16. Binary Search Trees

[Ottman/Widmayer, Kap. 5.1, Cormen et al, Kap. 12.1 - 12.3]

Dictionary implementation

Hashing: implementation of dictionaries with expected very fast access times.

Disadvantages of hashing:

Dictionary implementation

Hashing: implementation of dictionaries with expected very fast access times.

Disadvantages of hashing: linear access time in worst case. Some operations not supported at all:

- enumerate keys in increasing order

Dictionary implementation

Hashing: implementation of dictionaries with expected very fast access times.

Disadvantages of hashing: linear access time in worst case. Some operations not supported at all:

- enumerate keys in increasing order
- next smallest key to given key

Dictionary implementation

Hashing: implementation of dictionaries with expected very fast access times.

Disadvantages of hashing: linear access time in worst case. Some operations not supported at all:

- enumerate keys in increasing order
- next smallest key to given key
- Key k in given interval $k \in [l, r]$

Trees

Trees are

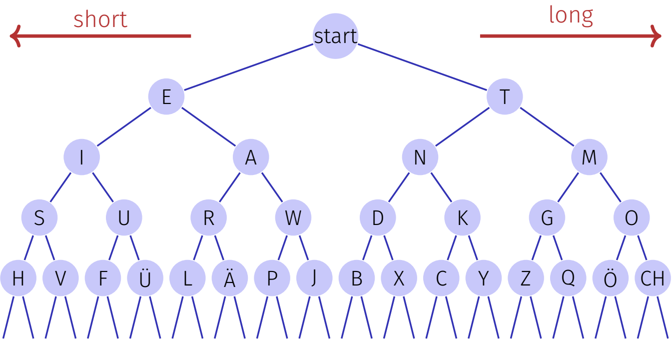
- Generalized lists: nodes can have more than one successor
- Special graphs: graphs consist of nodes and edges. A tree is a fully connected, directed, acyclic graph.

Trees

Use

- Decision trees: hierarchic representation of decision rules
- syntax trees: parsing and traversing of expressions, e.g. in a compiler
- Code tress: representation of a code, e.g. morse alphabet, huffman code
- Search trees: allow efficient searching for an element by value

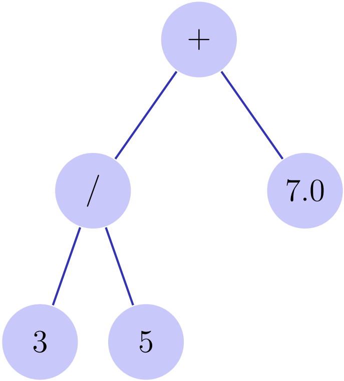
Examples



Morsealphabet

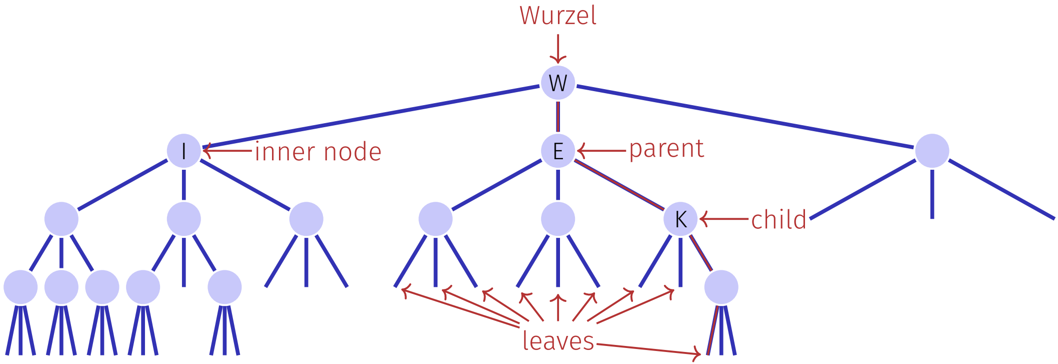
Examples

$3/5 + 7.0$



Expression tree

Nomenclature



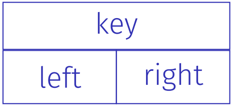
- *Order* of the tree: maximum number of child nodes (here: 3)
- *Height* of the tree: maximum path length root to leaf (here: 4)

Binary Trees

A *binary tree* is

- either a leaf, i.e. an empty tree,
- or an inner leaf with two trees T_l (left subtree) and T_r (right subtree) as left and right successor.

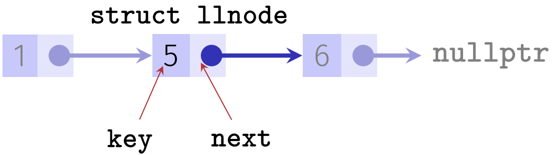
In each inner node **`v`** we store



- a key **`v.key`** and
- two nodes **`v.left`** and **`v.right`** to the roots of the left and right subtree.

a leaf is represented by the **null**-pointer

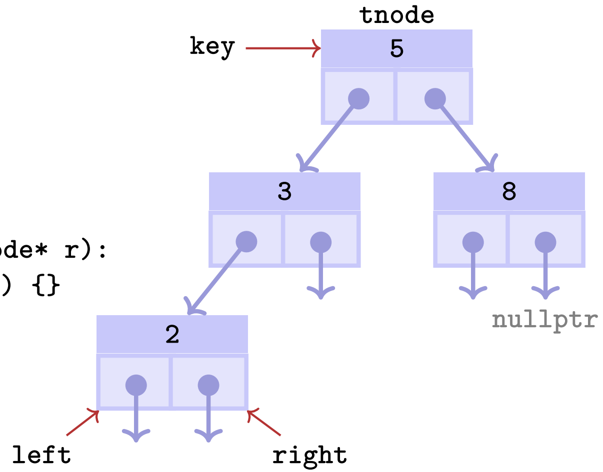
Recap: Linked-list Node in C++




```
struct llnode {
  int key;
  llnode* next;
  llnode(int k, llnode* n): key(k), next(n) {} // Constructor
};
```

Recap: Tree Nodes in C++

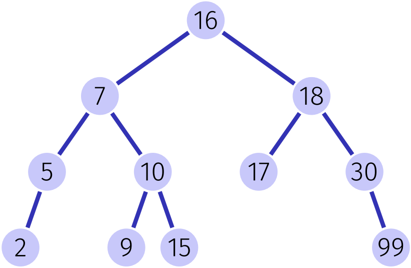
```
struct tnode {
  int key;
  tnode* left;
  tnode* right;
  tnode(int k, tnode* l, tnode* r):
    key(k), left(l), right(r) {}
};
```



Binary search tree

A *binary search tree* is a binary tree that fulfils the **search tree property**:

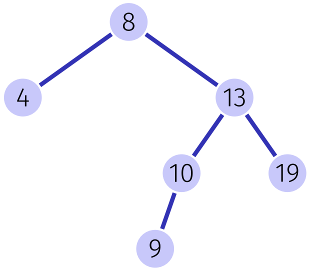
- Every node `v` stores a key
- Keys in left subtree `v.left` are smaller than `v.key`
- Keys in right subtree `v.right` are greater than `v.key`



Searching

Input: Binary search tree with root r, key k
Output: Node v with $v.\text{key} = k$ or **null**

```
v ← r
while v ≠ null do
    if k = v.key then
        return v
    else if k < v.key then
        v ← v.left
    else
        v ← v.right
return null
```

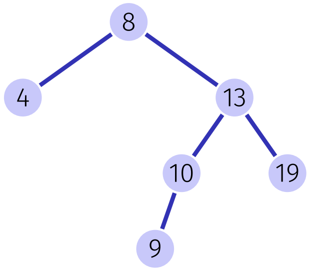


Searching

Input: Binary search tree with root r, key k
Output: Node v with $v.\text{key} = k$ or **null**

$v \leftarrow r$
while $v \neq$ **null do**
  if $k = v.\text{key}$ **then**
    return v
  else if $k < v.\text{key}$ **then**
    $v \leftarrow v.\text{left}$
  else
    $v \leftarrow v.\text{right}$

return null

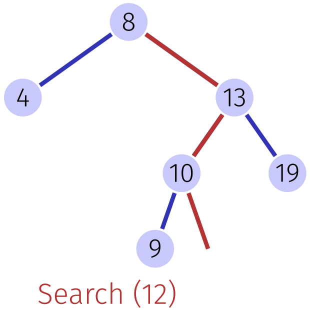


Search (12)

Searching

Input: Binary search tree with root r, key k
Output: Node v with $v.\text{key} = k$ or **null**

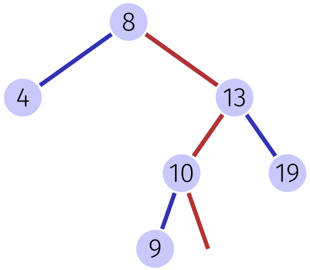
$v \leftarrow r$
while $v \neq$ **null do**
    if $k = v.\text{key}$ **then**
        return v
    else if $k < v.\text{key}$ **then**
        $v \leftarrow v.\text{left}$
    else
        $v \leftarrow v.\text{right}$
return null



Searching

Input: Binary search tree with root r, key k
Output: Node v with $v.\text{key} = k$ or **null**

```
v ← r
while v ≠ null do
    if k = v.key then
        return v
    else if k < v.key then
        v ← v.left
    else
        v ← v.right
return null
```



Search (12) $\rightarrow$ **null**

Searching in C++

```
bool contains(const llnode* root, int search_key) {
  while (root != nullptr) {
    if (search_key == root->key) return true;
    else if (search_key < root->key) root = root->left;
    else root = root->right;
  }

  return false;
}
```

Searching in C++

```
bool contains(const llnode* root, int search_key) {
  while (root != nullptr) {
    if (search_key == root->key) return true;
    else if (search_key < root->key) root = root->left;
    else root = root->right;
  }

  return false;
}
```

Remarks (pot. also for subsequent code):

- `contains` would typically be a member of function of `struct tnode` or `class bin_search_tree` ($\rightarrow$ slightly different signature)
- Recursive implementation also possible

Height of a tree

The height $h(T)$ of a binary tree T with root r is given by

Height of a tree

The height $h(T)$ of a binary tree T with root r is given by

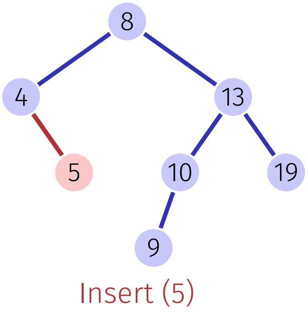
$$h(r) = \begin{cases} 0 & \text{if } r = \textbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{otherwise.} \end{cases}$$

The (worst case) run time of the search is thus $\mathcal{O}(h(T))$

Insertion of a key

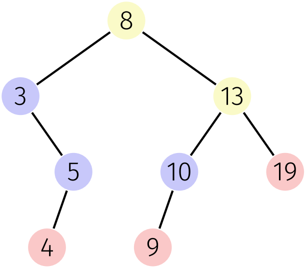
Insertion of the key k

- Search for k
- If successful search: e.g. output error
- If no success: insert the key at the leaf reached



Remove node

Three cases possible:

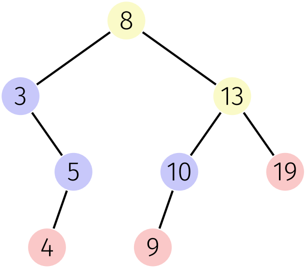


Remove node

Three cases possible:

- Node has no children
- Node has one child
- Node has two children

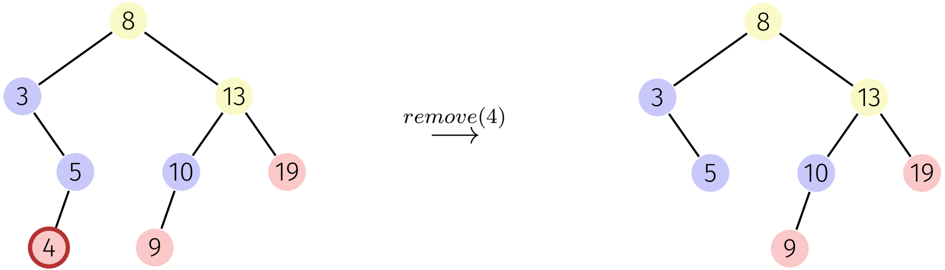
[Leaves do not count here]



Remove node

Node has no children

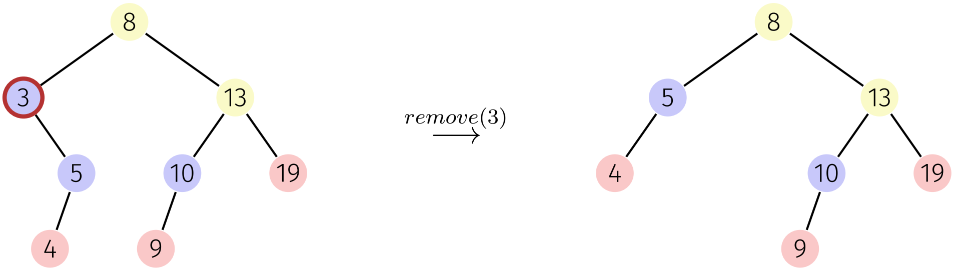
Simple case: replace node by leaf.



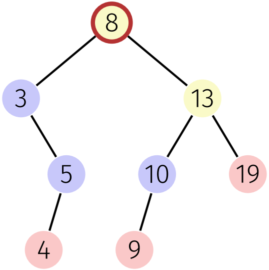
Remove node

Node has one child

Also simple: replace node by single child.



Remove node

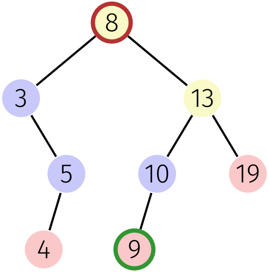


Node **v** has two children

Requirements for replacement node **w**:

1. **w.key** is larger than all keys in **v.left**
2. **w.key** is smaller than all keys in **v.right**
3. ideally has not children

Remove node



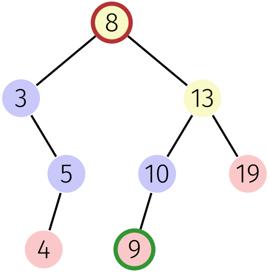
Node `v` has two children

Requirements for replacement node **`w`**:

1. **`w.key`** is larger than all keys in **`v.left`**
2. **`w.key`** is smaller than all keys in **`v.right`**
3. ideally has not children

Observation: the smallest key in the right subtree **`v.right`** (here: 9) meets requirements 1, 2; and has at most one (right) child.

Remove node



Node **v** has two children

Requirements for replacement node **w**:

1. **w.key** is larger than all keys in **v.left**
2. **w.key** is smaller than all keys in **v.right**
3. ideally has not children

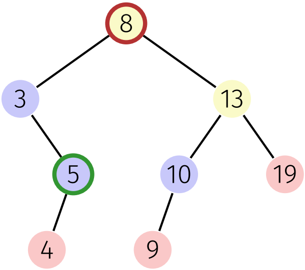
Observation: the smallest key in the right subtree **v.right** (here: 9) meets requirements 1, 2; and has at most one (right) child.

Solution: replace **v** by exactly this *symmetric successor*.

By symmetry ...

Node **v** has two children

Also possible: replace **v** by its *symmetric predecessor*.



Algorithm SymmetricSuccessor(v)

Input: Node v of a binary search tree.
Output: Symmetric successor of v

$w \leftarrow v.\text{right}$
$x \leftarrow w.\text{left}$
while $x \neq$ **null do**
    $w \leftarrow x$
    $x \leftarrow x.\text{left}$
return w

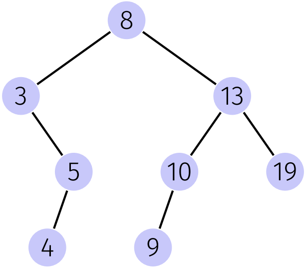
Analysis

Deletion of an element v from a tree T requires $\mathcal{O}(h(T))$ fundamental steps:

- Finding v has costs $\mathcal{O}(h(T))$
- If v has maximal one child unequal to **null**then removal takes $\mathcal{O}(1)$ steps
- Finding the symmetric successor n of v takes $\mathcal{O}(h(T))$ steps. Removal and insertion of n takes $\mathcal{O}(1)$ steps.

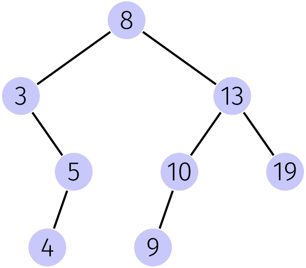
Traversal possibilities

- *preorder*:
  v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.



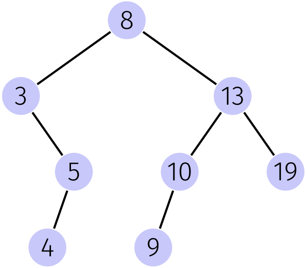
Traversal possibilities

- *preorder*:
  v, then $T_{\mathrm{left}}(v)$, then $T_{\mathrm{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19



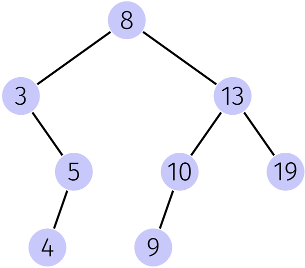
Traversal possibilities

- *preorder*:
  v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- *postorder*:
  $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.



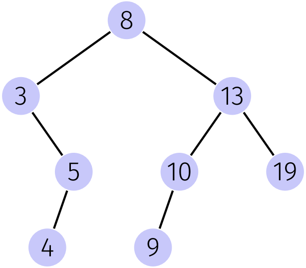
Traversal possibilities

- *preorder*:
  v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- *postorder*:
  $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8



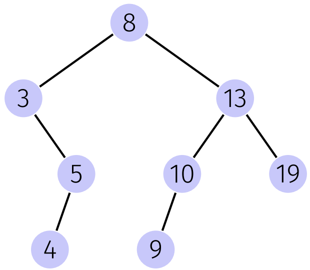
Traversal possibilities

- *preorder*:
  v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- *postorder*:
  $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- *inorder*:
  $T_{\text{left}}(v)$, then v, then $T_{\text{right}}(v)$.



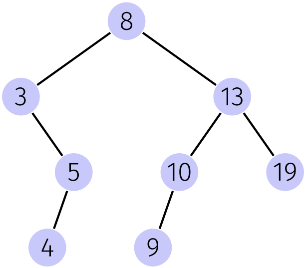
Traversal possibilities

- *preorder*:
  v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- *postorder*:
  $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- *inorder*:
  $T_{\text{left}}(v)$, then v, then $T_{\text{right}}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19

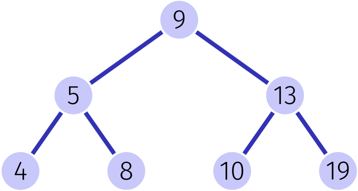


Further supported operations

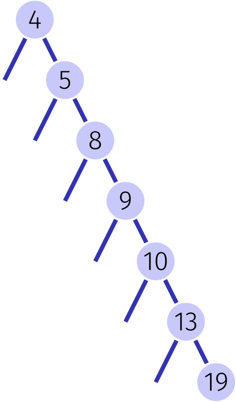
- *Min/Max(T)*: Query minimal/maximal value in $\mathcal{O}(h(T))$
- *ExtractMin/Max(T)*: Query and remove remove min/max in $\mathcal{O}(h(T))$
- *List(T)*: Output the sorted list of elements
- *Join(T_1, T_2)*: Merge two trees with *Max(T_1)* < *Min(T_2)* in $\mathcal{O}(h(T_1, T_2))$



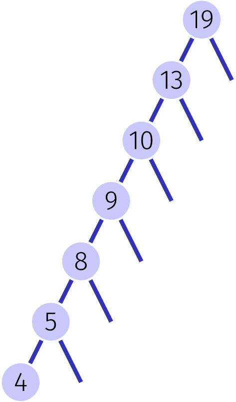
Search Trees: Balanced vs. Degenerated



insert 9,5,13,4,8,10,19:
ideally balanced



insert 4,5,8,9,10,13,19:
linear list



insert 19,13,10,9,8,5,4:
linear list

Probabilistically

A search tree constructed from a random sequence of numbers provides an an expected path length of $\mathcal{O}(\log n)$.

Attention: this only holds for insertions. If the tree is constructed by random insertions and deletions, the expected path length is $\mathcal{O}(\sqrt{n})$.

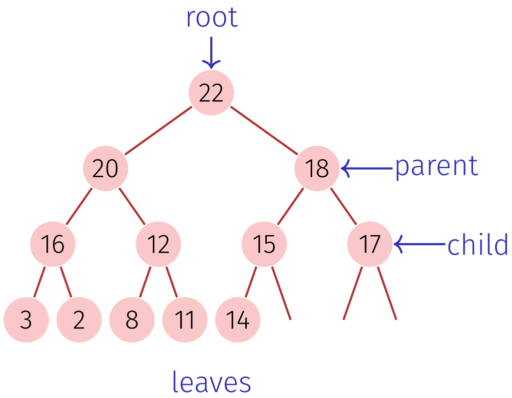
Balanced trees make sure (e.g. with *rotations*) during insertion or deletion that the tree stays balanced and provide a $\mathcal{O}(\log n)$ Worst-case guarantee.

17. Heaps

Data structure optimized for fast extraction of minimum or maximum and for sorting. [Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

[Max-]Heap*

Binary tree with the following properties

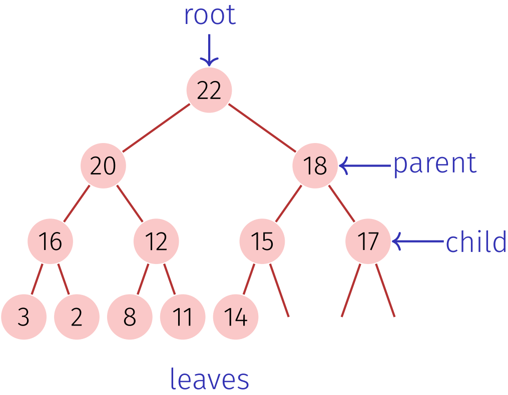


*Heap(data structure), not as in “heap and stack” (memory allocation)

[Max-]Heap*

Binary tree with the following properties

1. complete up to the lowest level

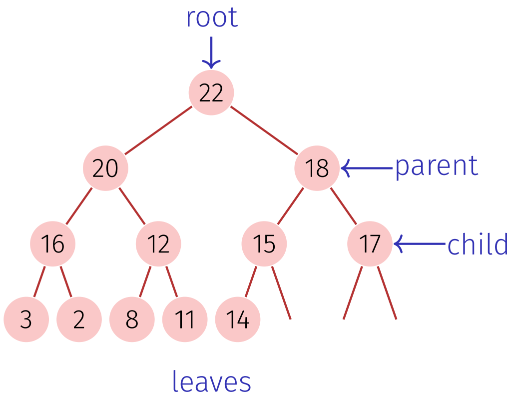


*Heap(data structure), not as in “heap and stack” (memory allocation)

[Max-]Heap*

Binary tree with the following properties

1. complete up to the lowest level
2. Gaps (if any) of the tree in the last level to the right

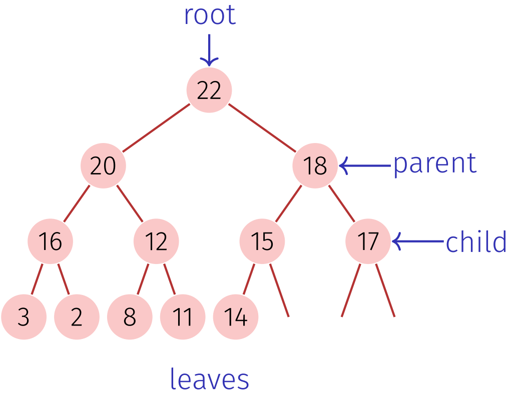


*Heap(data structure), not as in “heap and stack” (memory allocation)

[Max-]Heap*

Binary tree with the following properties

1. complete up to the lowest level
2. Gaps (if any) of the tree in the last level to the right
3. **Heap-Condition:**
   Max-(Min-)Heap: key of a child smaller (greater) than that of the parent node

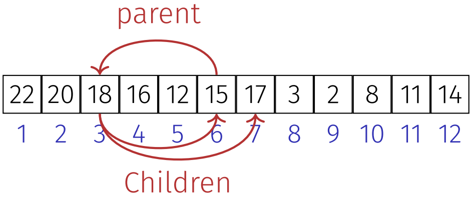


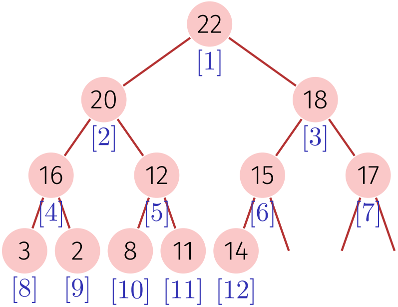
*Heap(data structure), not as in "heap and stack" (memory allocation)

Heap as Array

Tree → Array:

- children(i) $= \{2i, 2i+1\}$
- parent(i) $= \lfloor i/2 \rfloor$





Depends on the starting index[19]

[19] For arrays that start at 0: $\{2i, 2i+1\} \rightarrow \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i-1)/2 \rfloor$

Height of a Heap

What is the height $H(n)$ of Heap with n nodes? On the i-th level of a binary tree there are at most 2^i nodes. Modulo the last level of a heap, all levels are filled with values.

$$H(n) = \min\{h \in \mathbb{N} : \sum_{i=0}^{h-1} 2^i \geq n\}$$

with $\sum_{i=0}^{h-1} 2^i = 2^h - 1$:

$$H(n) = \min\{h \in \mathbb{N} : 2^h \geq n + 1\},$$

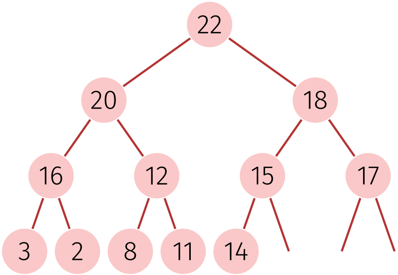
thus

$$H(n) = \lceil \log_2(n + 1) \rceil.$$

Heap in C++

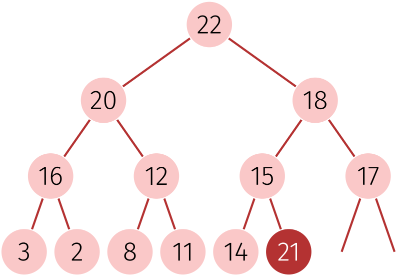
```
class MaxHeap {
  int* keys; // Pointer to first key
  unsigned int capacity; // Length of key array
  unsigned int count; // Keys in use <= capacity
  // Or even better: build on top of std::vector

public:
  MaxHeap(unsigned int initial_capacity):
      keys(new int[initial_capacity]),
      capacity(initial_capacity),
      count(0)
  {}

  void insert(unsigned int key) { ...}
  int remove_max() { ...}
  ...
}
```

Insert



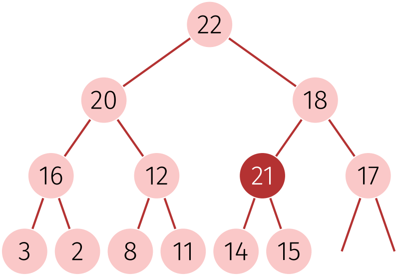
Insert

- Insert new kez at the first free position. Potentially violates the heap property.



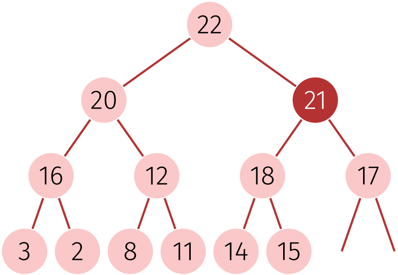
Insert

- Insert new kez at the first free position. Potentially violates the heap property.
- Reestablish heap property: ascend successively



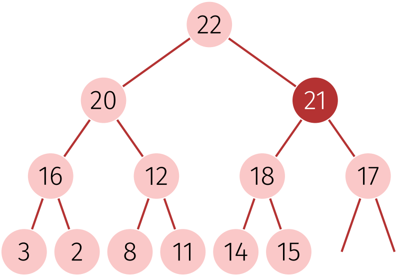
Insert

- Insert new kez at the first free position. Potentially violates the heap property.
- Reestablish heap property: ascend successively



Insert

- Insert new kez at the first free position. Potentially violates the heap property.
- Reestablish heap property: ascend successively
- Worst-case number of operations: $\mathcal{O}(\log n)$

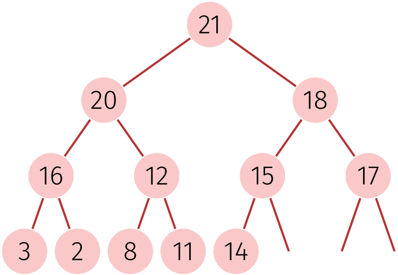


Algorithm Sift-Up(A, m)

Input: Array A with at least m keys and heap structure on $A[1,\ldots,m-1]$
Output: Array A with heap structure on $A[1,\ldots,m]$

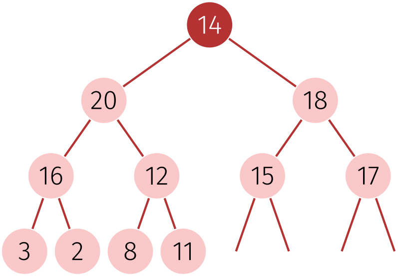
```
v ← A[m] // new key
c ← m // index current node (child)
p ← ⌊c/2⌋ // index parent node
while c > 1 and v > A[p] do
    A[c] ← A[p] // key parent node → key current node
    c ← p // parent node → current node
    p ← ⌊c/2⌋
A[c] ← v // place new key
```

Remove the Maximum



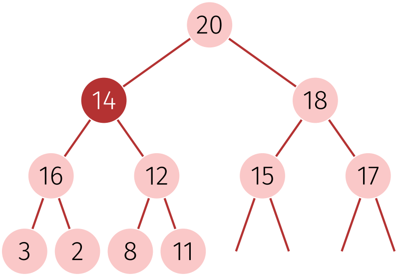
Remove the Maximum

- Replace the maximum by the lower right element



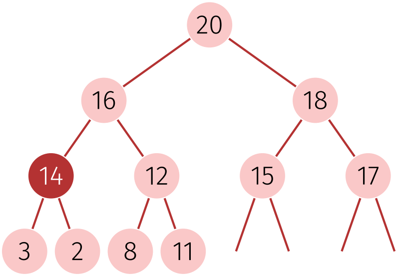
Remove the Maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)



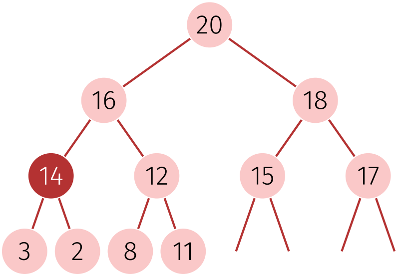
Remove the Maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)



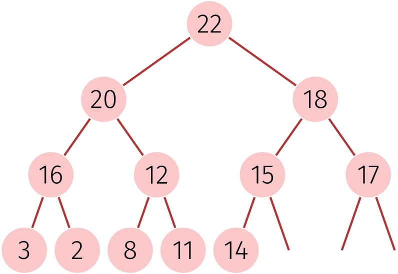
Remove the Maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)
- Worst case number of operations: $\mathcal{O}(\log n)$



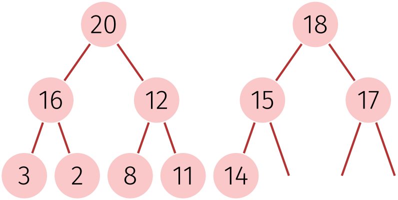
Why this is correct: Recursive heap structure

A heap consists of two heaps:



Why this is correct: Recursive heap structure

A heap consists of two heaps:

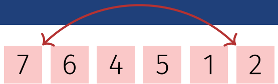


Algorithm SiftDown(A, i, m)

Input: Array A with heap structure for the children of i. Last element m.
Output: Array A with heap structure for i with last element m.

while $2i \leq m$ **do**
    $j \leftarrow 2i$; // j left child
    if $j < m$ and $A[j] < A[j+1]$ **then**
        $j \leftarrow j + 1$; // j right child with greater key
    if $A[i] < A[j]$ **then**
        Swap($A[i], A[j]$)
        $i \leftarrow j$; // keep sinking down
    else
        $i \leftarrow m$; // sift down finished

Sorting Heaps



Let $A[1, ..., n]$ be a heap.

While $n > 1$:

1. *Swap*($A[1]$, $A[n]$)
2. *SiftDown*($A, 1, n - 1$)
3. $n \leftarrow n - 1$

Sorting Heaps

	7	6	4	5	1	2
swap $\Rightarrow$	2	6	4	5	1	7

Let $A[1, ..., n]$ be a heap.

While $n > 1$:

1. *Swap*($A[1]$, $A[n]$)
2. *SiftDown*($A, 1, n - 1$)
3. $n \leftarrow n - 1$

Sorting Heaps

Let $A[1, ..., n]$ be a heap.

While $n > 1$:

1. *Swap*($A[1]$, $A[n]$)
2. *SiftDown*($A, 1, n - 1$)
3. $n \leftarrow n - 1$

		7	6	4	5	1	2
swap	$\Rightarrow$	2	6	4	5	1	7
siftDown	$\Rightarrow$	6	5	4	2	1	7

Sorting Heaps

Let $A[1, ..., n]$ be a heap.

While $n > 1$:

1. *Swap*$(A[1], A[n])$
2. *SiftDown*$(A, 1, n-1)$
3. $n \leftarrow n - 1$

		7	6	4	5	1	2
swap	$\Rightarrow$	2	6	4	5	1	7
siftDown	$\Rightarrow$	6	5	4	2	1	7
swap	$\Rightarrow$	1	5	4	2	6	7

Sorting Heaps

Let $A[1, ..., n]$ be a heap.

While $n > 1$:

1. *Swap*$(A[1], A[n])$
2. *SiftDown*$(A, 1, n-1)$
3. $n \leftarrow n - 1$

		7	6	4	5	1	2
swap	$\Rightarrow$	2	6	4	5	1	7
siftDown	$\Rightarrow$	6	5	4	2	1	7
swap	$\Rightarrow$	1	5	4	2	6	7
siftDown	$\Rightarrow$	5	4	2	1	6	7
swap	$\Rightarrow$	1	4	2	5	6	7
siftDown	$\Rightarrow$	4	1	2	5	6	7
swap	$\Rightarrow$	2	1	4	5	6	7
siftDown	$\Rightarrow$	2	1	4	5	6	7
swap	$\Rightarrow$	1	2	4	5	6	7

Heap creation

Observation: Every leaf of a heap is trivially a correct heap.

Consequence:

Heap creation

Observation: Every leaf of a heap is trivially a correct heap.

Consequence: Induction from below!

Algorithm HeapSort(A, n)

```
Input:  Array A with length n.
Output: A sorted.
// Build the heap
for i ← n/2 downto 1 do
    SiftDown(A, i, n)
// Now A is a heap
for i ← n downto 2 do
    Swap(A[1], A[i])
    SiftDown(A, 1, i − 1)
// Now A is sorted.
```

Analysis: sorting a heap

SiftDown traverses at most $\log n$ nodes. For each node, 2 key comparisons.
$\Rightarrow$ sorting a heap costs $2 \log n$ comparisons in the worst case.

Number of memory movements while sorting a heap also $\mathcal{O}(n \log n)$.

Analysis: creating a heap

Calls to *SiftDown*: $n/2$.

Thus number of comparisons and movements: $v(n) \in \mathcal{O}(n \log n)$.

But mean length of the sift-down paths is much smaller:

We use that $h(n) = \lceil \log_2 n + 1 \rceil = \lfloor \log_2 n \rfloor + 1$ für $n > 0$

$$v(n) = \sum_{l=0}^{\lfloor \log_2 n \rfloor} \underbrace{2^l}_{\text{number heaps on level l}} \cdot (\underbrace{\lfloor \log_2 n \rfloor + 1 - l}_{\text{height heaps on level l}} - 1) = \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^{\lfloor \log_2 n \rfloor - k} \cdot k$$

$$= 2^{\lfloor \log_2 n \rfloor} \cdot \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{k}{2^k} \leq n \cdot \sum_{k=0}^{\infty} \frac{k}{2^k} \leq n \cdot 2 \in \mathcal{O}(n)$$

with $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad (0 < x < 1)$ and $s(\frac{1}{2}) = 2$

Disadvantages

Heapsort: $\mathcal{O}(n \log n)$ Comparisons and movements.

Disadvantages of heapsort?

Disadvantages

Heapsort: $\mathcal{O}(n \log n)$ Comparisons and movements.

Disadvantages of heapsort?

- Missing locality: heapsort jumps around in the sorted array (negative cache effect).

Disadvantages

Heapsort: $\mathcal{O}(n \log n)$ Comparisons and movements.

Disadvantages of heapsort?

- Missing locality: heapsort jumps around in the sorted array (negative cache effect).
- Two comparisons required before each necessary memory movement.