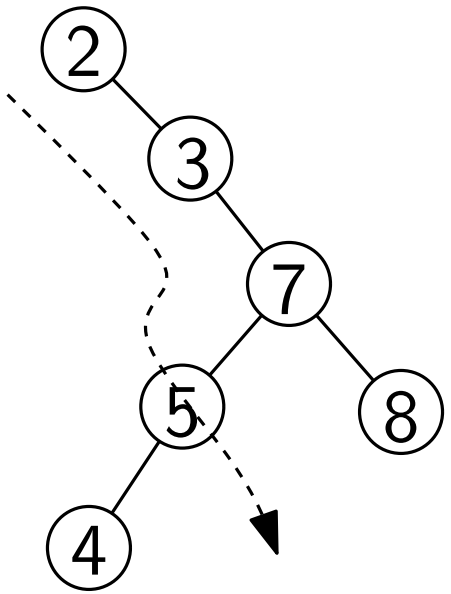
Insert 6



Insertion

Suppose that we want to insert a new value k. Search for k in the tree. We will fall off a leaf x eventually. Create a new node to store k and make this new node an appropriate child of x.

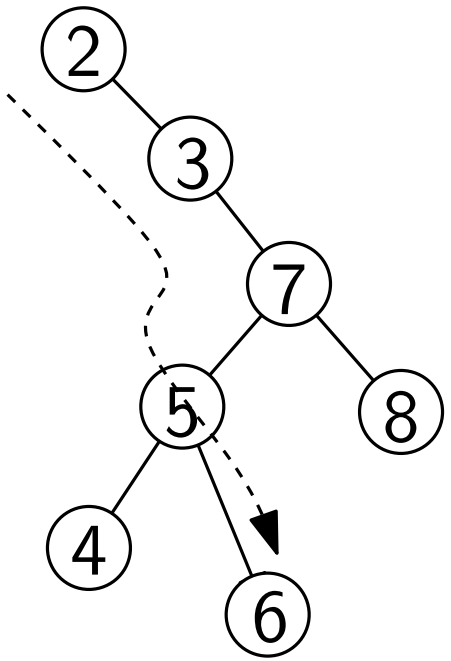
Insert 6



Insertion

Suppose that we want to insert a new value k. Search for k in the tree. We will fall off a leaf x eventually. Create a new node to store k and make this new node an appropriate child of x.

Insert 6

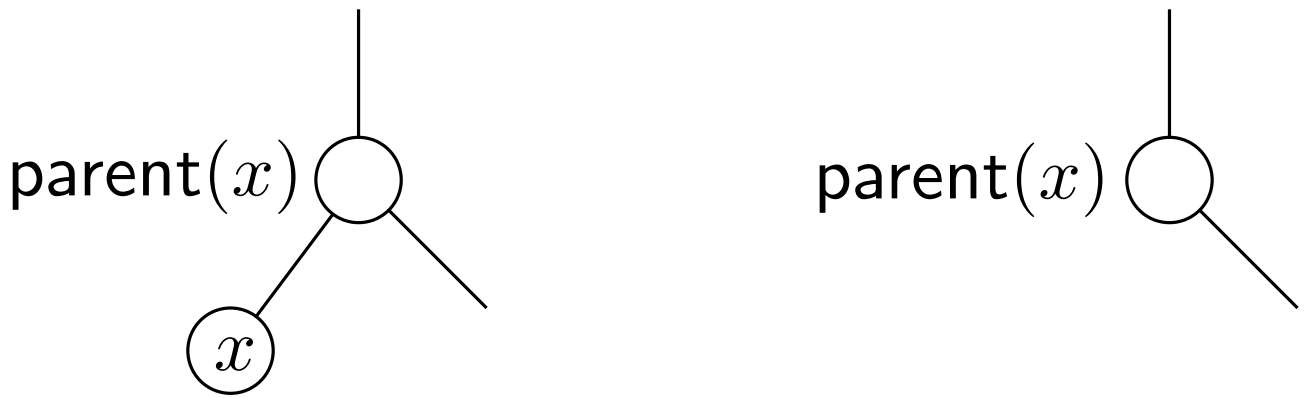


Insert(k) // Assume that k is absent

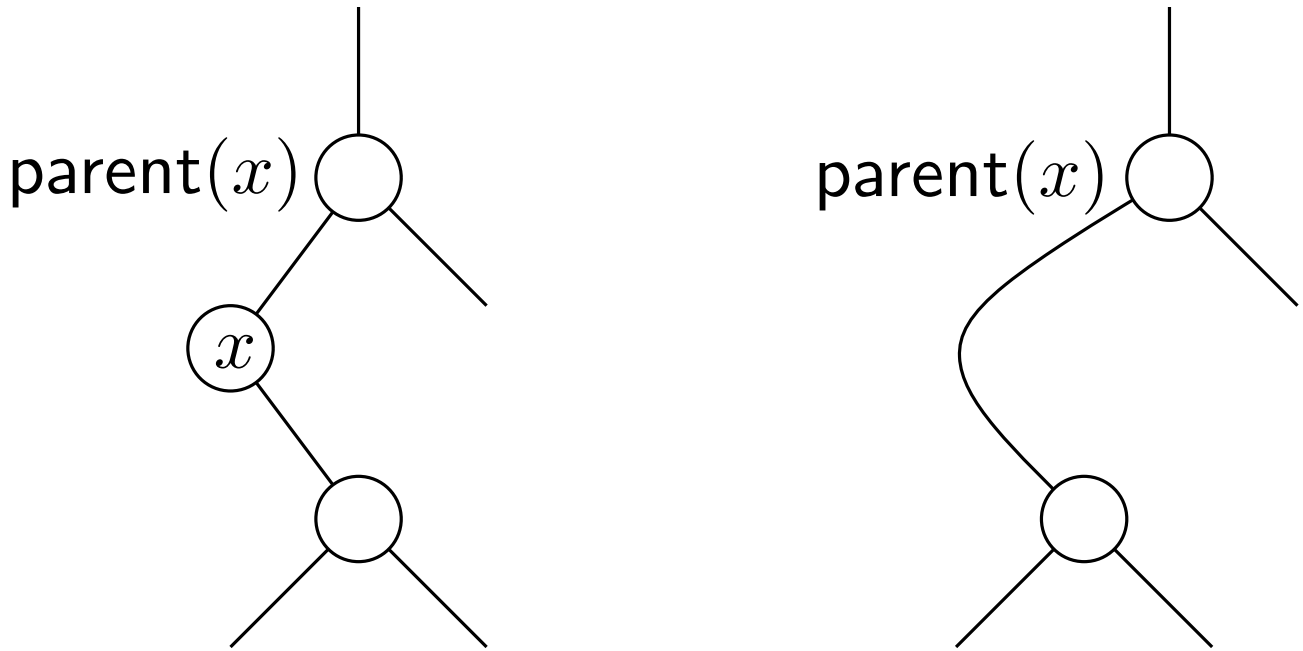
1. $y :=$ null; $x :=$ root
2. while x is not null do
   (a) $y := x$
   (b) if $k <$ value(x) then $x :=$ left(x); otherwise, $x :=$ right(x)
3. Create a new node z to store k.
4. parent(z) $:= y$
5. If y is null then root $:= z$, else if $k <$ value(y), then left(y) $:= z$, else right(y) $:= z$

Deletion

Suppose that we want to delete a value stored at a node x in the binary search tree. There are three cases.

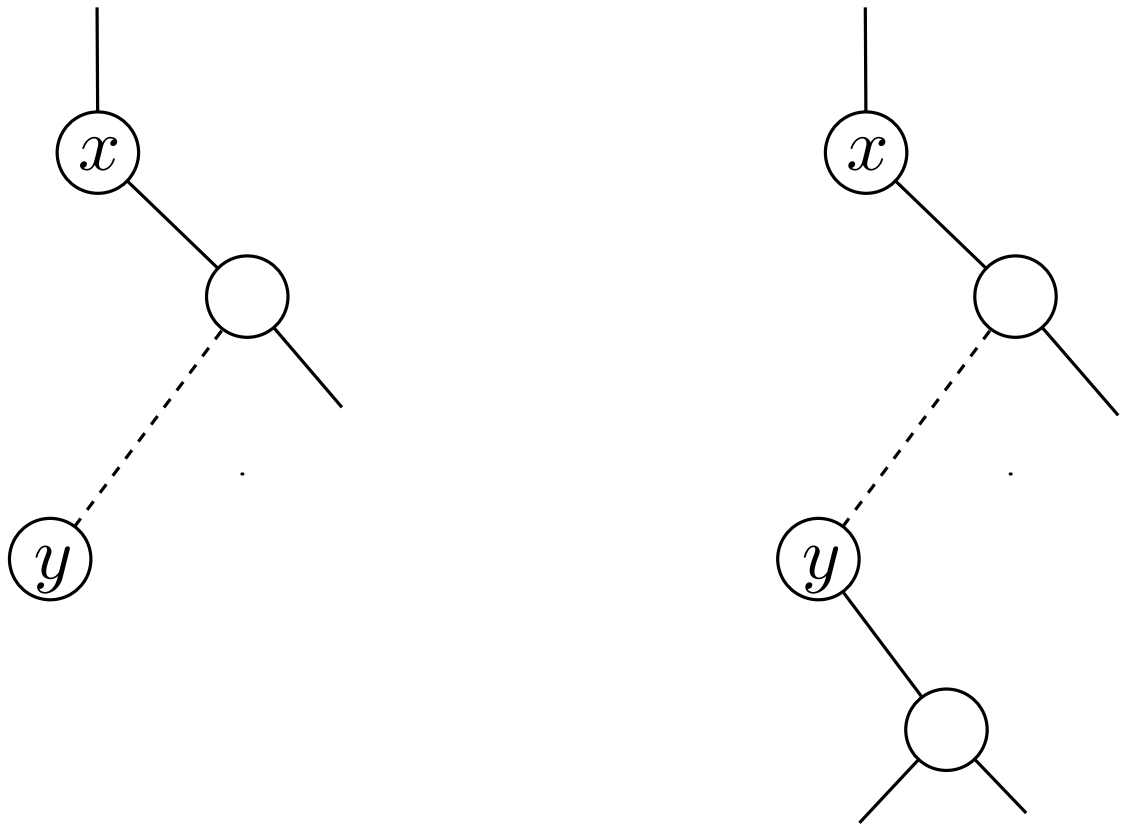


Case 1:x is a leaf.



Case 2: x has exactly one child.

Case 3: x has two children. Locate the successor y of x, which must have at most one child. Swap the contents of x and y. Then, delete the node y as in Case 1 or 2.



Red-Black Tree

A red-black tree T is a binary search tree that satisfies the red-black tree properties:

1. Every node is colored red or black.
2. If a node is red, either it is a leaf or it has two black children.
3. For any node x, every path from x to a leaf contains the same number of black nodes. This number is the black-height of x, denoted by $b(x)$.
4. The root of T is black.

Lemma: For any node x, the subtree rooted at x has at least $2^{b(x)} - 1$ nodes.

Lemma: For any node x, the subtree rooted at x has at least $2^{b(x)} - 1$ nodes.

Proof By induction on $b(x)$. The base cases are that $b(x) = 0$ or 1. In these cases, $2^{b(x)} - 1 \leq 2 - 1 = 1$, so the lemma holds.

Lemma: For any node x, the subtree rooted at x has at least $2^{b(x)} - 1$ nodes.

Proof By induction on $b(x)$. The base cases are that $b(x) = 0$ or 1. In these cases, $2^{b(x)} - 1 \leq 2 - 1 = 1$, so the lemma holds.

Assume $b(x) \geq 2$ and that the lemma is true when the black-height is less than $b(x)$.

Lemma: For any node x, the subtree rooted at x has at least $2^{b(x)} - 1$ nodes.

Proof By induction on $b(x)$. The base cases are that $b(x) = 0$ or 1. In these cases, $2^{b(x)} - 1 \leq 2 - 1 = 1$, so the lemma holds.

Assume $b(x) \geq 2$ and that the lemma is true when the black-height is less than $b(x)$.

If x is black, by induction assumption, the number of nodes is at least $2(2^{b(x)-1} - 1) + 1 = 2^{b(x)} - 1$.

Lemma: For any node x, the subtree rooted at x has at least $2^{b(x)} - 1$ nodes.

Proof By induction on $b(x)$. The base cases are that $b(x) = 0$ or 1. In these cases, $2^{b(x)} - 1 \leq 2 - 1 = 1$, so the lemma holds.

Assume $b(x) \geq 2$ and that the lemma is true when the black-height is less than $b(x)$.

If x is black, by induction assumption, the number of nodes is at least $2(2^{b(x)-1} - 1) + 1 = 2^{b(x)} - 1$.

If x is red, then x has two black children. By induction assumption, the number of nodes is at least $4(2^{b(x)-1} - 1) + 3 = 2^{b(x)+1} - 1 > 2^{b(x)} - 1$.

Theorem: The height of a red-back tree of n nodes is at most $2\log_2(n+1) = O(\log n)$.

Theorem: The height of a red-back tree of n nodes is at most $2\log_2(n+1) = O(\log n)$.

Proof: Let x be the root. Let $h(x)$ denote the height of x (i.e., the length of the longest path from x to a leaf).

We have $b(x) \geq h(x)/2$. Thus, the number of nodes is at least $2^{b(x)} - 1 \geq 2^{h(x)/2} - 1$. Thus, $n \geq 2^{h(x)/2} - 1$, implying that $h(x) \leq 2\log_2(n+1)$.

Consequently, each of the following operations takes $O(\log n)$ time on a red-black tree:

- search
- successor and predecessor
- insertion
- deletion

Consequently, each of the following operations takes $O(\log n)$ time on a red-black tree:

- search
- successor and predecessor
- insertion
- deletion

But how do we retore the red-back tree properties after an insertion or deletion?

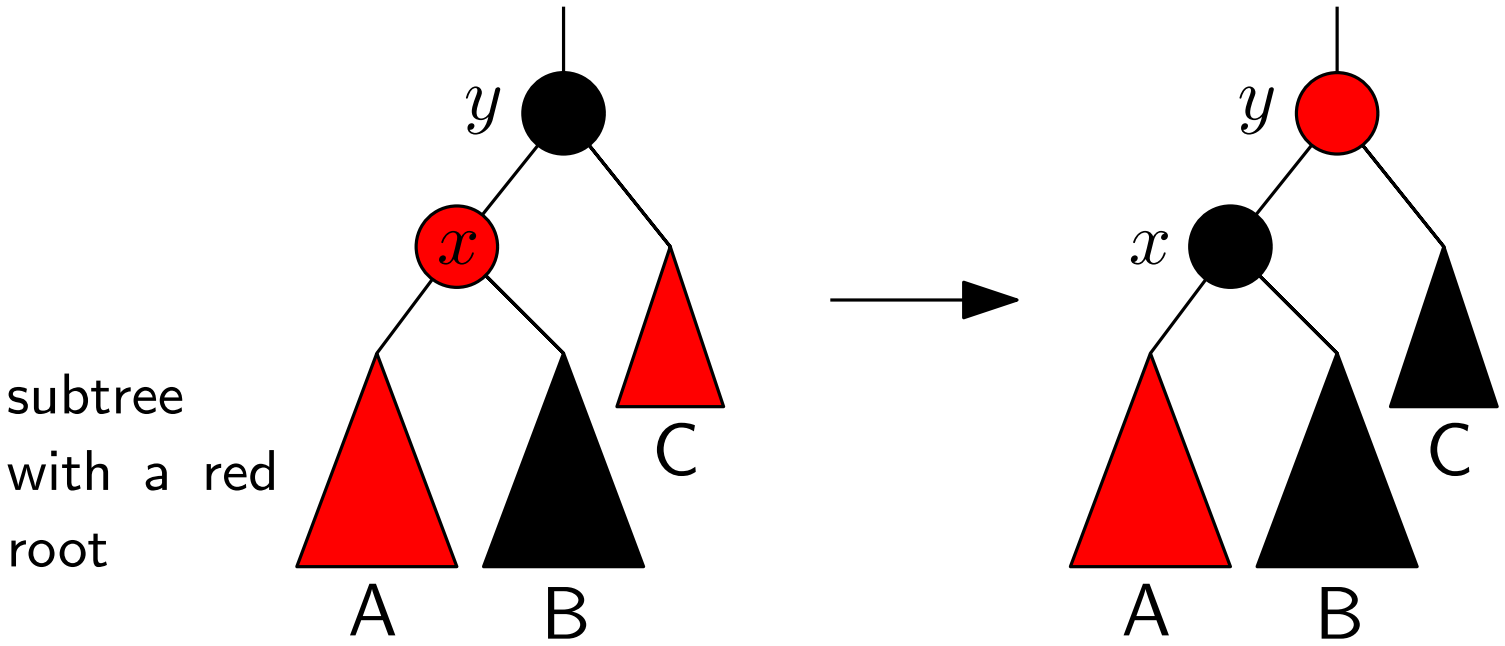
Insertion

A new node is first inserted as a red node as in the case of an ordinary binary search tree. Afterwards, the new red node may be the child of a red parent. Thus, we need to fix this violation of the red-black tree properties.

Insertion

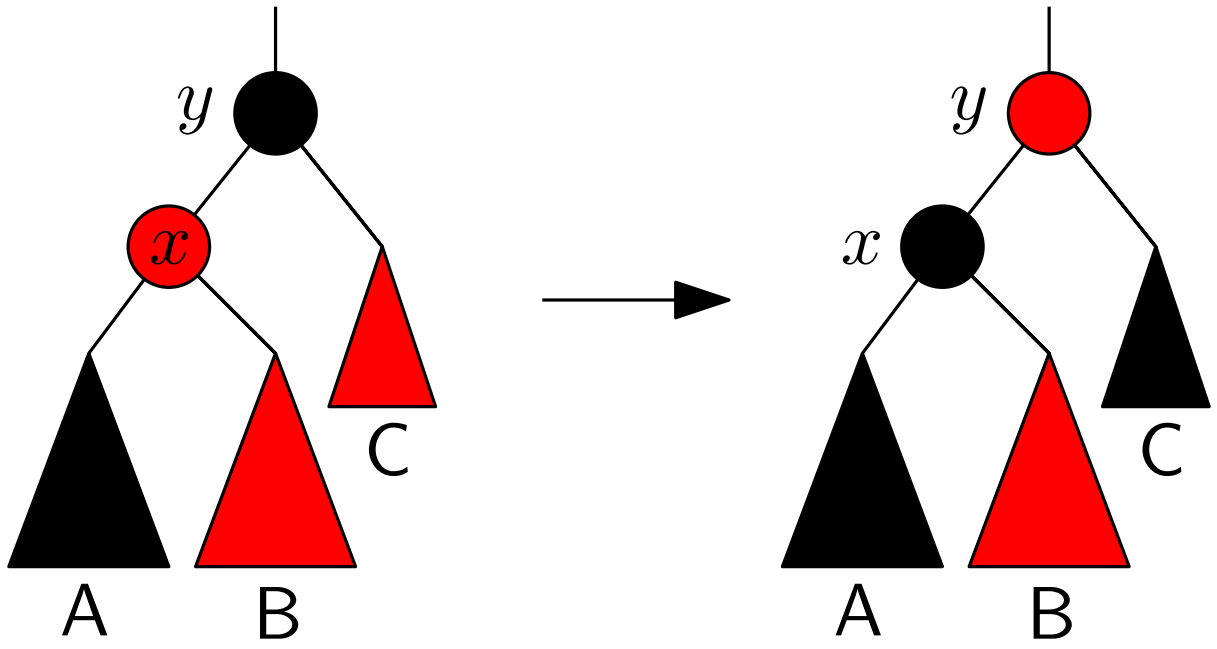
A new node is first inserted as a red node as in the case of an ordinary binary search tree. Afterwards, the new red node may be the child of a red parent. Thus, we need to fix this violation of the red-black tree properties.

The idea is to propagate this violation upward until we can fix it using at most two rotations or there is no more violation but the root becomes red. In the latter case, we just color the root black again.

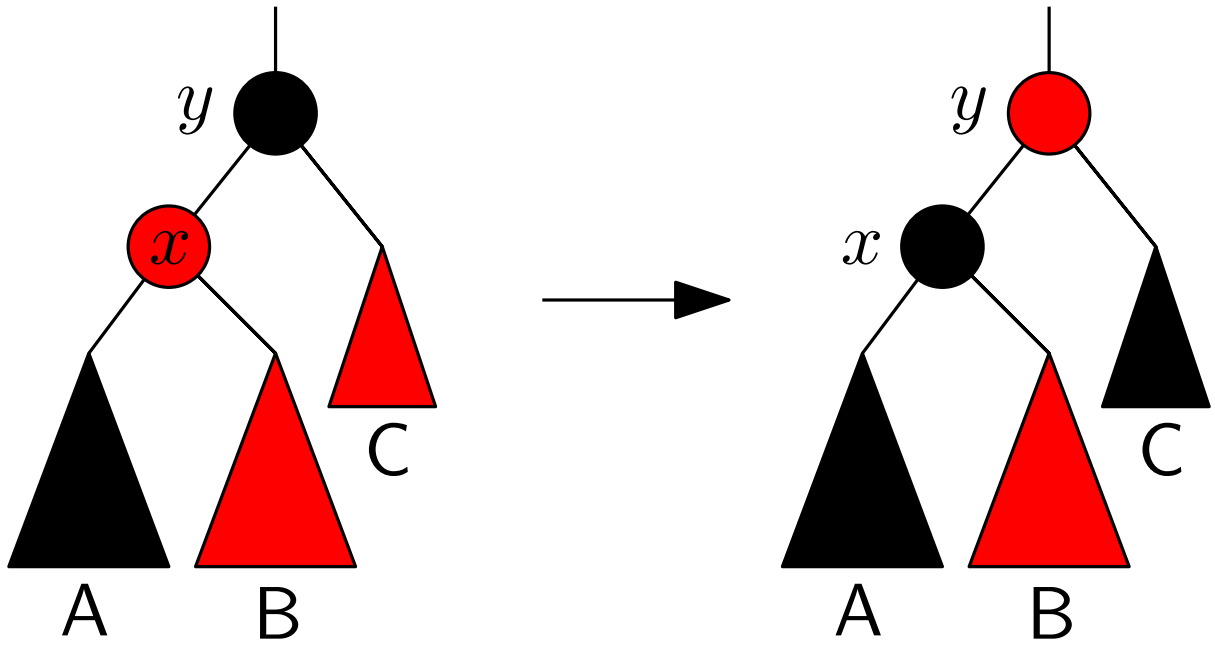


The sibling of x is red. Propagate upward.

A null pointer is treated as a black subtree.

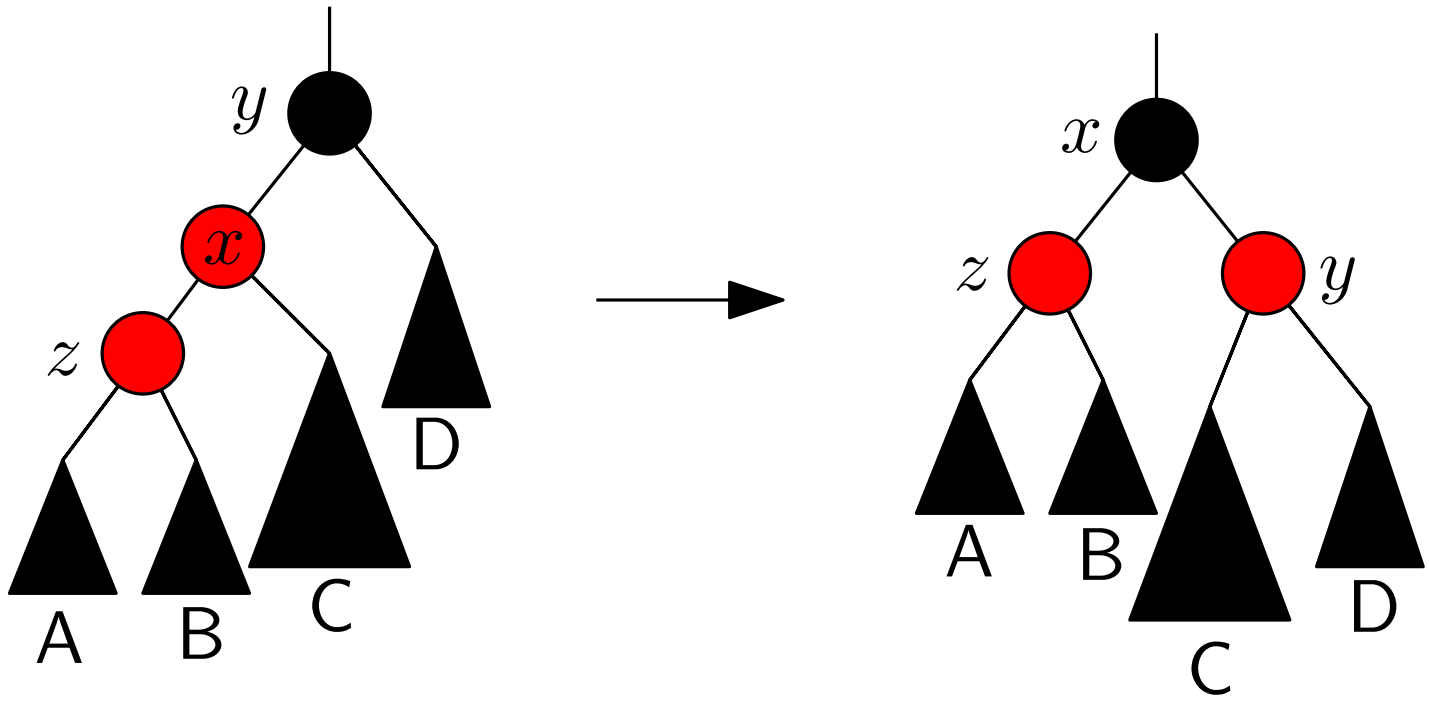


The sibling of x is red. Propagate upward.

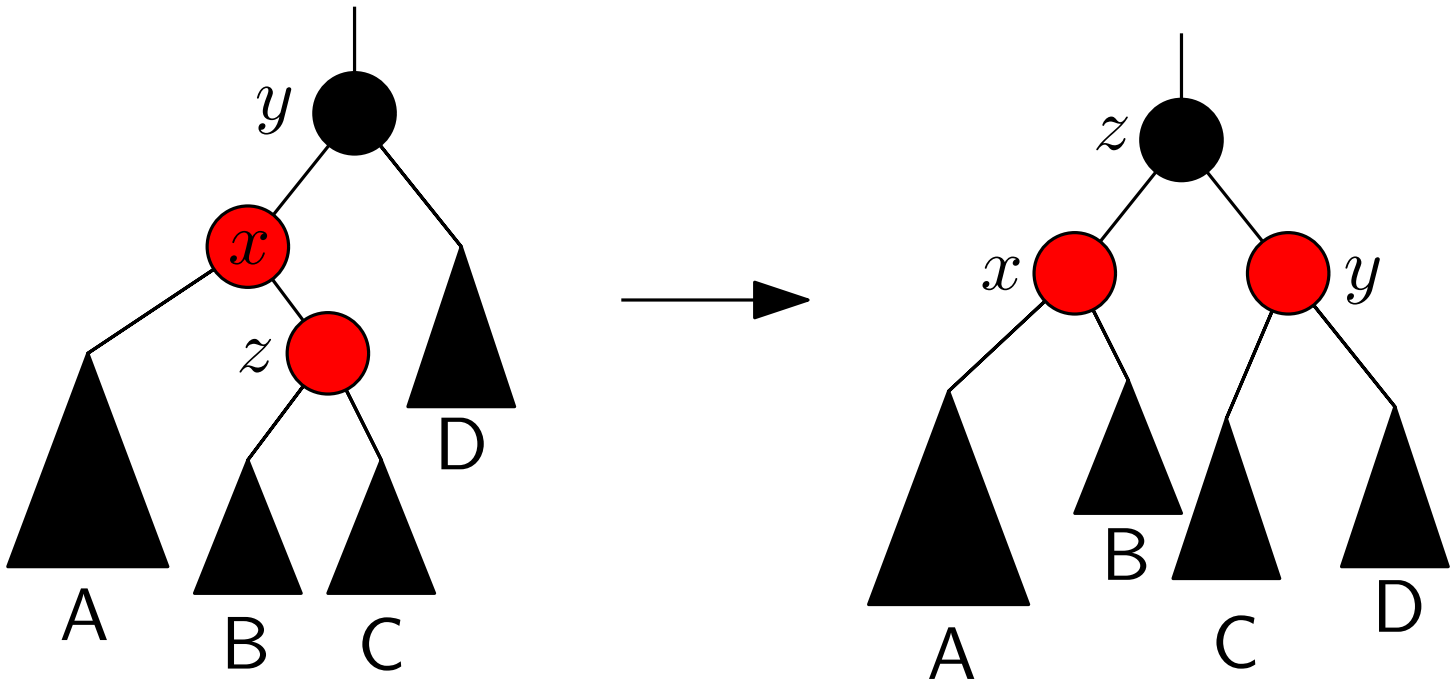


The sibling of x is red. Propagate upward.

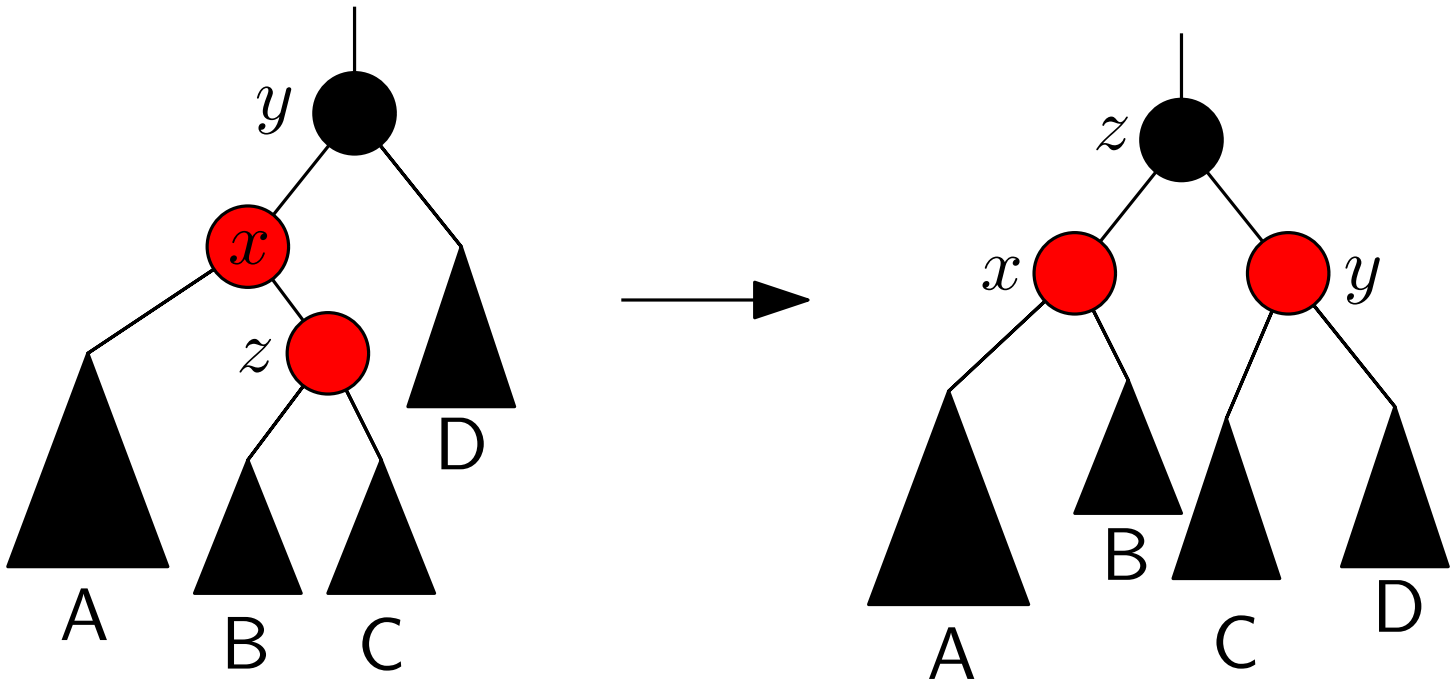
Symmetric cases: x is the right child of its black parent.



The sibling of x is black. Terminating case.



The sibling of x is black. Terminating case.



The sibling of x is black. Terminating case.

Symmetric cases: x is the right child of its parent.

Deletion

Delete a node as in the case of an ordinary binary search tree. Note that the deleted node has at most one child. If the deleted node is red, nothing goes wrong. If the deleted node is black, the property about black-height is violated.

Deletion

Delete a node as in the case of an ordinary binary search tree. Note that the deleted node has at most one child. If the deleted node is red, nothing goes wrong. If the deleted node is black, the property about black-height is violated.

If the deleted black node has a child, we color that child black. If that child was red, then we are done. If it was already black, it becomes doubly black.

Deletion

Delete a node as in the case of an ordinary binary search tree. Note that the deleted node has at most one child. If the deleted node is red, nothing goes wrong. If the deleted node is black, the property about black-height is violated.

If the deleted black node has a child, we color that child black. If that child was red, then we are done. If it was already black, it becomes doubly black.

If the deleted node is a black leaf, we view that its parent gets a doubly black null child pointer.

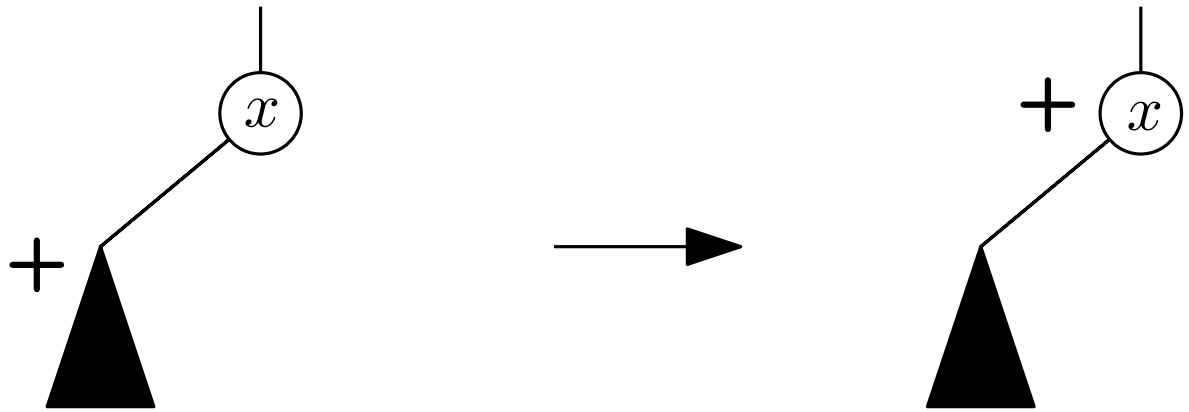
Deletion

Delete a node as in the case of an ordinary binary search tree. Note that the deleted node has at most one child. If the deleted node is red, nothing goes wrong. If the deleted node is black, the property about black-height is violated.

If the deleted black node has a child, we color that child black. If that child was red, then we are done. If it was already black, it becomes doubly black.

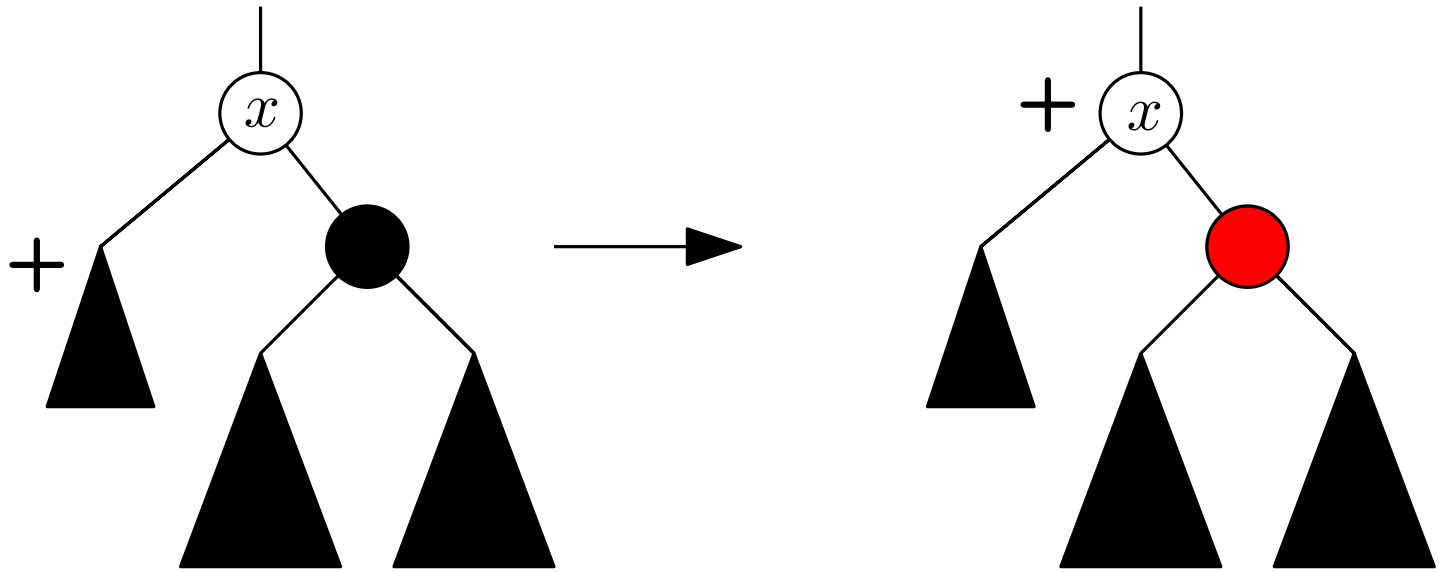
If the deleted node is a black leaf, we view that its parent gets a doubly black null child pointer.

In any case, we need to fix this double blackness issue. The idea is to propagate the double blackness upward until we can fix it using a constant number of rotations.

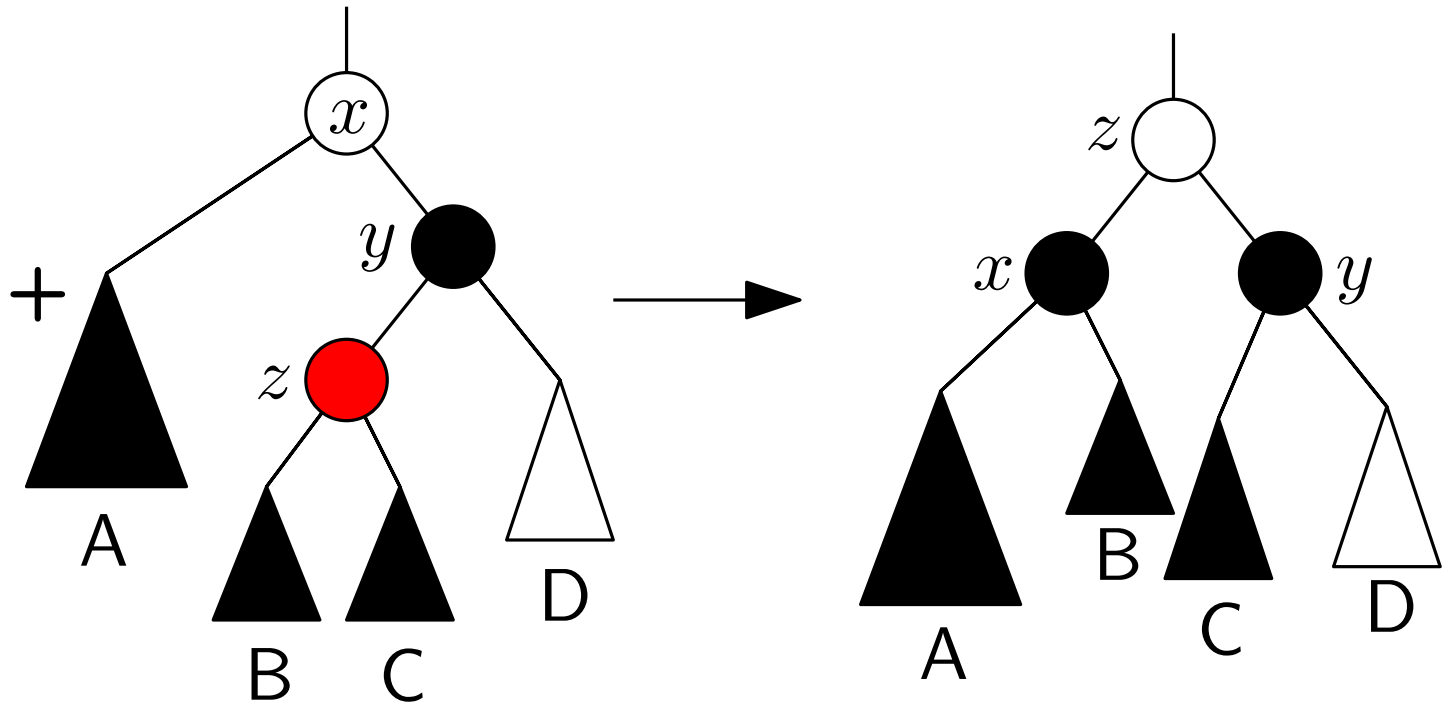


subtree with a doubly black root or a doubly black null pointer

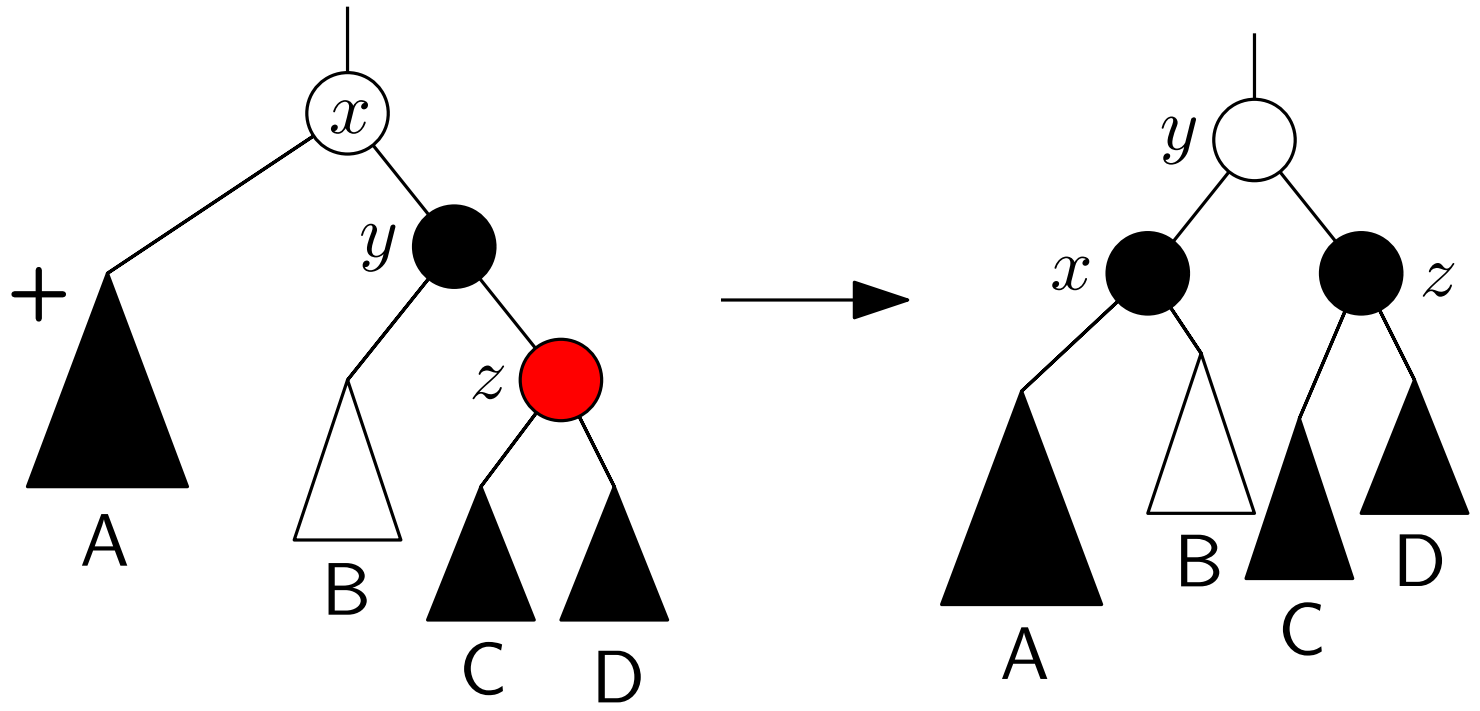
The sibling of the doubly blackness is absent. Note that x could not be red. So we have propagated the doubly blackness upward.



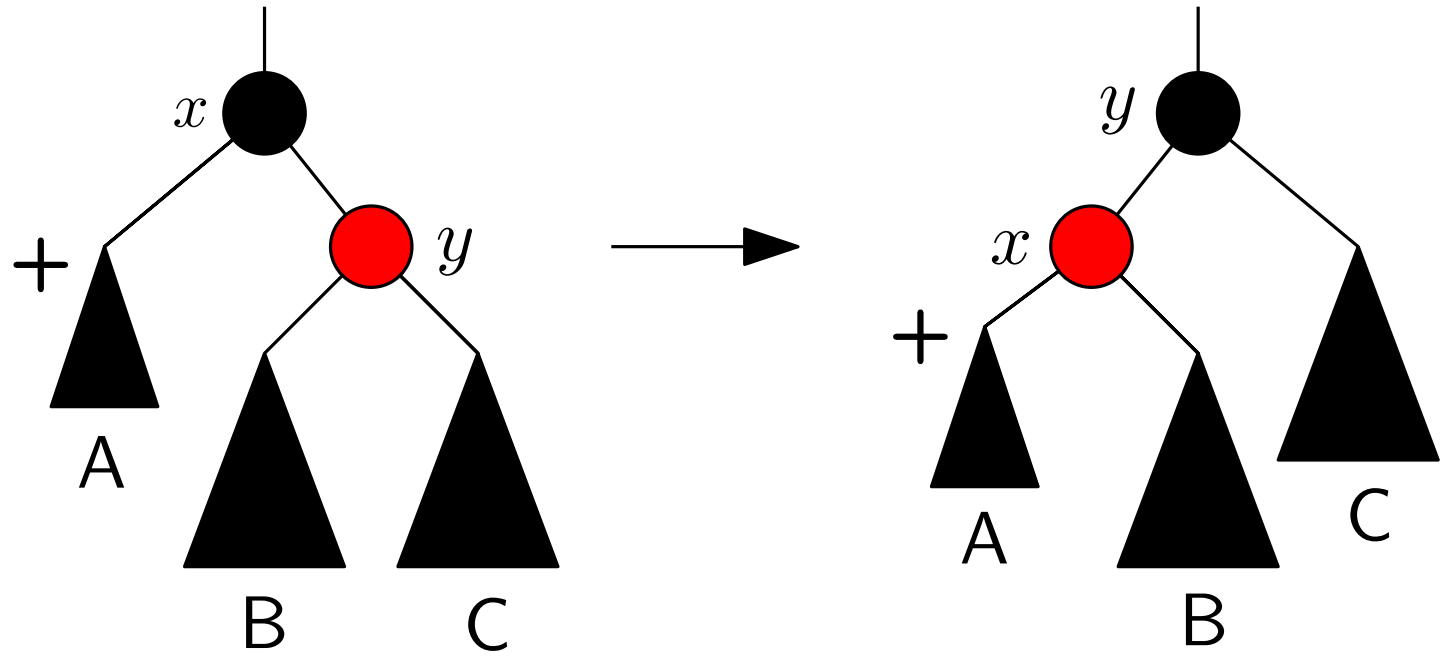
The sibling of the doubly blackness is black and so are the children of this sibling. If x is red, this is a terminaing case because we are done after coloring x black. Otherwise, we have propagated the doubly blackness upward.



The sibling of the doubly blackness is black and a child of this sibling is red. Terminating case.



The sibling of the doubly blackness is black and a child of this sibling is red. Terminating case.



The sibling of the doubly blackness is red. A single rotation makes the sibling of the doubly blackness black. Also, note that the parent of the doubly blackness is red. So we have reduced the case to one of the previous cases and there will be no more upward propagation afterwards as the parent of the doubly blackness is red.

It takes $O(\log n)$ time to fix the red-black tree properties after an insertion or deletion. Specifically, there are $O(\log n)$ upward propagations and $O(1)$ rotations.

The running time of insert and delete is $O(\log n)$.