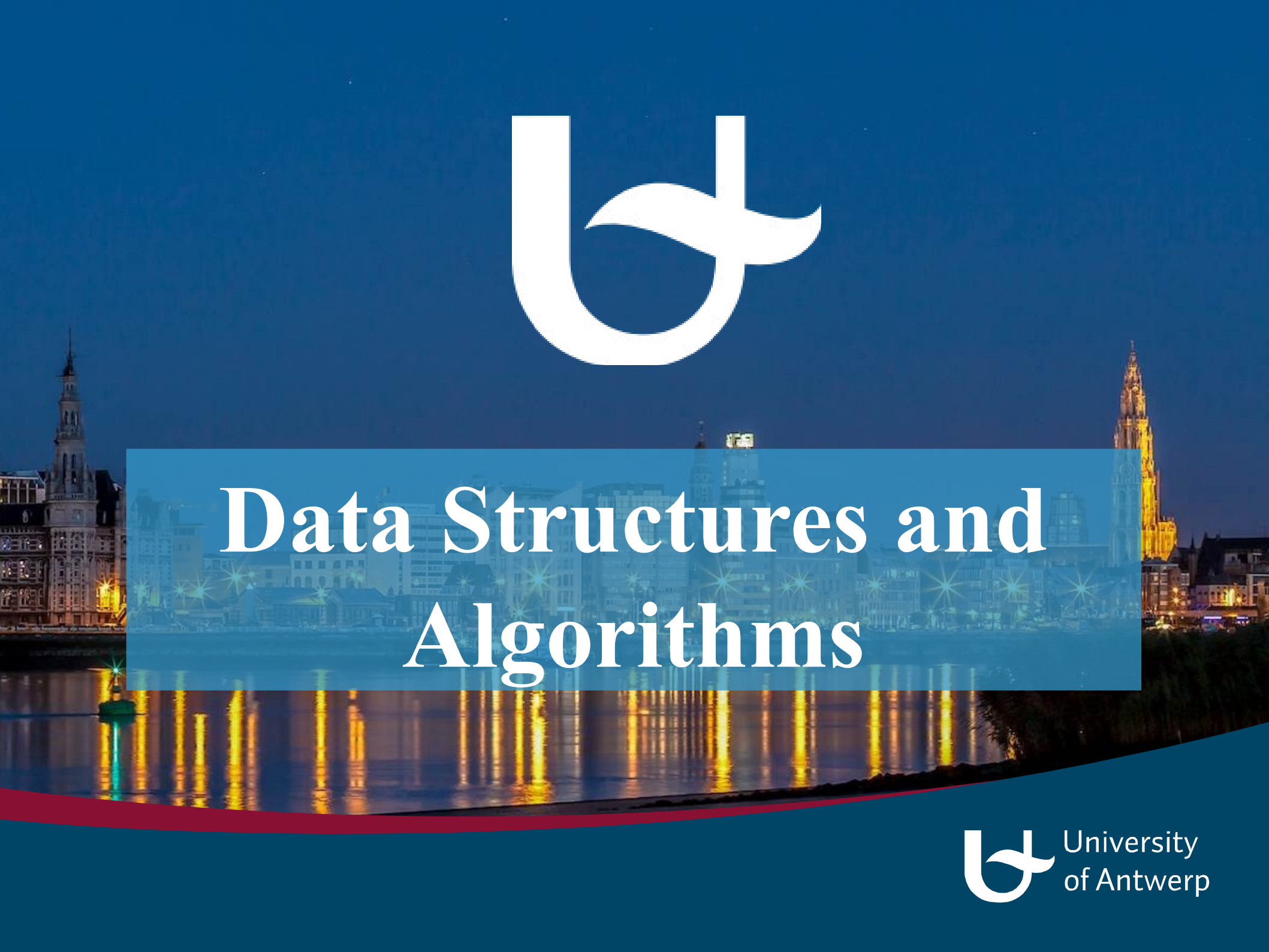


Data Structures and Algorithms

University of Antwerp

Trees

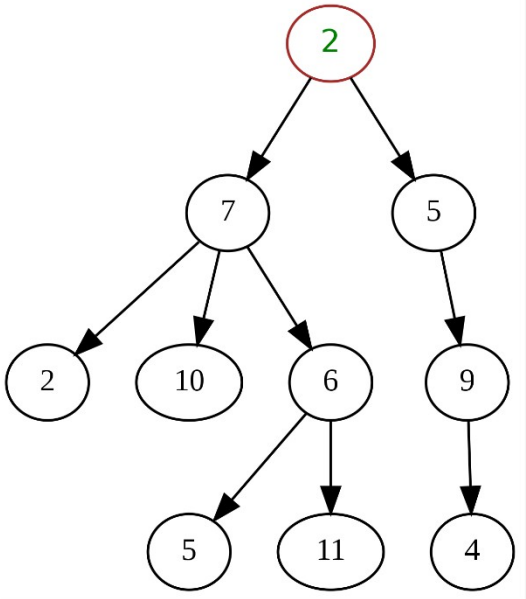
Lecturer: Dr. Ali Anwar

Objectives

- Introduction of
  - Trees
    - Terminology, ADT, Traversals
  - Binary Trees
    - Arithmetic Expression Tree
    - Decision Tree
  - Binary Search Trees
    - Ordered Maps, Search Tables
  - Heaps
    - Upheap, Downheap, Heap-Sort, Merging Two Heaps

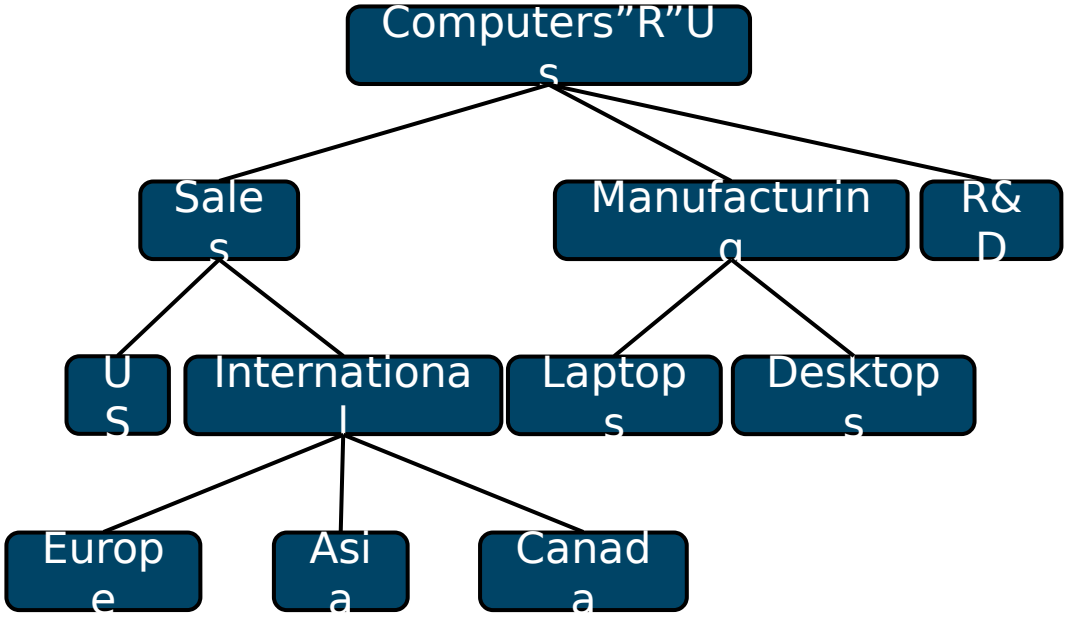
Part 1

Trees



What is a Tree?

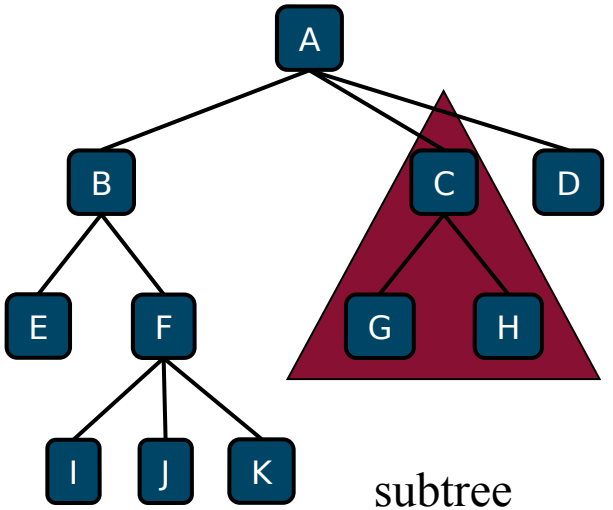
- In computer science, a **tree** is an abstract model of a hierarchical structure
- A tree consists of **nodes** with a **parent-child** relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Tree Terminology

- **Root**: node without parent (A)
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors of a node**: parent, grandparent, grand-grandparent, etc.
- **Depth of a node**: number of ancestors
- **Height of a tree**: maximum depth of any node (3)
- **Descendant of a node**: child, grandchild, grand-grandchild, etc.

- **Subtree**: tree consisting of a node and its descendants



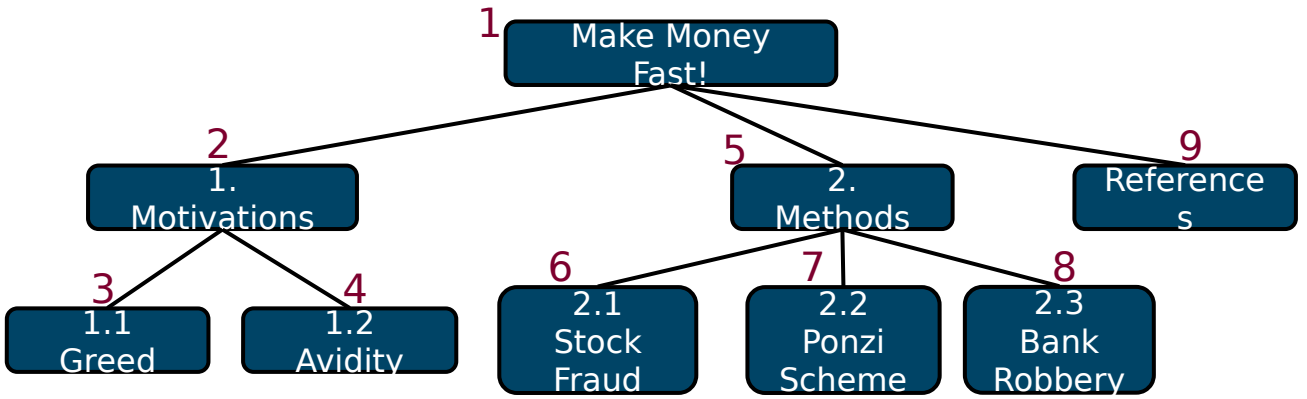
Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer *size*()
  - boolean *empty*()
- Accessor methods:
  - position *root*()
  - list<position> *positions*()
- Position-based methods:
  - position p.*parent*()
  - list<position> p.*children*()
- Query methods:
  - boolean p.*isRoot*()
  - boolean p.*isExternal*()
- Additional update methods may be defined by data structures implementing the Tree ADT

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

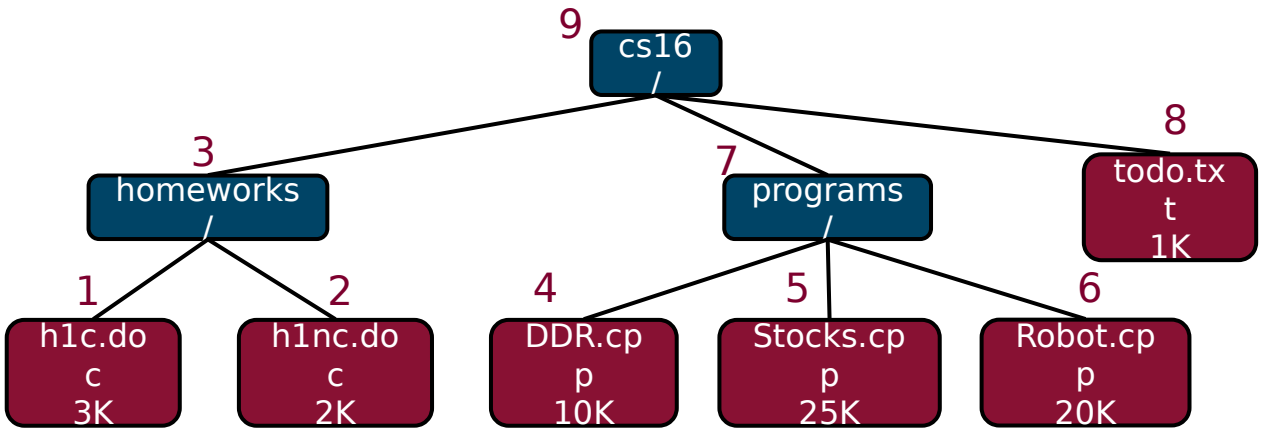
Algorithm ***preOrder*(*v*)**
    ***visit*(*v*)**
    for each child ***w*** of ***v***
        preorder **(*w*)**



Postorder Traversal

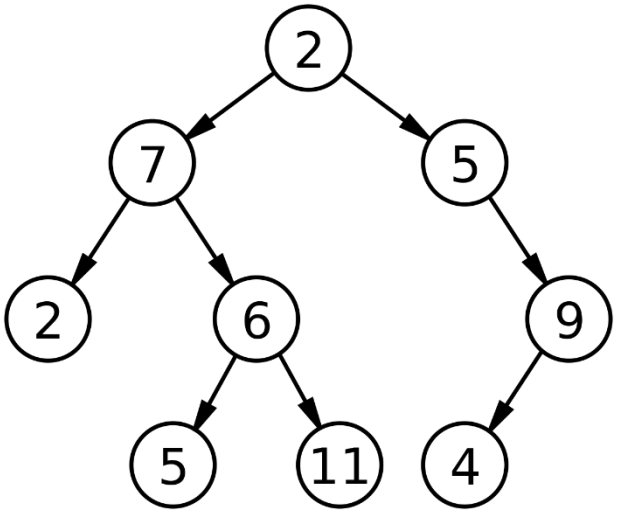
- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm ***postOrder(v)***
    for each child ***w*** of ***v***
        postOrder **(*w*)**
            visit**(*v*)**



Part 2

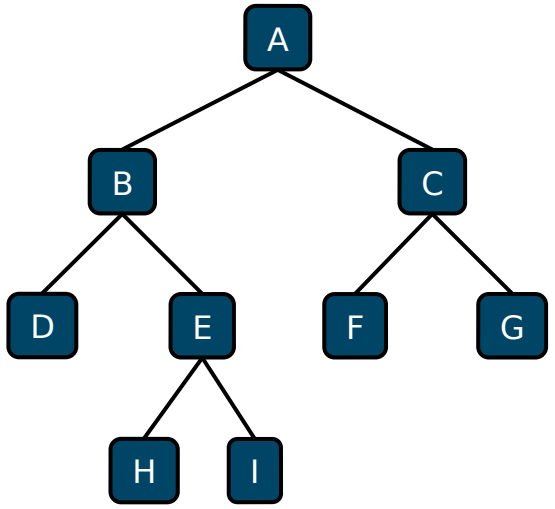
Binary Trees



Binary Trees

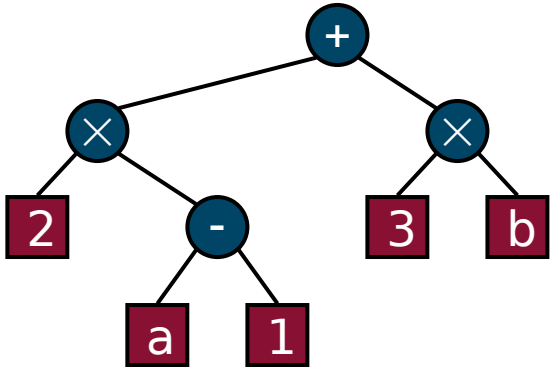
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



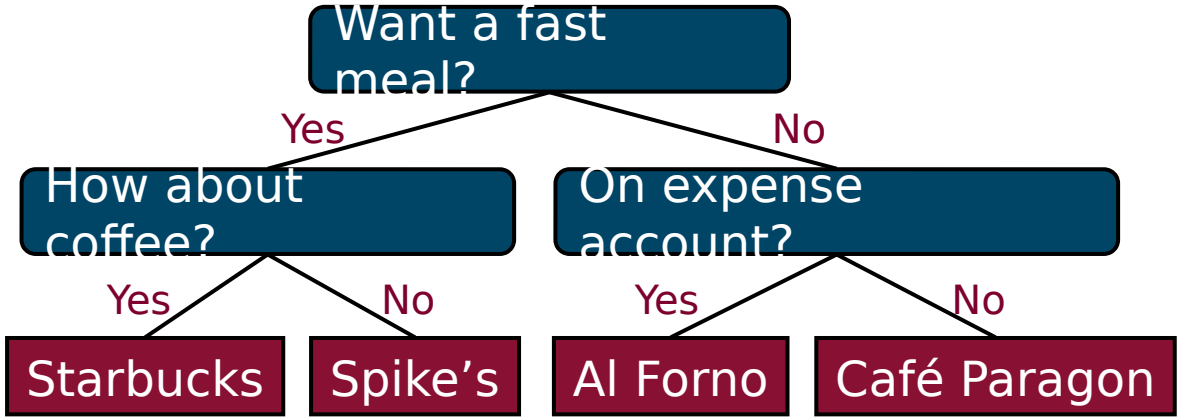
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- *Example*: arithmetic expression tree for the expression (2 × (a - 1) + (3 × b))



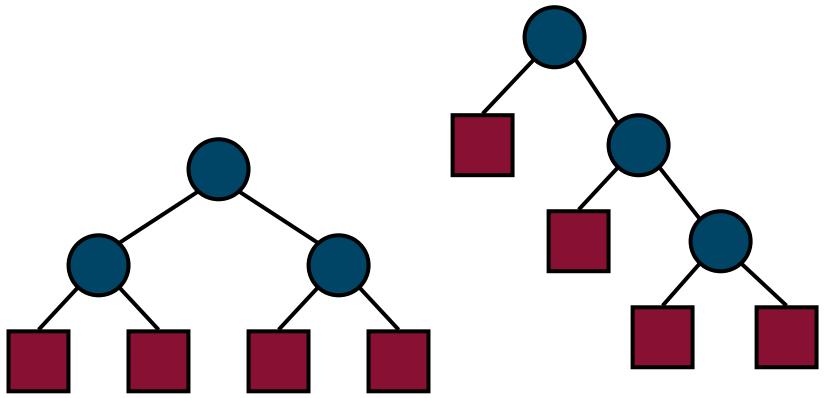
Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- *Example*: dining decision



Properties of Proper Binary Trees

- Notation
  - ***n*** number of nodes
  - ***e*** number of external nodes
  - ***i*** number of internal nodes
  - ***h*** height



- Properties:
  - ***e = i + 1***
  - ***n = 2e - 1***
  - ***h ≤ i***
  - ***h ≤ (n - 1)/2***
  - ***e ≤ 2h***
  - ***h ≥ log2 e***
  - ***h ≥ log2 (n + 1) - 1***

Binary Tree ADT

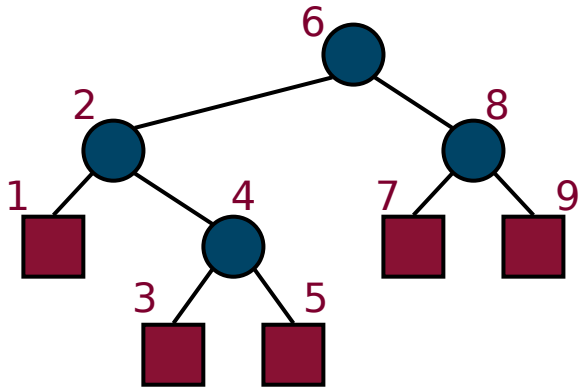
- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position p.*left*()
  - position p.*right*()

- Update methods may be defined by data structures implementing the BinaryTree ADT
- Proper binary tree: Each node has either 0 or 2 children

Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree

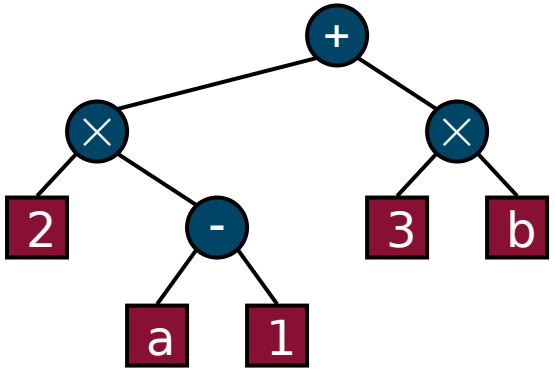
```
Algorithm inOrder(v)
    if ¬ v.isExternal()
        inOrder(v.left())
    visit(v)
    if ¬ v.isExternal()
        inOrder(v.right())
```



Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

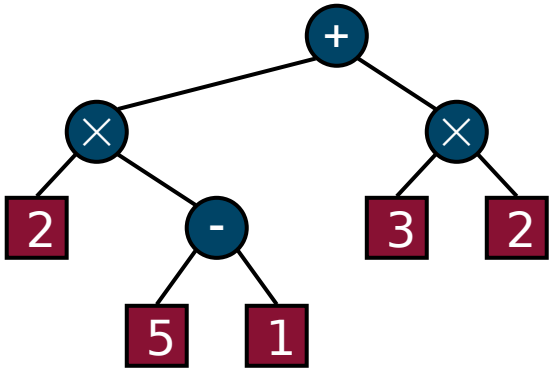
```
Algorithm printExpression(v)
    if ¬v.isExternal()
        print("(")
        inOrder(v.left())
    print(v.element())
    if ¬v.isExternal()
        inOrder(v.right())
        print (")")
```



$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expressions

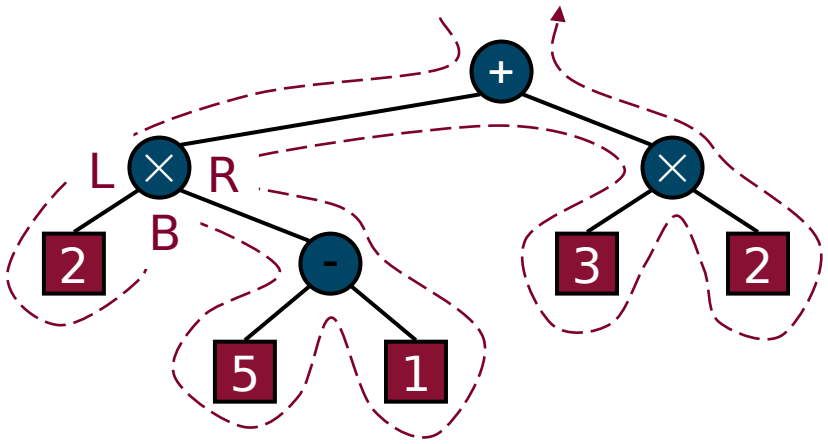
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees



Algorithm *evalExpr*(*v*)
    if *v.isExternal*()
        return *v.element*()
    else
        x ← *evalExpr*(*v.left*())
        y ← *evalExpr*(*v.right*())
        ◊ ← operator stored at *v*
        return *x* ◊ *y*

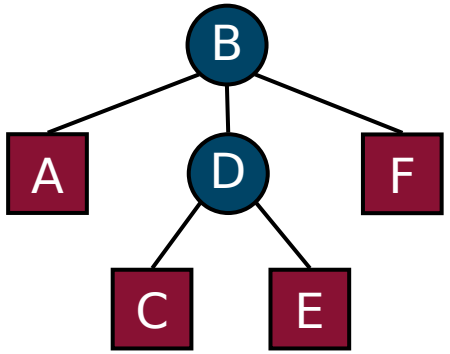
Euler Tour Traversal

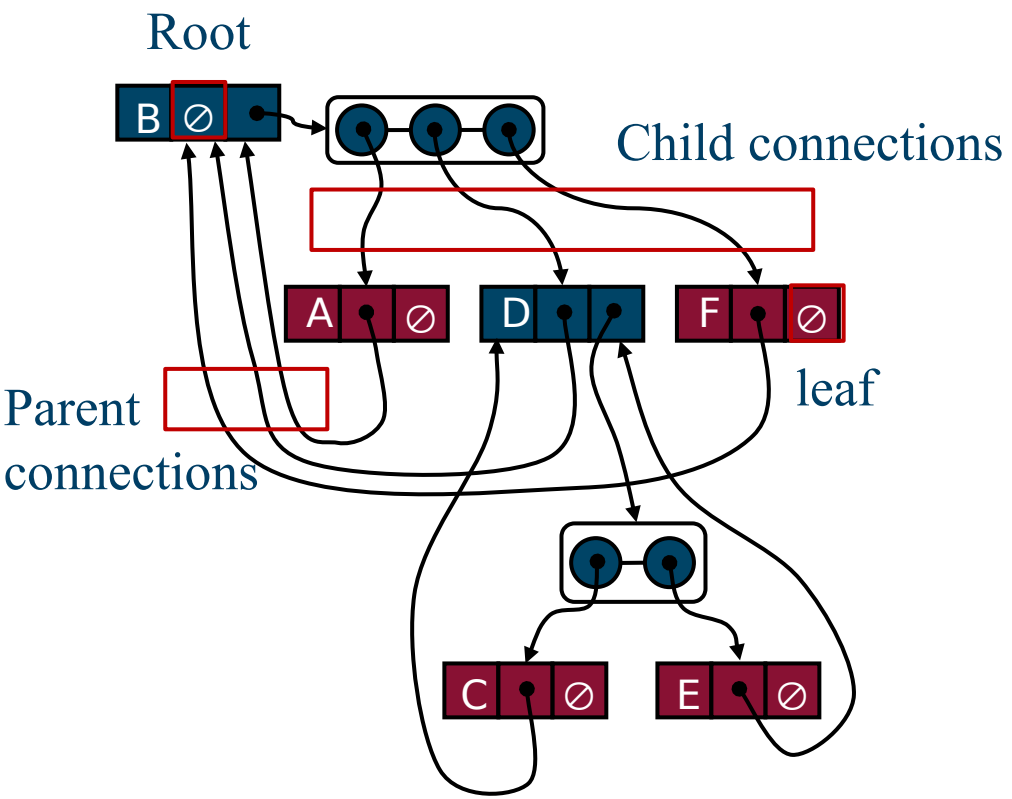
- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



Linked Structure for Trees

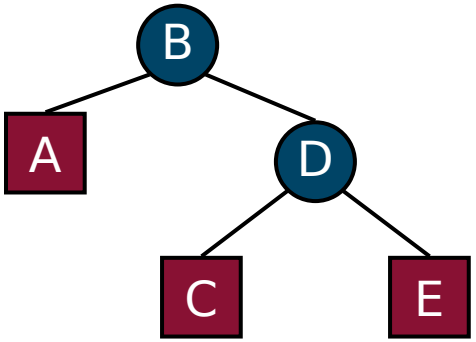
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT

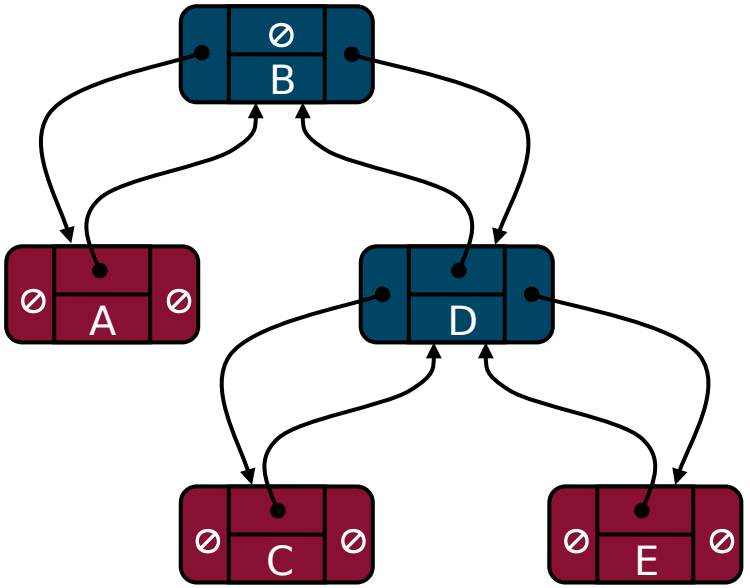




Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT

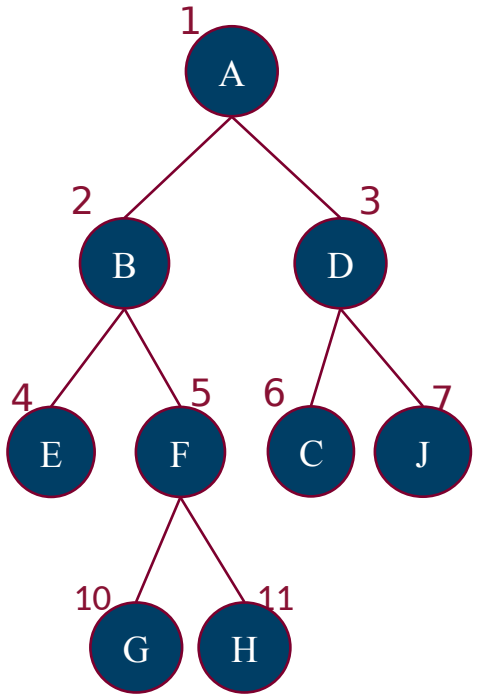




Array-Based Representation of Binary Trees

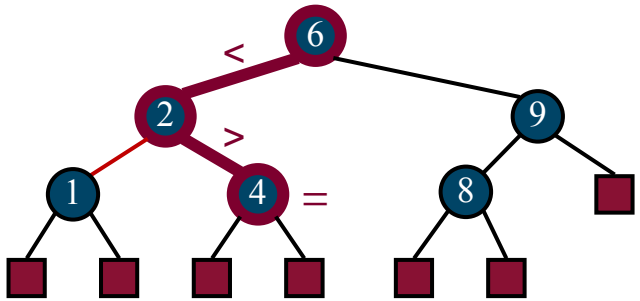
- Nodes are stored in an array A

- Node v is stored at $A[\text{rank}(v)]$
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2 · rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2 · rank(parent(node)) + 1



Part 3

Binary Search Trees

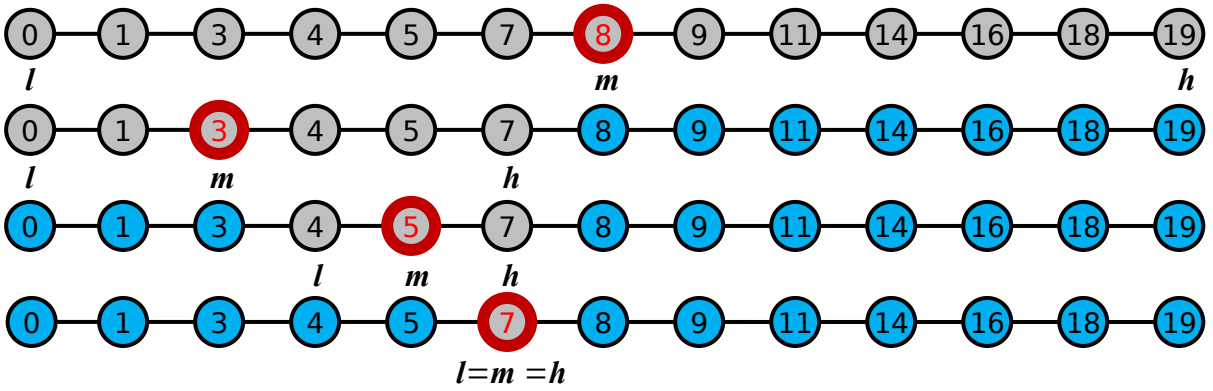


Ordered Maps

- Keys come from a total order
- New operations:
  - Each returns an *iterator* to an entry:
  - *firstEntry*(): smallest key in the map
  - *lastEntry*(): largest key in the map
  - *floorEntry*(k): largest key $\leq k$
  - *ceilingEntry*(k): smallest key $\geq k$
  - All return *end* if the map is empty

Binary Search

- Binary search can perform operations *get*, *floorEntry* and *ceilingEntry* on an ordered map implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: *find*(7)



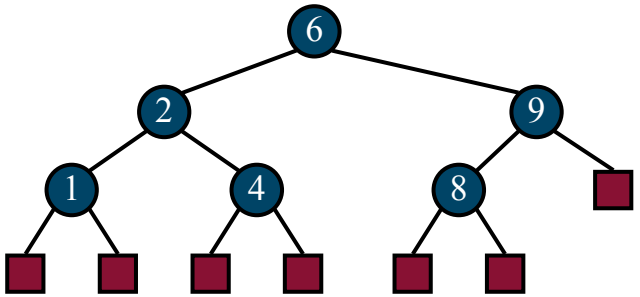
Search Tables

- A search table is an ordered map implemented by means of a sorted sequence
  - We store the items in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - *get*, *floorEntry* and *ceilingEntry* take O(log n) time, using binary search
  - *set* takes O(n) time since in the worst case we have to shift $n/2$ items to make room for the new item
  - *erase* take O(n) time since in the worst case we have to shift $n/2$ items to compact the items after the removal
- The lookup table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Binary Search Trees

- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have:
    $$\text{key}(u) \leq \text{key}(v) \leq \text{key}(w)$$
- External nodes do not store items

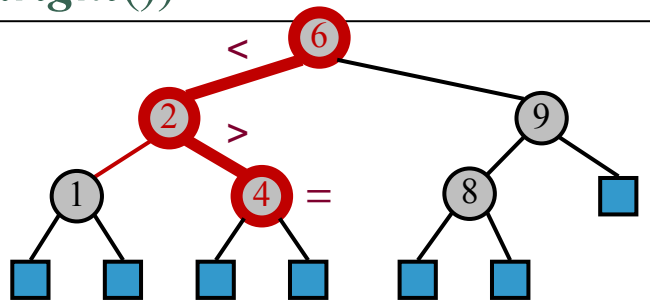
- An inorder traversal of a binary search trees visits the keys in increasing order



Search

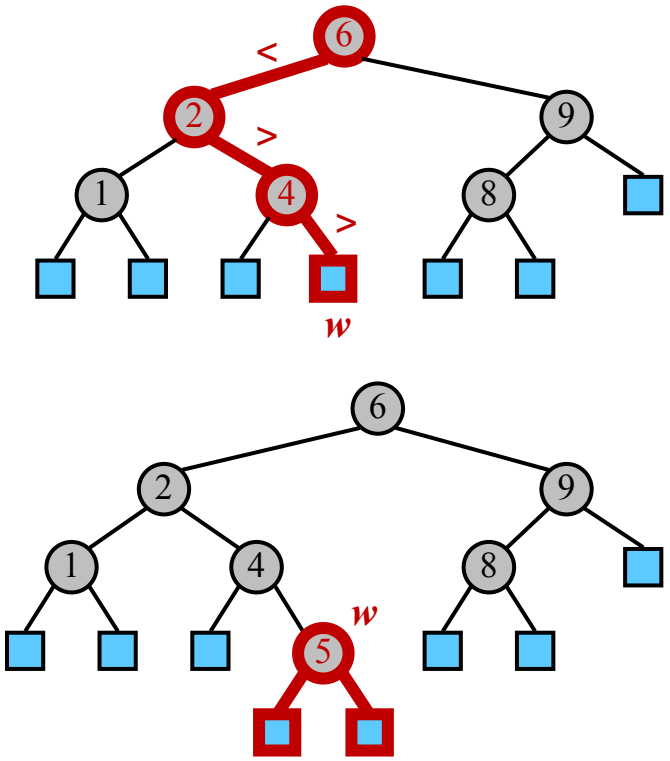
- To search for a key *k*, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf, the key is not found
- Example: *get*(4):
  - Call *TreeSearch*(4,root)
- The algorithms for *floorEntry* and *ceilingEntry* are similar

```
Algorithm TreeSearch(k, v)
        if v.isExternal ()
            return v
    if k < v.key()
            return TreeSearch(k,
    v.left())
    else if k = v.key()
            return v
    else { k > v.key() }
            return TreeSearch(k,
    v.right())
```



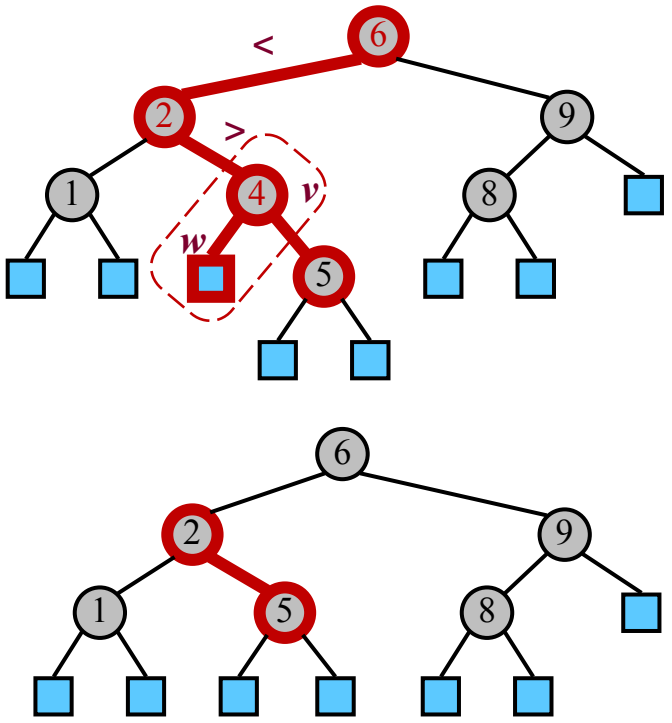
Insertion

- To perform operation *put*(*k*, *o*), we search for key *k* (using TreeSearch)
- Assume *k* is not already in the tree, and let *w* be the leaf reached by the search
- We insert *k* at node *w* and expand *w* into an internal node
- Example: insert 5



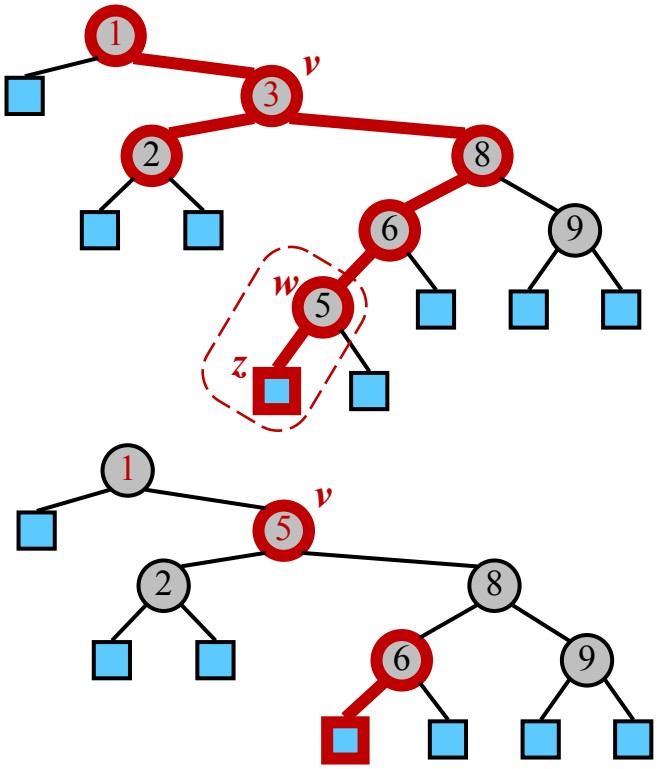
Deletion

- To perform operation *erase*(k), we search for key k
- Assume key k is in the tree, and let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation *removeExternal*(w), which removes w and its parent
- Example: remove 4



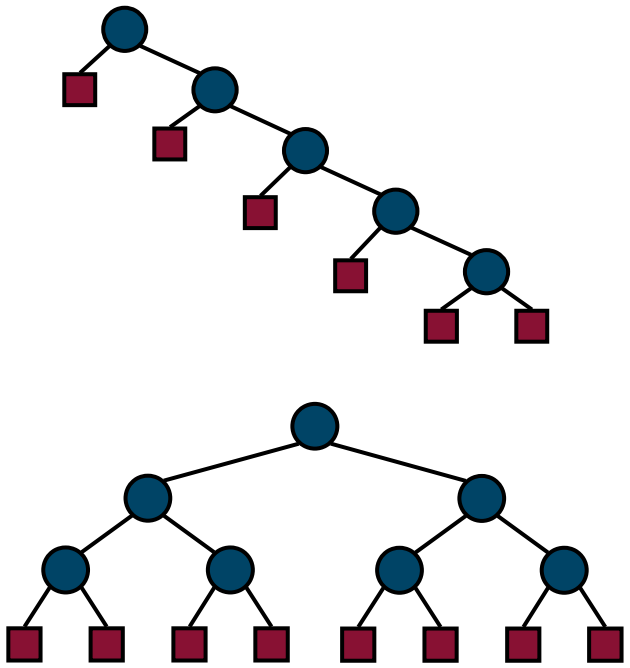
Deletion (cont.)

- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy key(w) into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation *removeExternal*(z)
- Example: remove 3



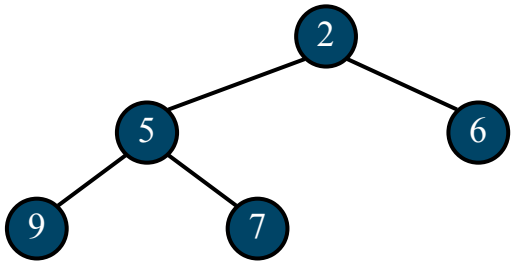
Performance

- Consider an ordered map with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods *get*, *floorEntry*, *ceilingEntry*, *put* and *erase* take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case



Part 4

Heaps



Recall Priority Queue ADT

- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - *insert*(*e*) inserts an entry e
  - *removeMin*() removes the entry with smallest key
- Additional methods
  - *min*() returns (but does not remove) an entry with smallest key
  - *size*(), *empty*()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Recall PQ Sorting

- We use a priority queue
  - Insert the elements with a series of *insert* operations
  - Remove the elements in sorted order with a series of *removeMin* operations
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives selection-sort: $O(n^2)$ time
  - Sorted sequence gives insertion-sort: $O(n^2)$ time
- Can we do better?

Algorithm *PQ-Sort*(*S*, *C*)
Input sequence *S*, comparator *C* for the elements of *S*
Output sequence *S* sorted in increasing order according to *C*
P ← priority queue with comparator *C*
while ¬*S.empty* ()
e ← *S.front*();
S.eraseFront()
P.insert (*e*, ⊘)
while ¬*P.empty*()
e ← *P.removeMin*()
S.insertBack(*e*)

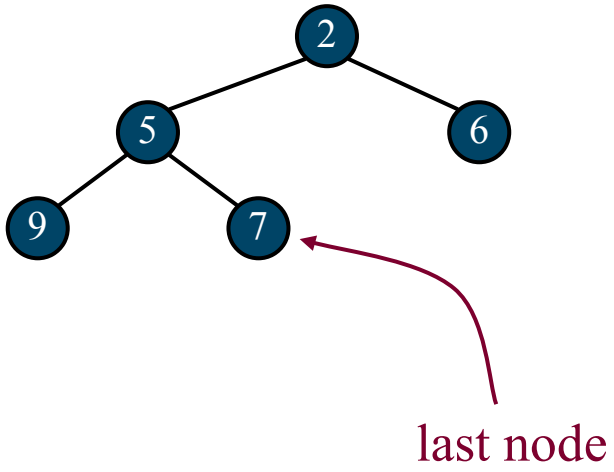
Heaps

- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
- Heap-Order: for every internal node v other than the root,

$$\text{key}(v) \geq \text{key}(\text{parent}(v))$$

- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
  - at depth $h - 1$, the internal nodes are to the left of the external nodes

- The last node of a heap is the rightmost node of maximum depth

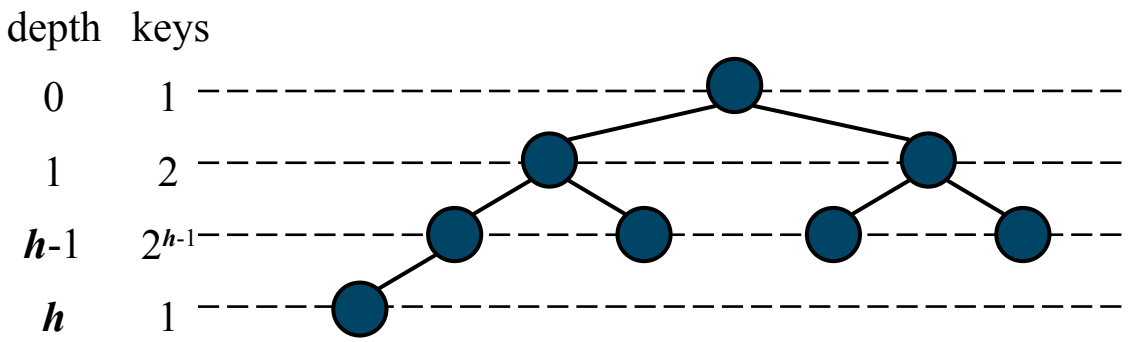


Height of a Heap

- Theorem: A heap storing n keys has height O(log n)

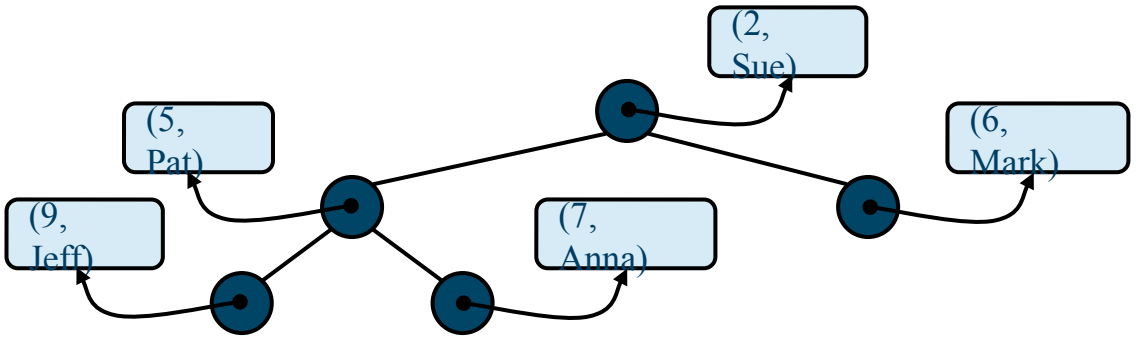
  Proof: (we apply the complete binary tree property)

  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h - 1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$



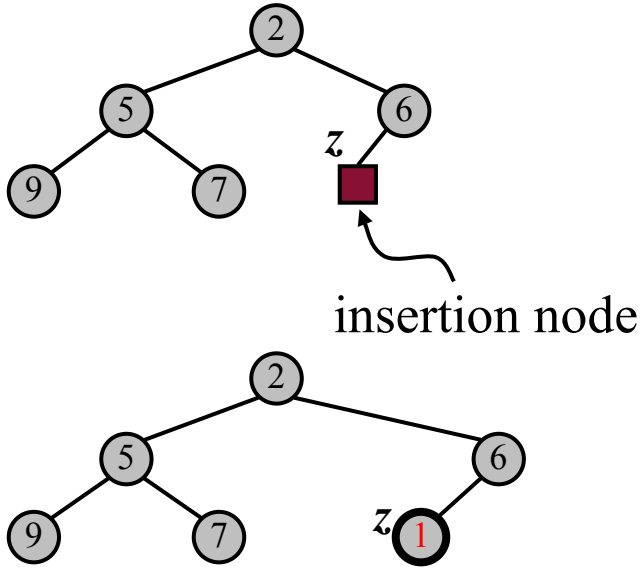
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node



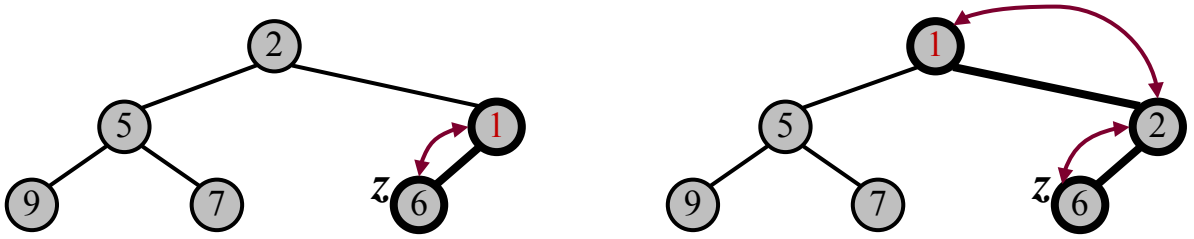
Insertion into a Heap

- Method *insertItem* of the priority queue ADT corresponds to the insertion of a key *k* to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node *z* (the new last node)
  - Store *k* at *z*
  - Restore the heap-order property (discussed next)



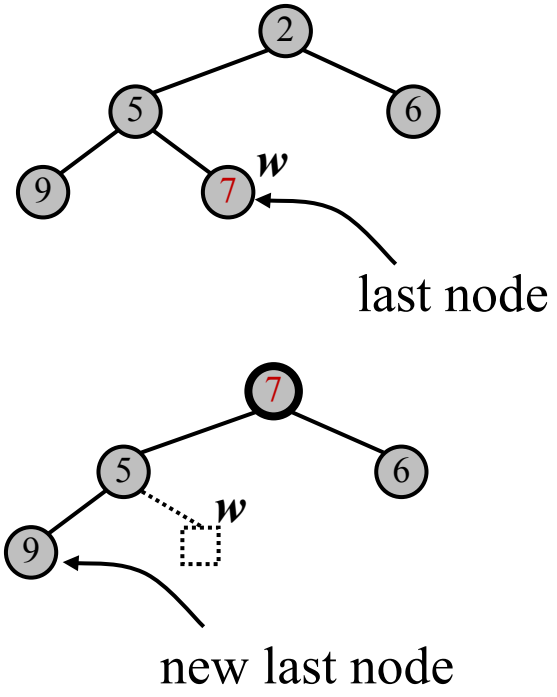
Upheap

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height O(log n), upheap runs in O(log n) time



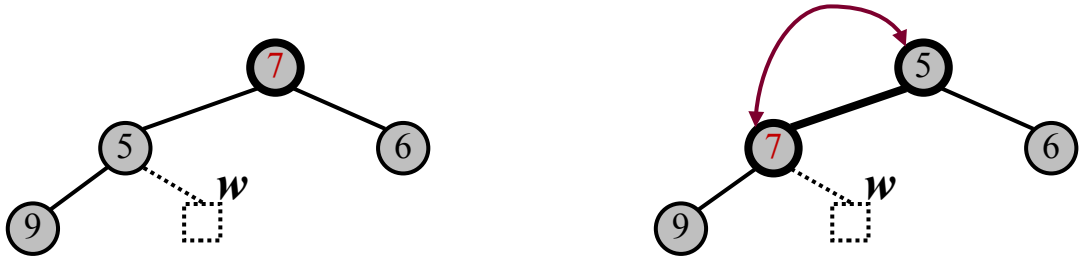
Removal from a Heap

- Method *removeMin* of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node *w*
  - Remove *w*
  - Restore the heap-order property (discussed next)



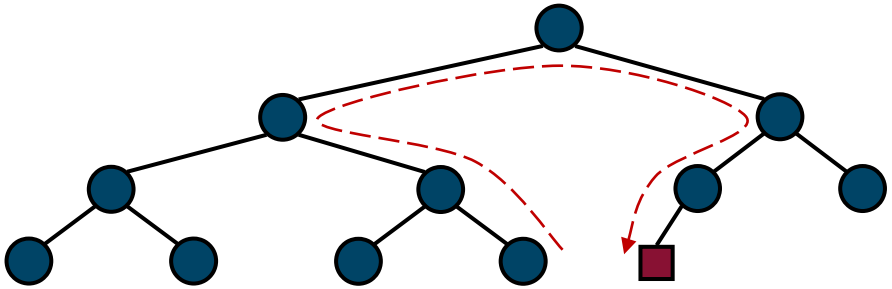
Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height O(log n), downheap runs in O(log n) time



Updating the Last Node

- The insertion node can be found by traversing a path of O(log *n*) nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal

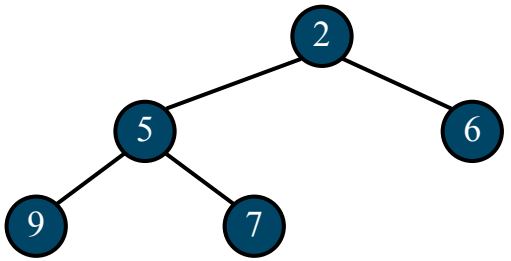


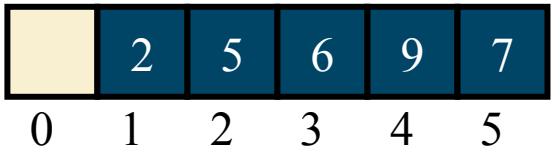
Heap-Sort

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods *insert* and *removeMin* take $O(\log n)$ time
  - methods *size*, *empty*, and *min* take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Vector-based Heap Implementation

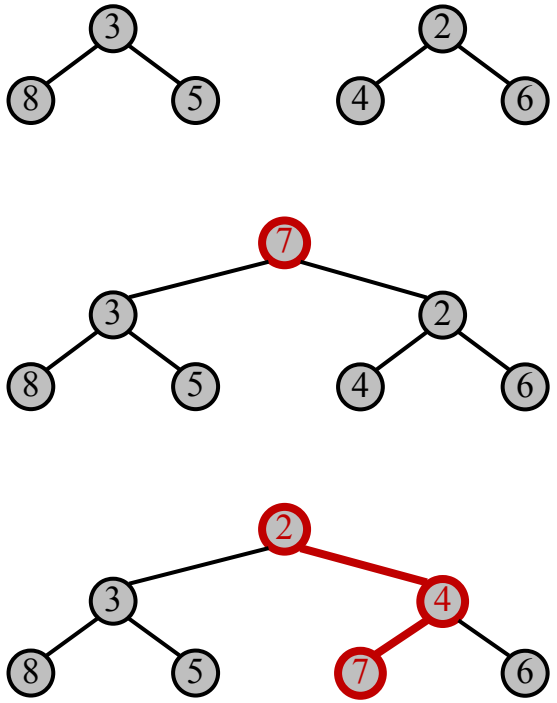
- We can represent a heap with n keys by means of a vector of length $n + 1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- The cell of at rank 0 is not used
- Operation insert corresponds to inserting at rank $n + 1$
- Operation *removeMin* corresponds to removing at rank n
- Yields in-place heap-sort





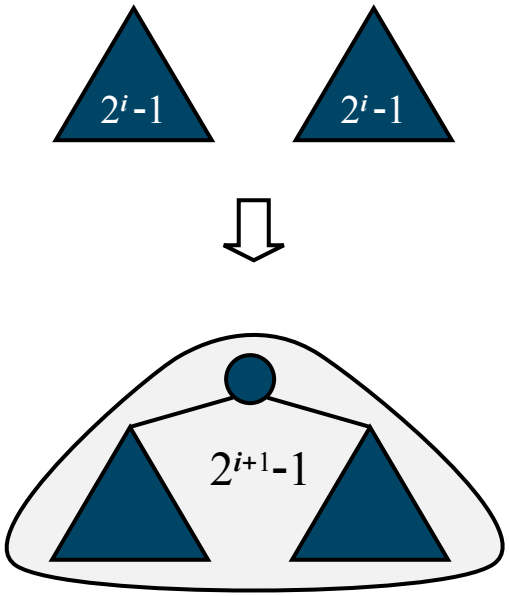
Merging Two Heaps

- We are given two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property

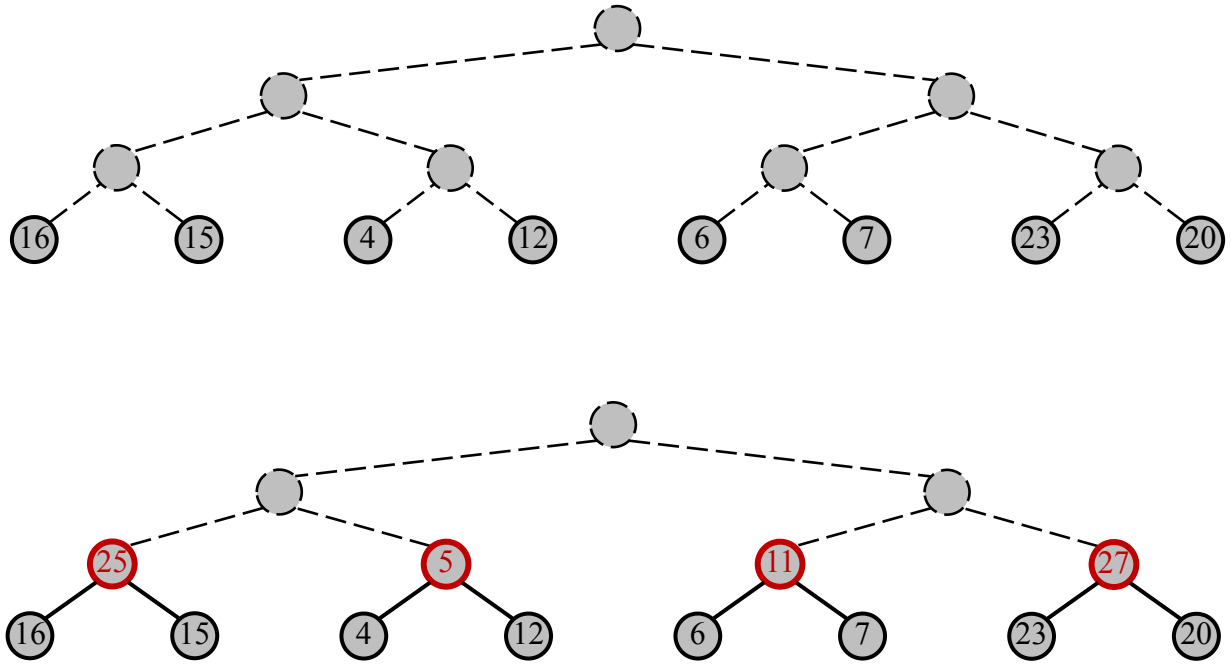


Bottom-up Heap Construction

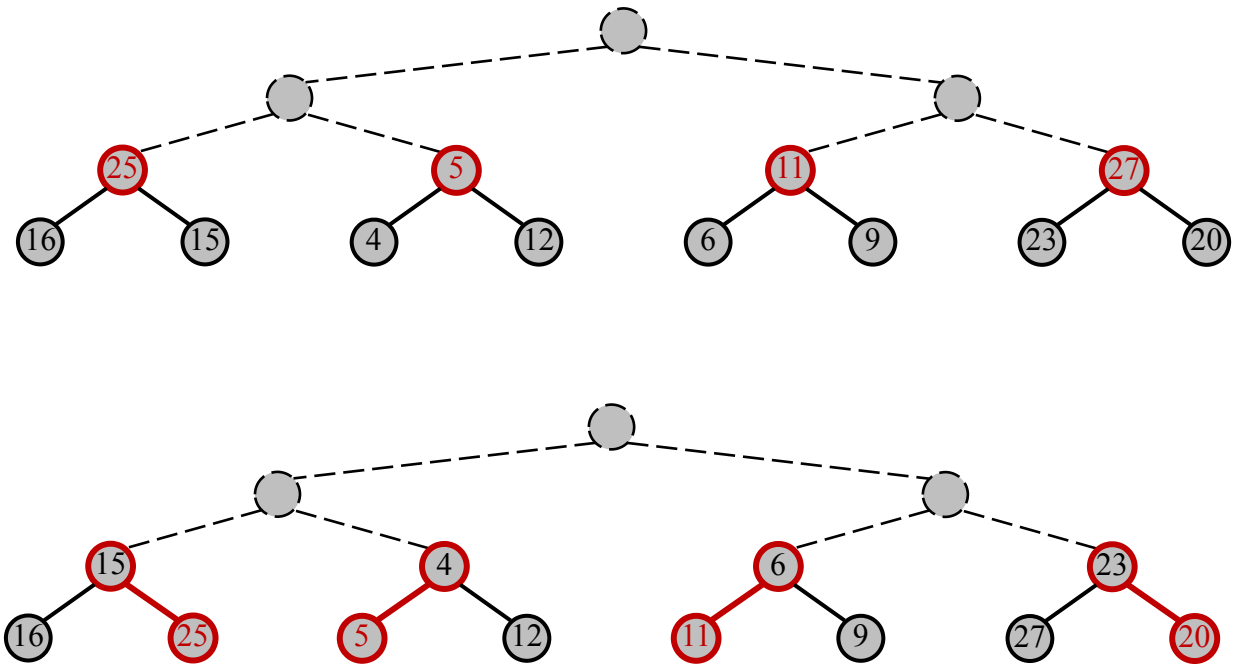
- We can construct a heap storing n given keys in using a bottom-up construction with log n phases
- In phase i, pairs of heaps with 2^i -1 keys are merged into heaps with 2^{i+1}-1 keys



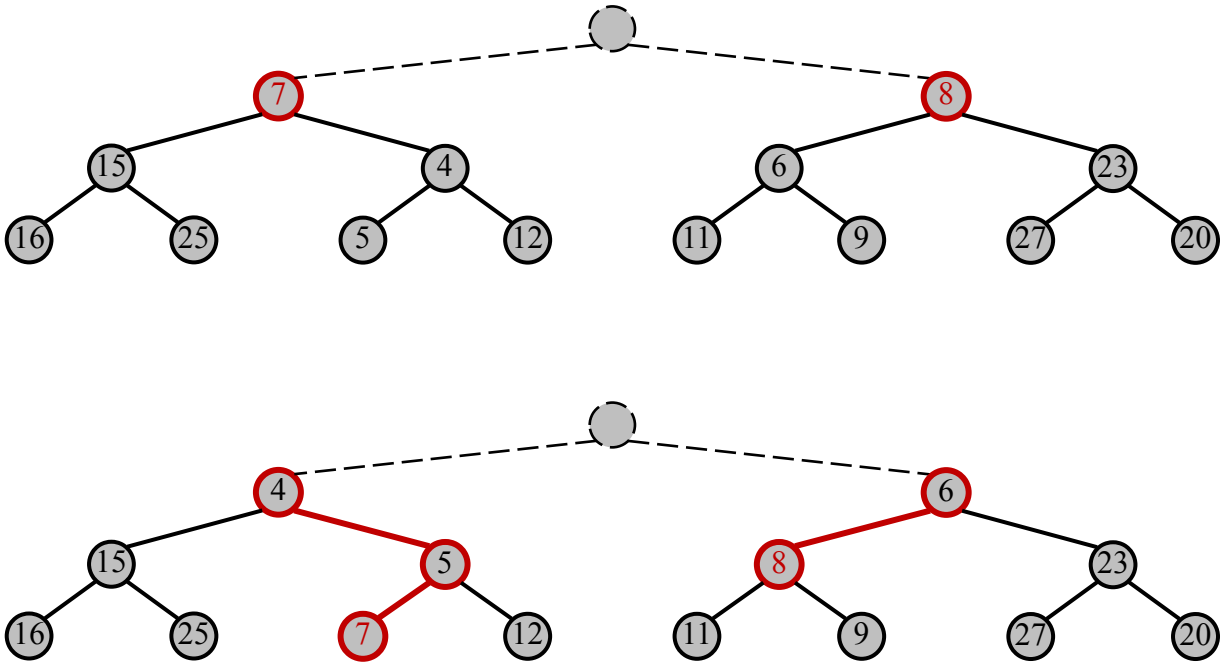
Example



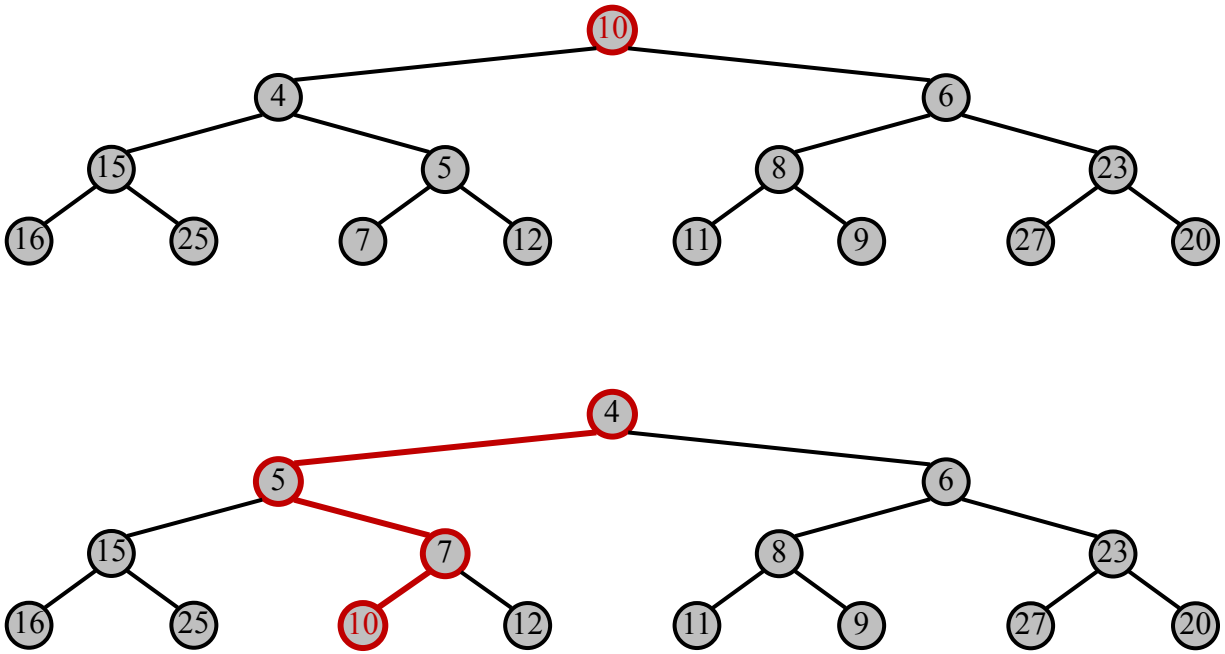
Example (cont.)



Example (cont.)



Example (end)



Analysis

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is O(n)
- Thus, bottom-up heap construction runs in O(n) time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort