COMP20010: Algorithms and Imperative Programming

Lecture 1

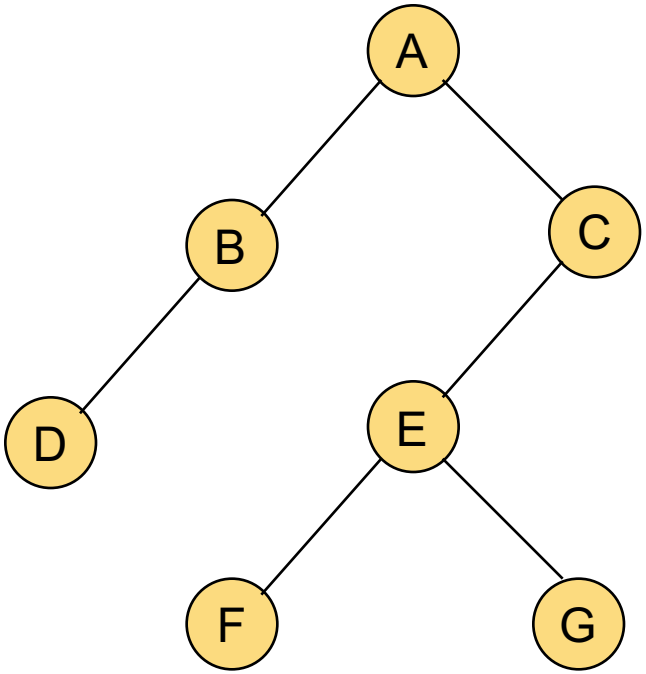
Trees

Lecture outline

- Motivation
- Definitions
- Ordered trees
- Generic methods for tree operations
- Tree traversal (preorder, postorder, inorder)
- Binary trees – tree traversal

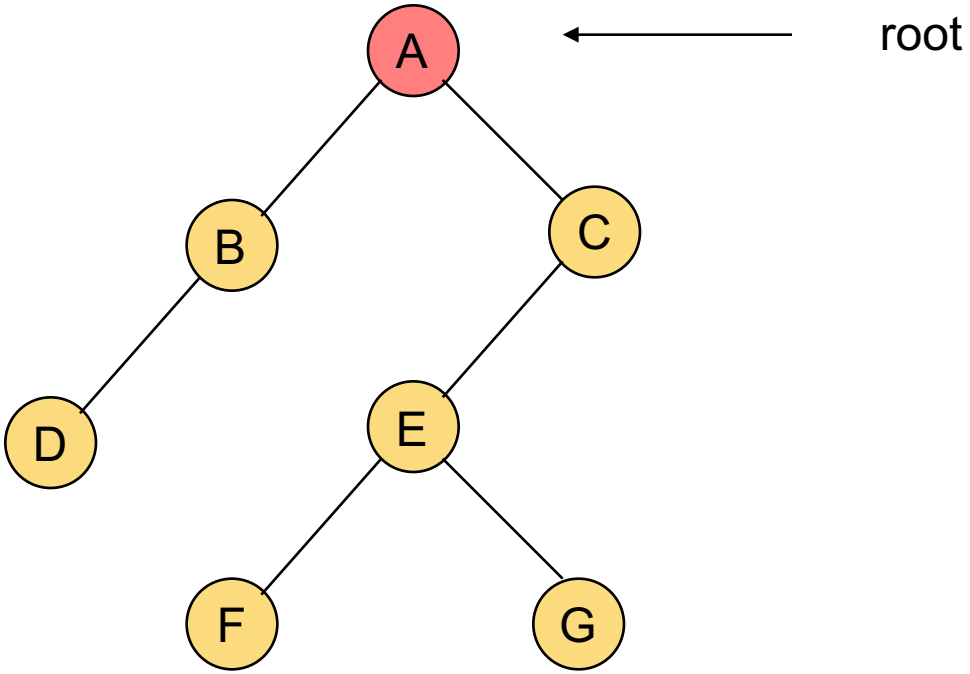
Definitions

- An abstract data type for hierarchical storage of information;



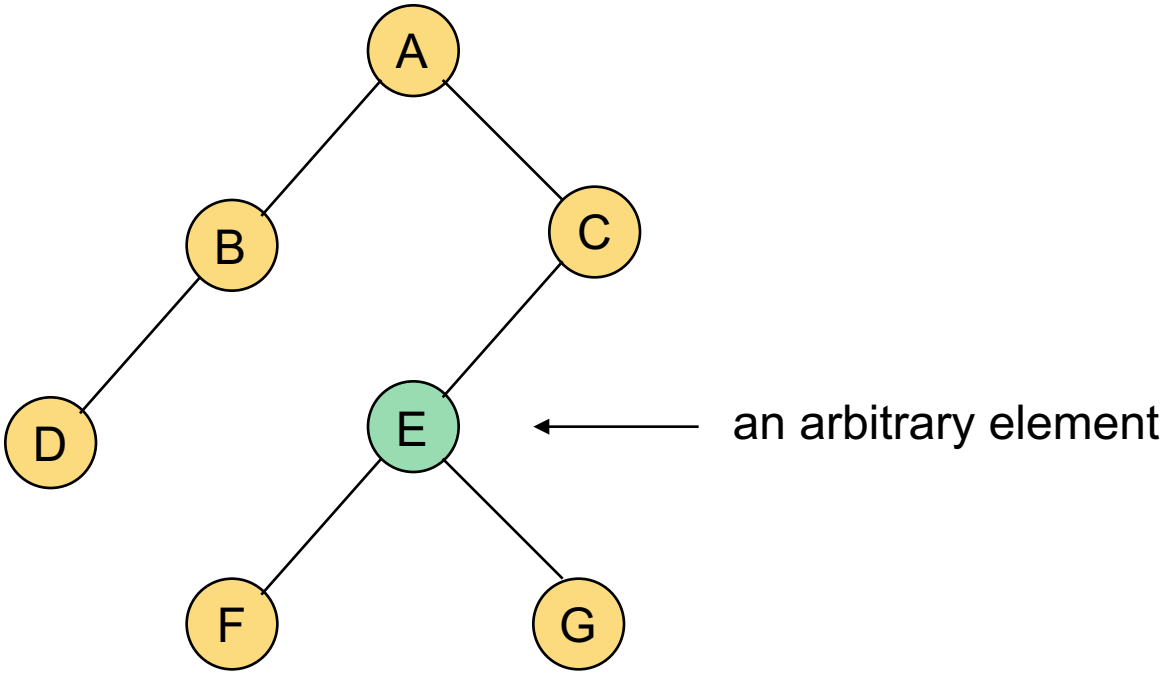
Definitions

- The top element of a tree is referred to as the root of a tree;



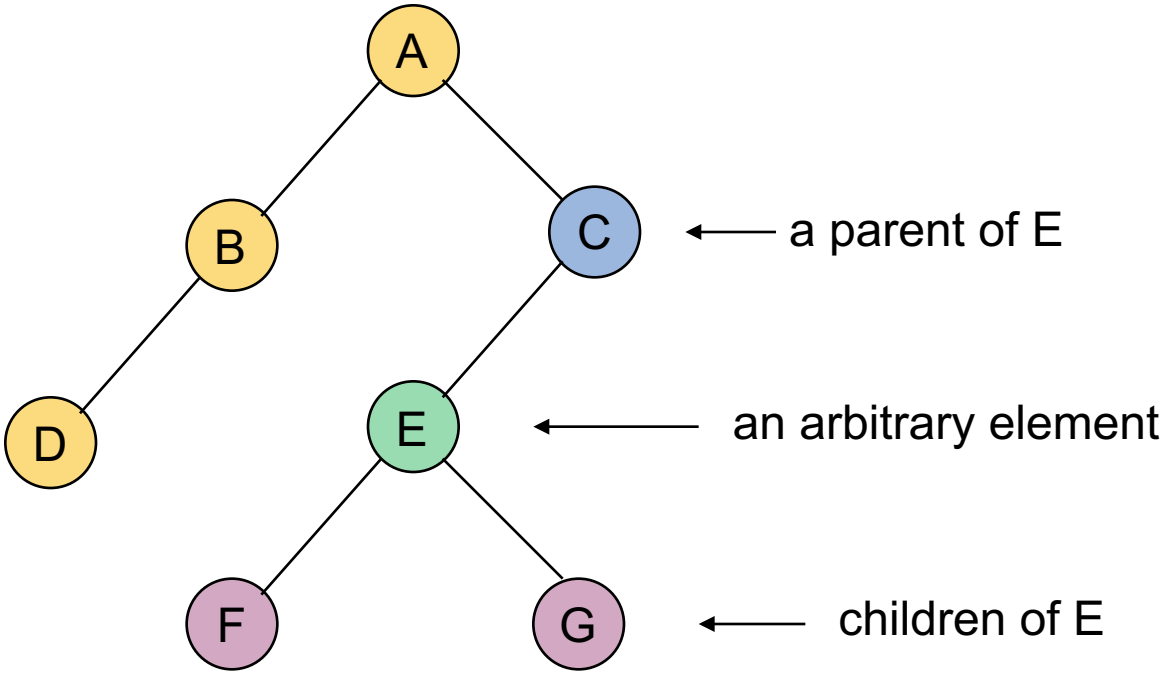
Definitions

- Each element in a tree (except the root)

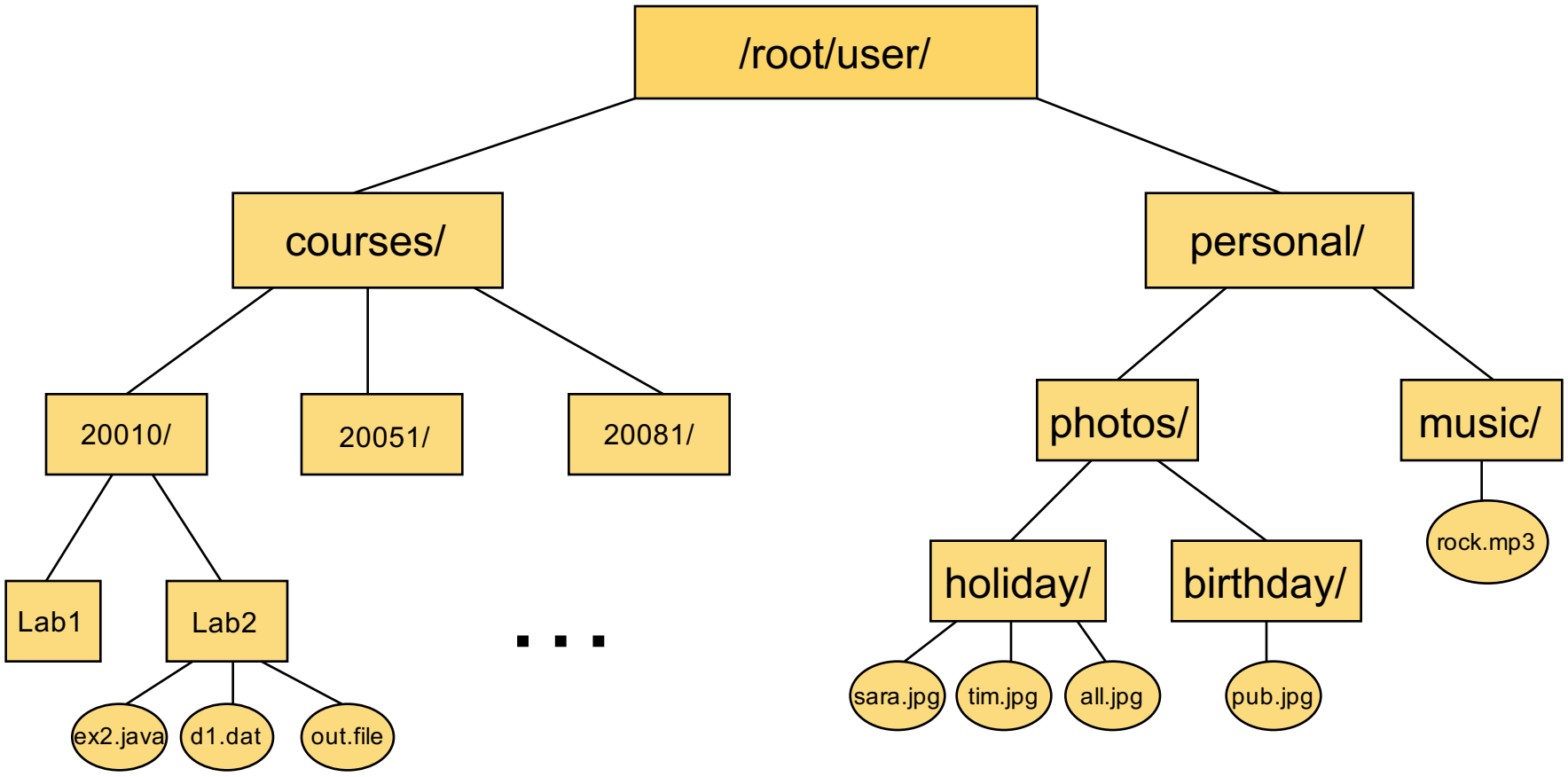


Definitions

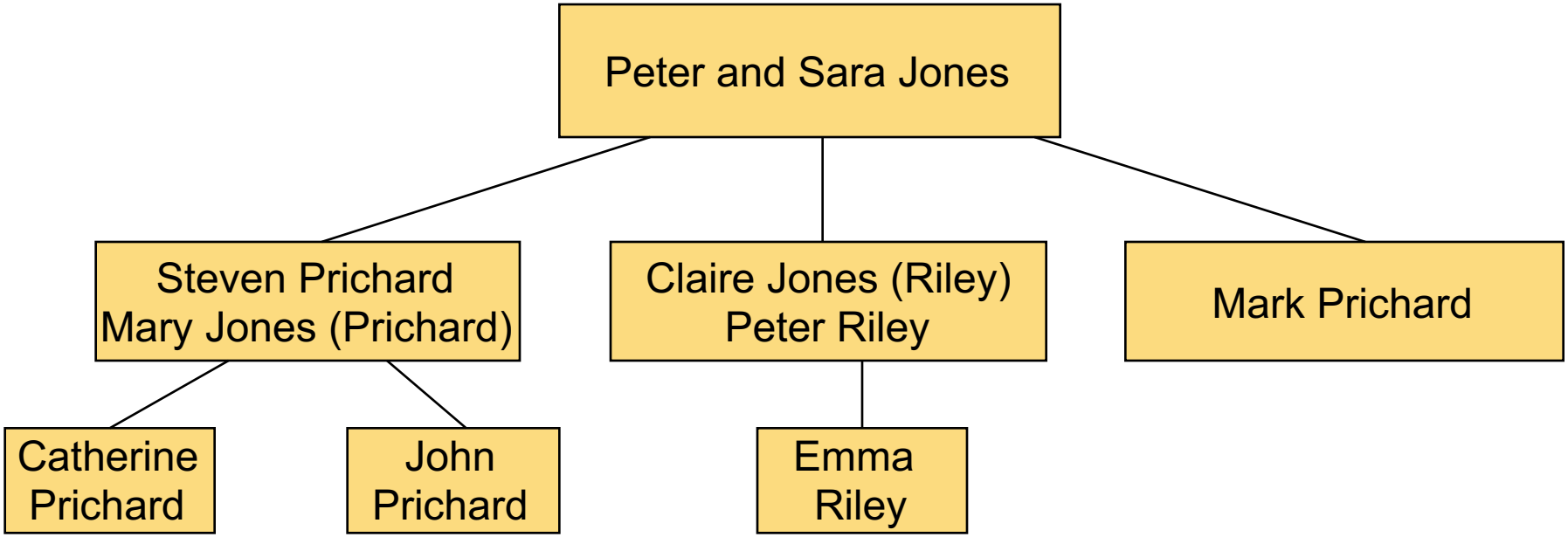
- Each element in a tree (except the root) has a parent and zero or more children elements;



Examples
Computer disc directory structure



Examples
A family tree

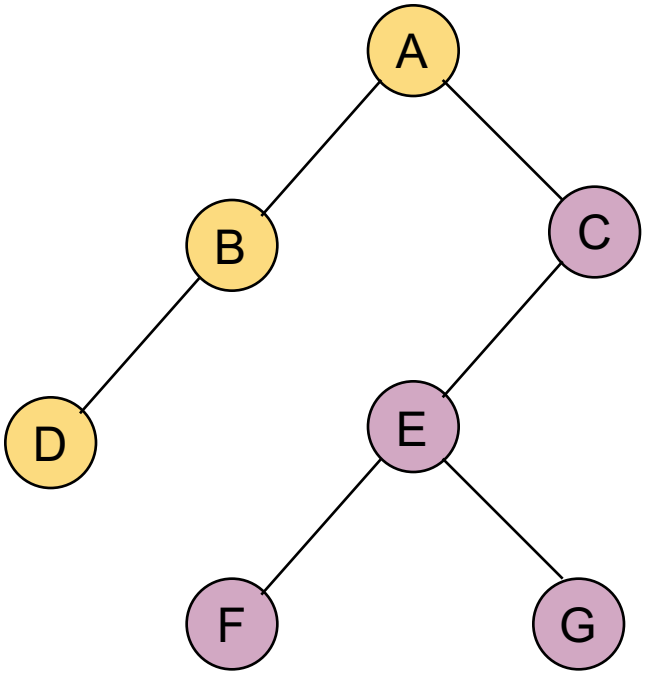


Definitions

- A tree *T* is a non-empty set of nodes storing useful information in a parent-child relationship with the following properties:
  - *T* has a special node *r* referred to as the root;
  - Each node *v* of *T* different from *r* has a parent node *u*;
- If the node *u* is the parent (ancestor) node of *v*, then *v* is a child (descendent) of *u*. Two children of the same parent are siblings.
- A node is external (a leaf node) if it has no children and internal if it has one or more children.

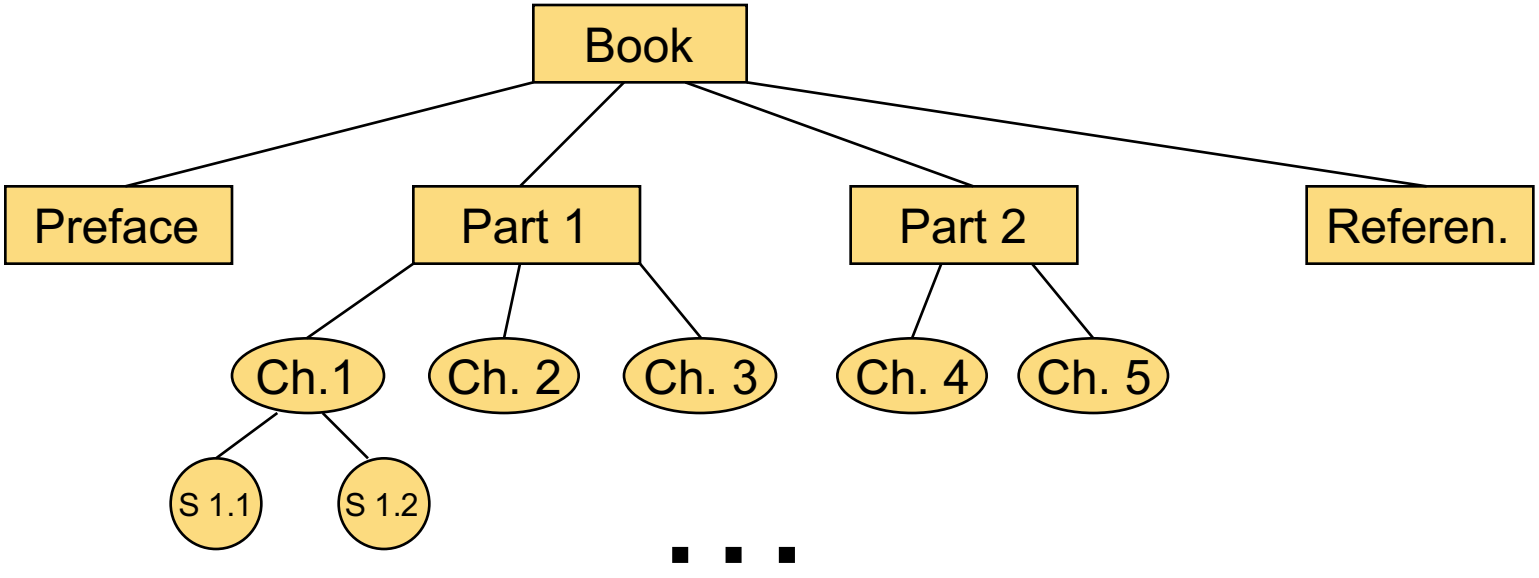
Definitions

- A sub-tree of *T* rooted at the node *v* is a tree consisting of all the descendants of *v* in *T*, including *v* itself.



Ordered trees

- A tree is ordered if a linear ordering relation is defined for the children of each node, that is, we can define an order among them.
- Example: a book



Binary trees

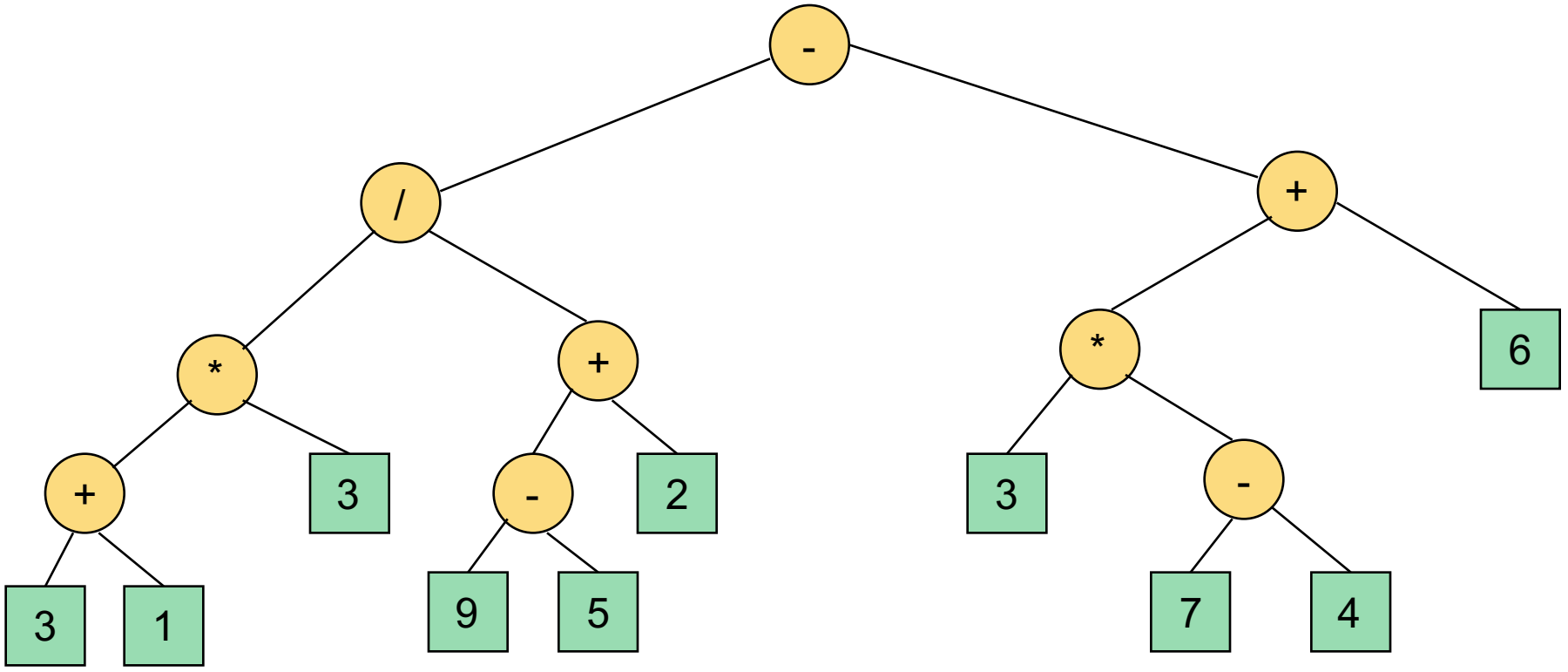
- A binary tree is an ordered tree in which each node has at most two children.
- A binary tree is proper if each internal node has two children.
- For each internal node its children are labelled as a left child and a right child.
- The children are ordered so that a left child comes before a right child.

An example of a binary tree

- Representing an arithmetic expression by a binary tree in which the external nodes are associated with variables or constants and the internal nodes are associated with 4 arithmetic operations.

An example of a binary tree

- ((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))



The tree abstract data type

- Elements of a tree are stored at positions (tree nodes) which are defined relative to neighbouring positions (parent-child relationships).
- Accessor methods for the tree ADT:
  - root() – returns the root of the tree;
  - parent(*v*) – returns the parent node of *v* (an error if *v* is the root);
  - children(*v*) – returns an iterator of the children of the node *v* (if *v* is a leaf node it returns an empty iterator);

The tree abstract data type

- Querry methods for the tree ADT:
  - isInternal(*v*) – tests whether the node *v* is internal;
  - isExternal(*v*) – tests whether the node *v* is external;
  - isRoot(*v*) – tests whether the node *v* is the root;
- Generic methods (not necessarily related to the tree structure):
  - size() – returns the number of nodes of the tree;
  - elements() – returns the an iterator of all the elements stored at the nodes of the tree;
  - positions() – returns an iterator of all the nodes of the tree;
  - swapElements(*v,w*) – swaps the elements stored at the nodes *v* and *w*;
  - replaceElement(*v,e*) – returns the element *v* and replaces it with the element *e*;

Tree traversal

- Complexity of the methods of the tree ADT:
  - root() and parent(*v*) take O(1) time;
  - isInternal(*v*), isExternal(*v*), isRoot(*v*) take O(1) time;
  - children(*v*) takes O(c_v) time, where c_v is the number of children of *v;*
  - swapElements(*v,w*) and replaceElement(*v,e*) take O(1) time;
  - elements() and positions() take O(*n*) time, where *n* is the number of elements in the tree;

Depth of a node

- Let *v* be a node of a tree *T*. The depth of *v* is a number of ancestors of *v*, excluding *v* itself.
- The depth of the root is 0;
- Recursive definition:
  - If *v* is a root, then the depth of *v* is 0;
  - Otherwise, the depth of *v* is one plus the depth of its parent;

```
Algorithm depth(T,v)
  if T.isroot(v) then
     return 0
  else
     return 1+depth(T,T.parent(v))
```

The height of a tree

- It is equal to the maximum depth of an external node of *T.*
- If the previous depth-finding algorithm is applied, the complexity would be $O(n^2)$.
- A recursive definition of height of a node *v* in a tree *T:*
  - If *v* is an external node, the height of *v* is 0;
  - Otherwise, the height of *v* is one plus the maximum height of a child of *v*;

```
Algorithm height(T,v)
  if T.isExternal(v) then
    return 0
  else
    h=0
    for each w in T.children(v) do
      h=max(h,height(T,w)
    return 1+h
```

A traversal of a tree

- A traversal of a tree is a sustematic way of accessing (visiting) all the nodes of *T.*
- There are two different traversal schemes for trees referred to as preorder and postorder.
- In a preorder traversal of a tree *T* the root is visited first, and then the subtrees rooted at its children are traversed recursively.

Algorithm preorder(*T,v)*

perform the action on the node *v*

for each child *w* of *v*

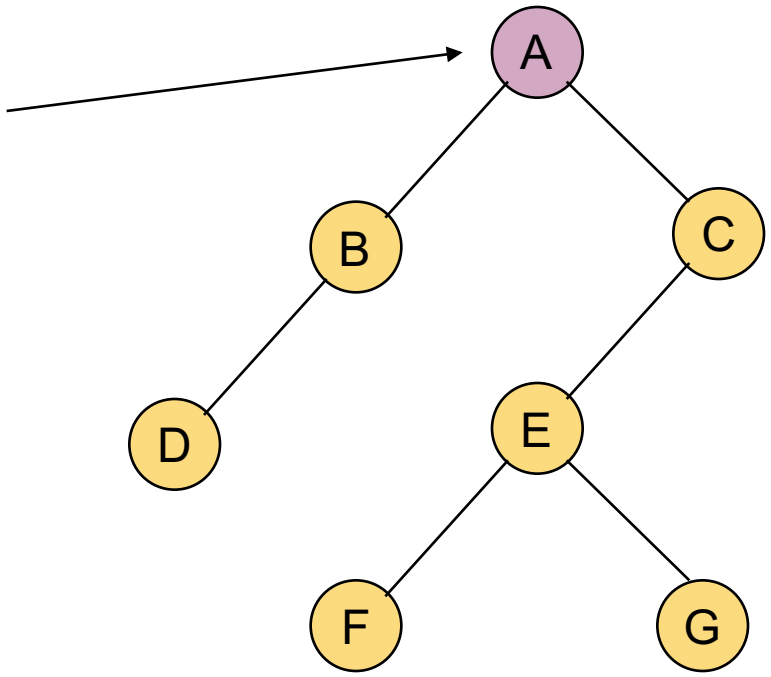
call preorder(*T,w*)

Preorder traversal of a tree

call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)
perform the action on the node *v*

for each child *w* of *v*
call preorder(*T*, *w*)



Preorder traversal of a tree

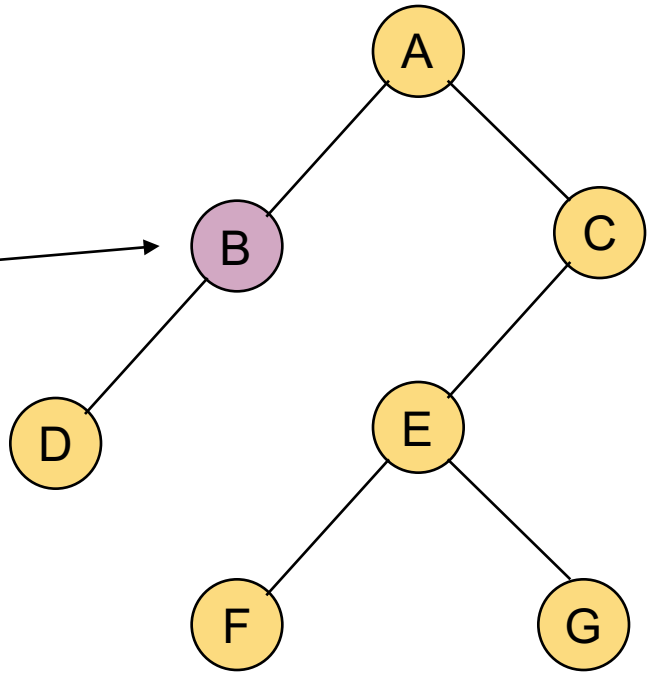
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

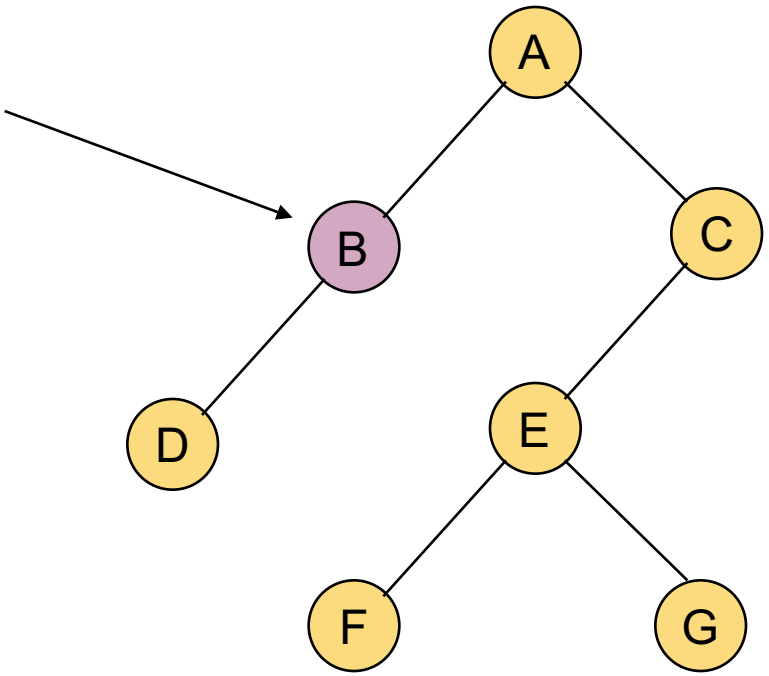
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

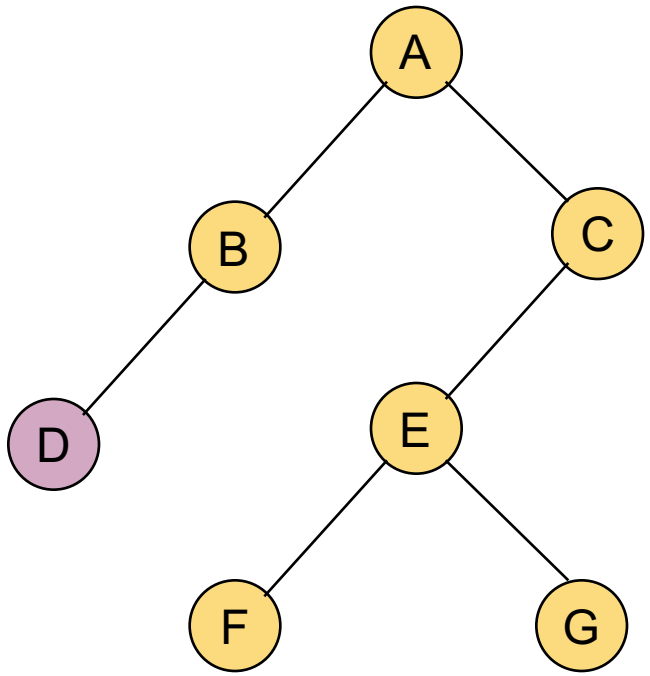
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

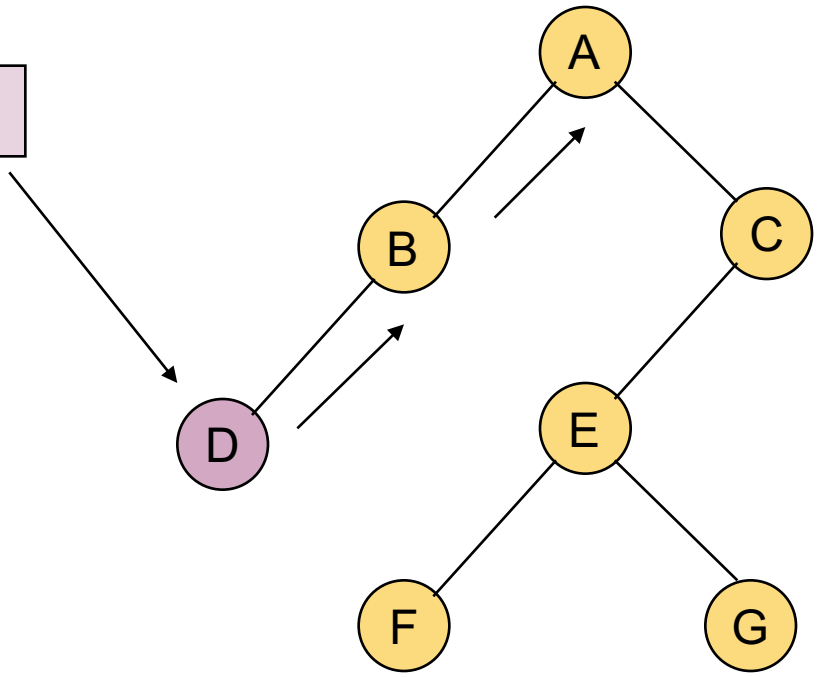
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

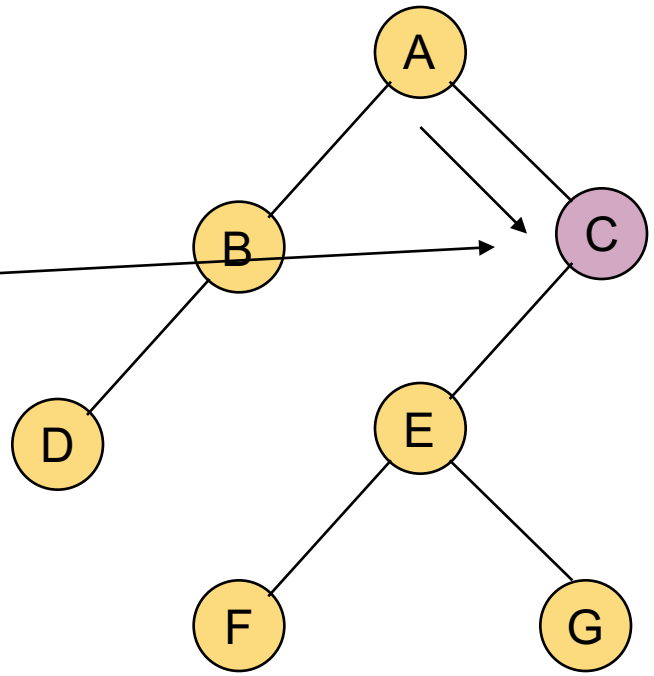
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

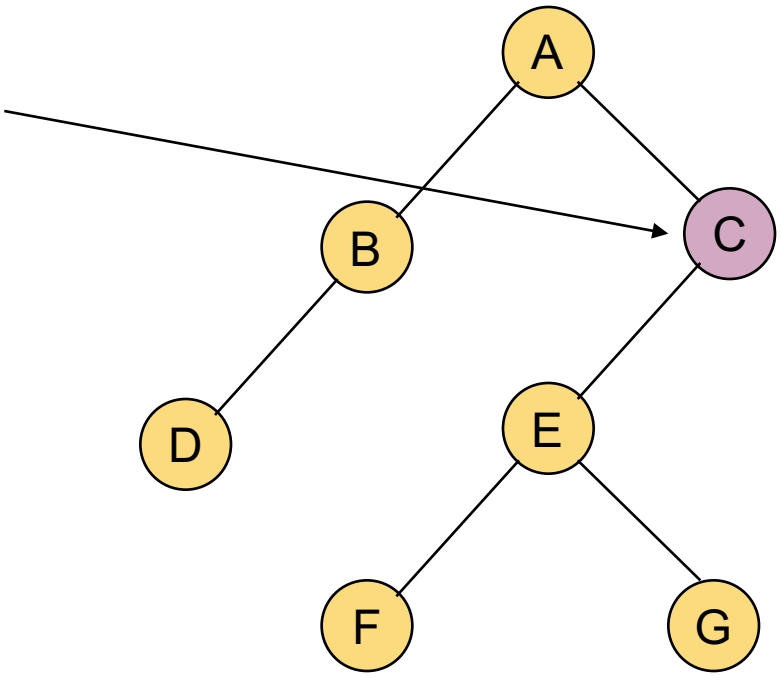
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

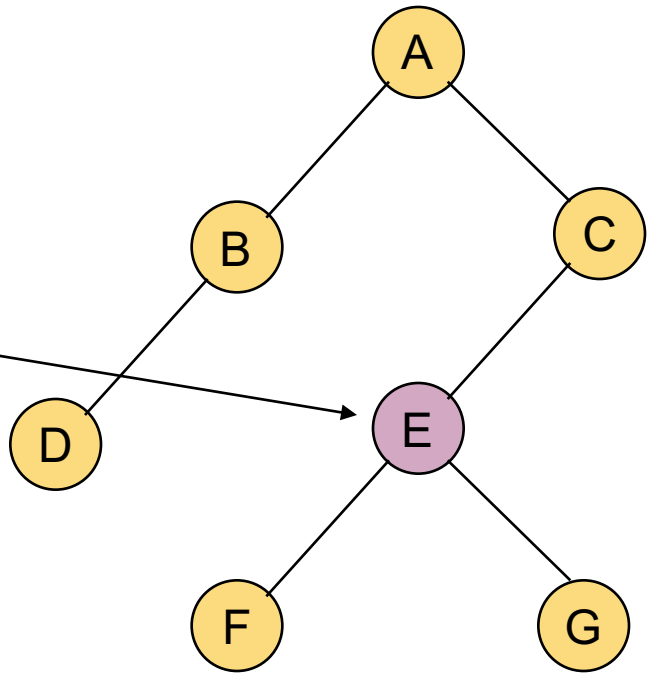
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

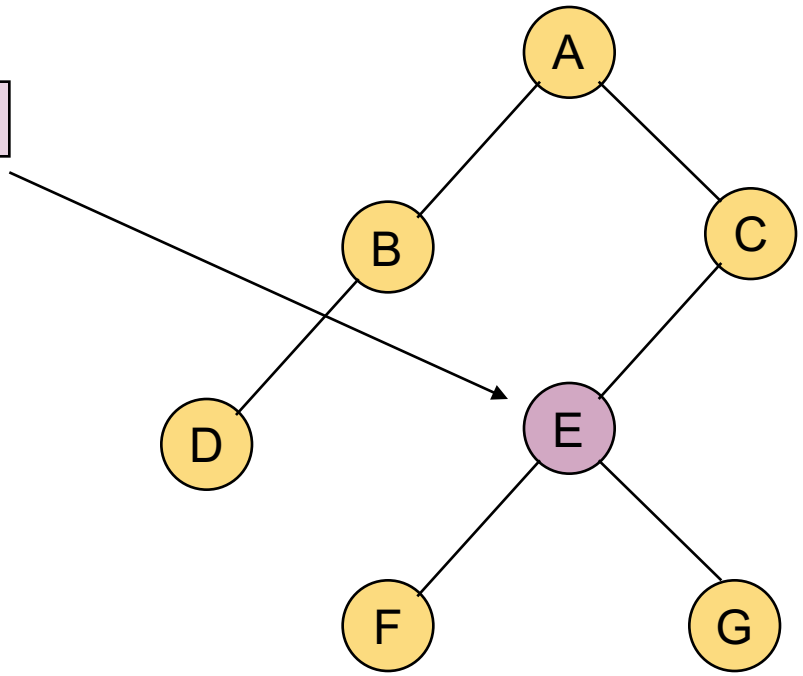
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

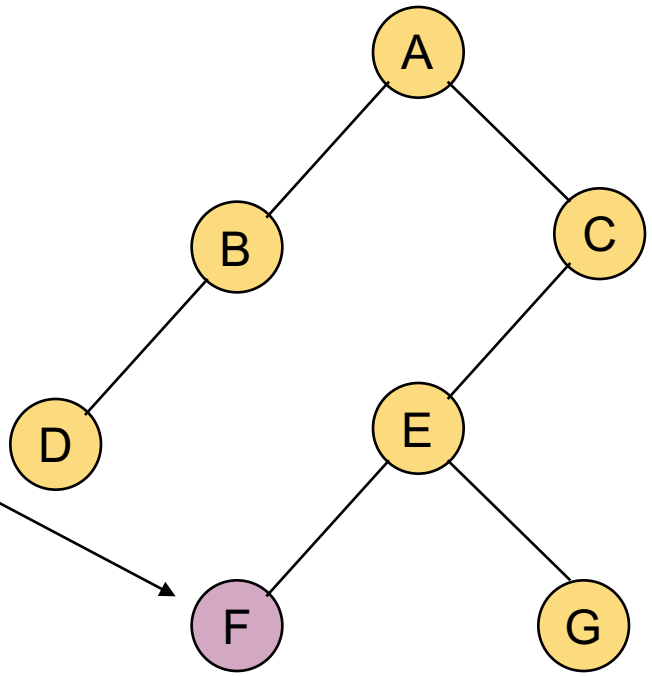
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

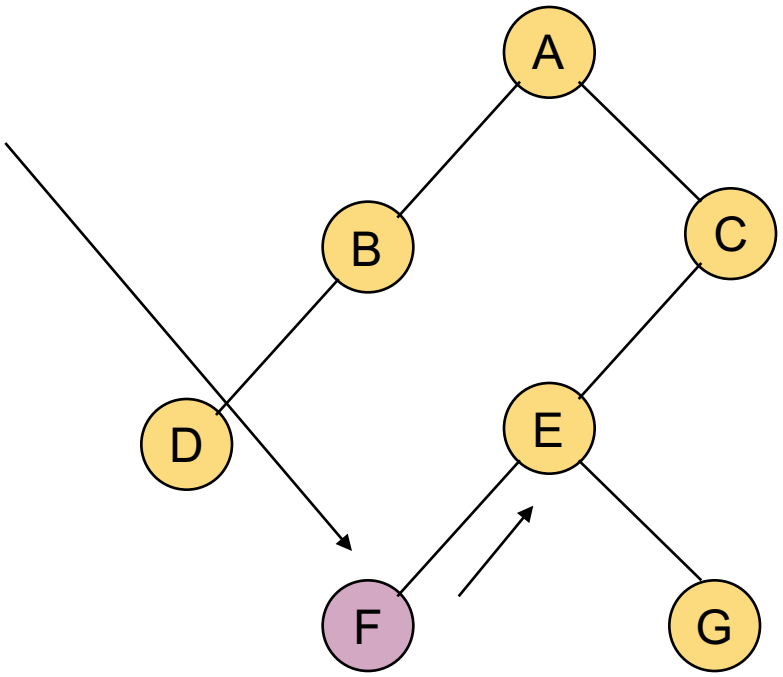
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

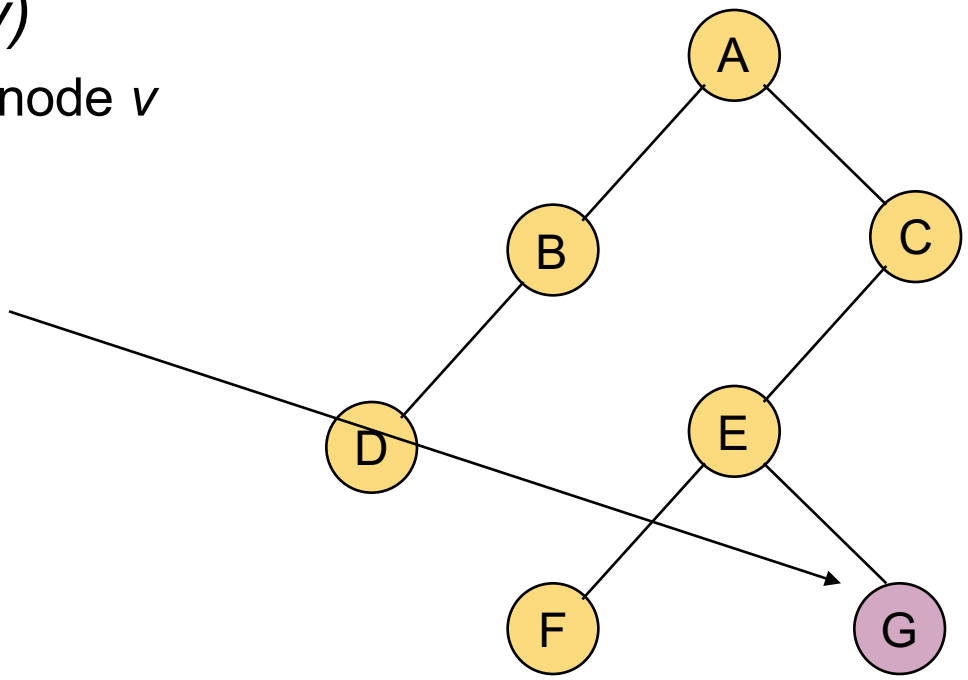
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

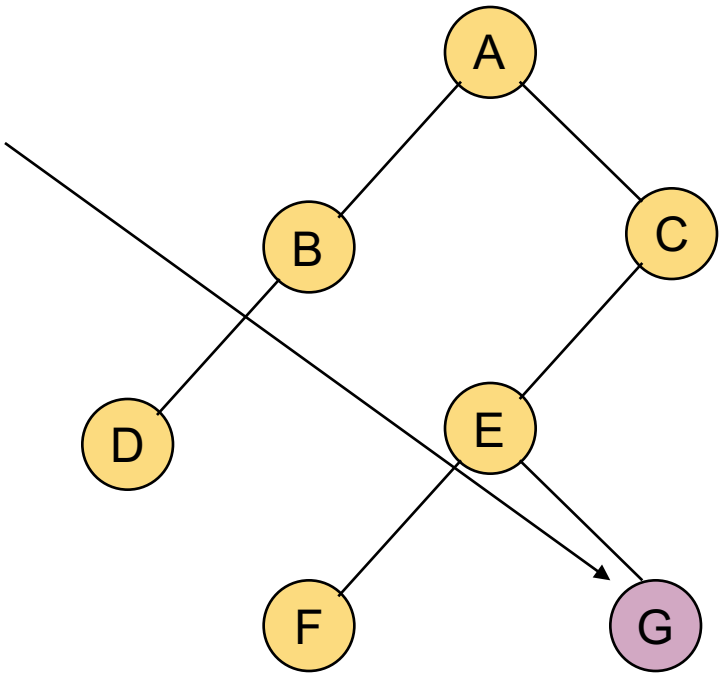
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

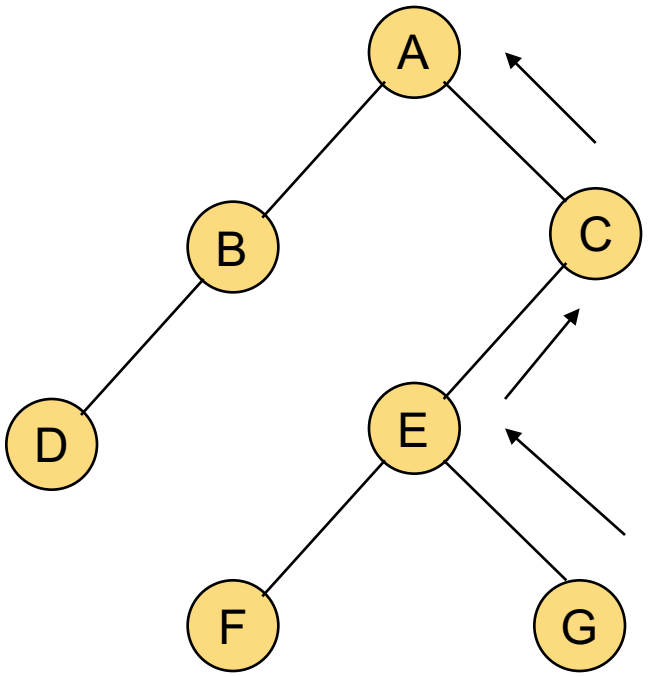
call preorder(*T*, *T*.root)

Algorithm preorder(*T*, *v*)

perform the action on the node *v*

for each child *w* of *v*

call preorder(*T*, *w*)



Preorder traversal of a tree

- It is useful for producing a linear ordering of the nodes in a tree where parents are always before their children.
- If a document is represented as a tree, the preorder traversal examines the document sequentially.
- The overall running time of the preorder traversal is $O(n)$.

Postorder traversal of a tree

- It is a complementary algorithm to preorder traversal, as it traverses recursively the subtrees rooted at the children of the root first before visiting the root.

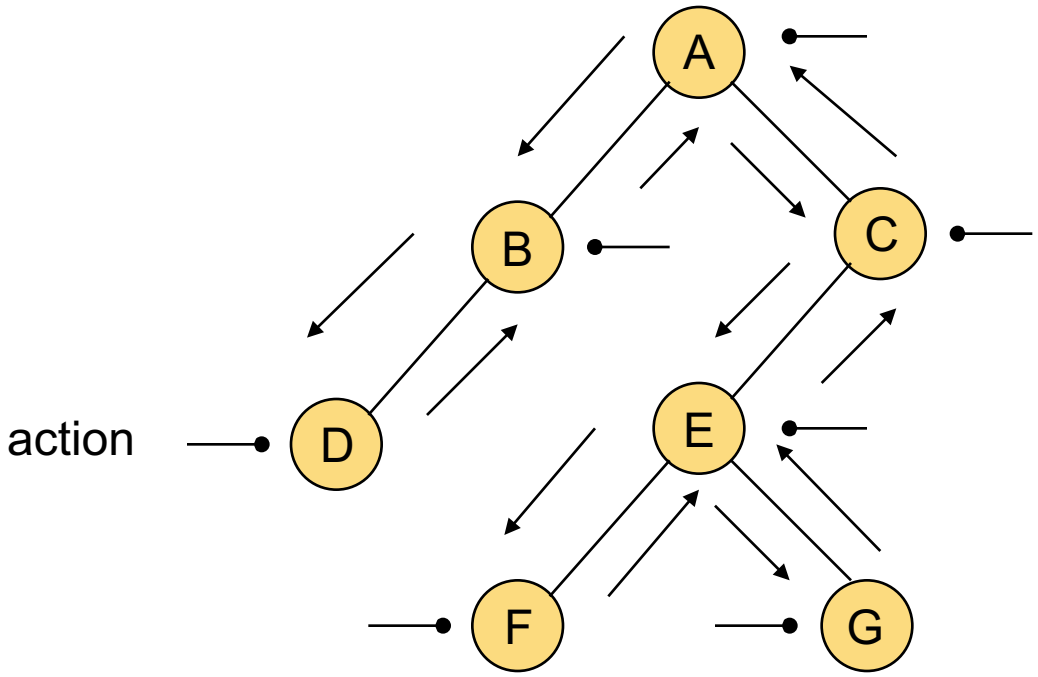
Algorithm postorder(*T*,*v*)

for each child *w* of *v* **do**

call postorder(*T*,*w*)

perform the action on the node *v*

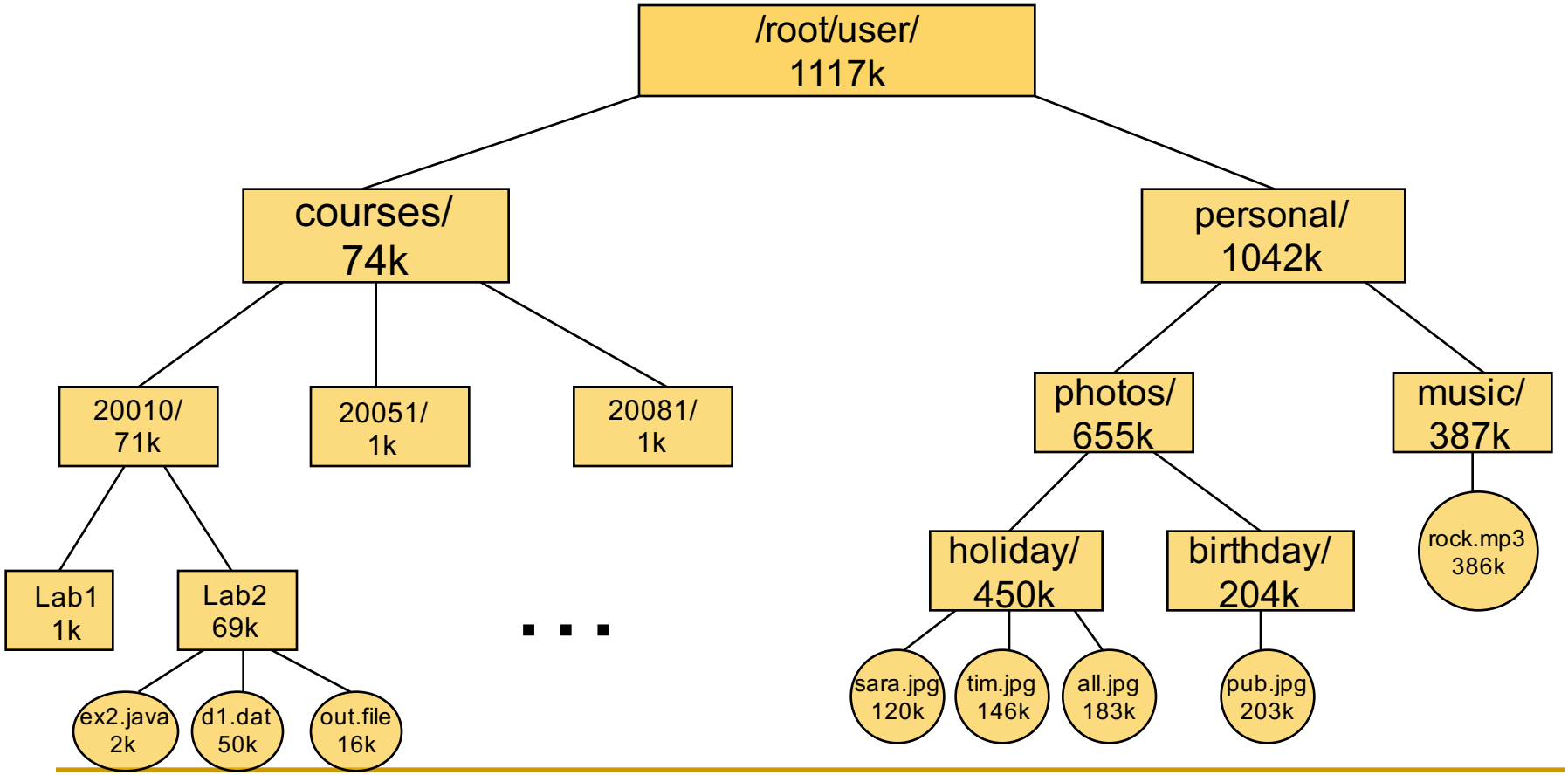
Postorder traversal of a tree



Postorder traversal of a tree

- The postorder traversal of a tree with n nodes takes O(n) time, assuming that visiting each node takes O(1) time.
- The algorithm is useful if computing a certain property of a node in a tree requires that this property is previously computed for all its children.

Postorder traversal of a tree



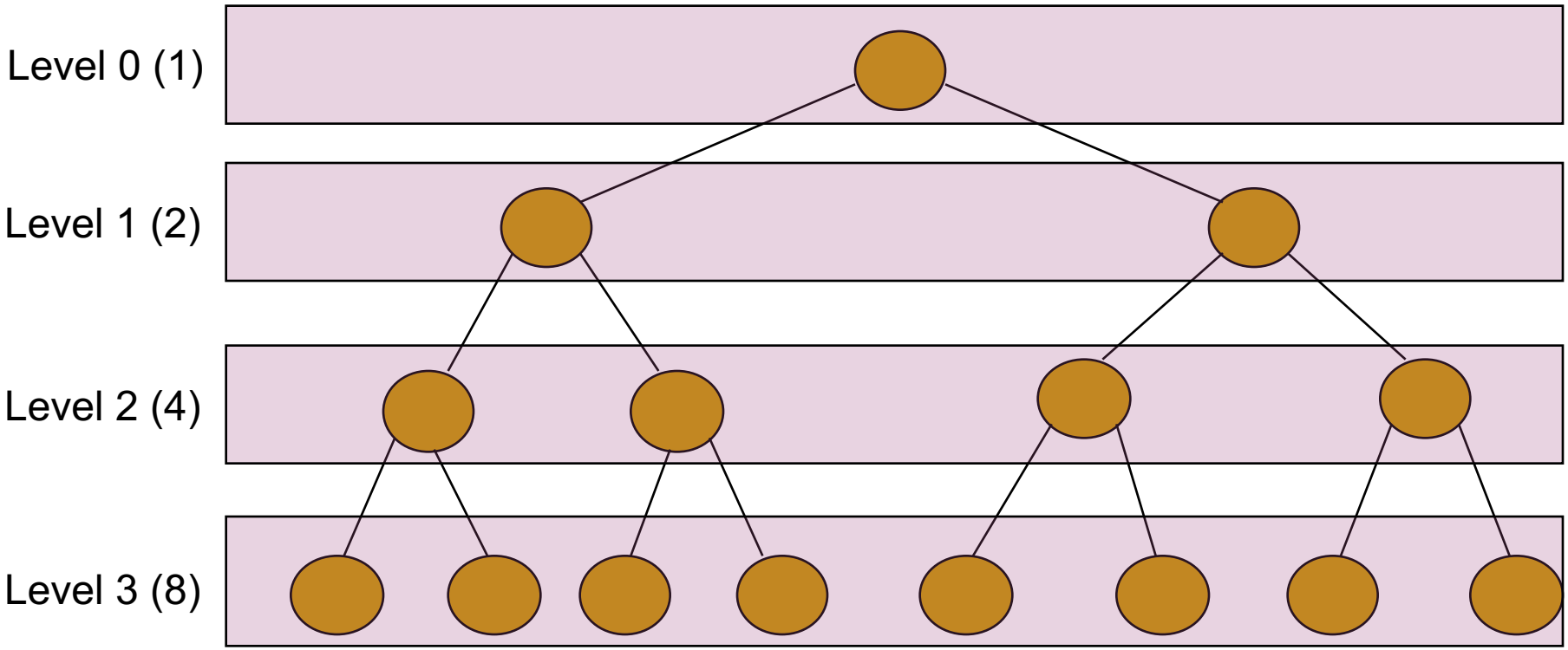
Binary trees

- A proper binary tree is an ordered tree in which each internal node has exactly two children.
- As an ADT, a binary tree supports 3 additional accessor methods:
  - leftChild(*v*) – returns the left child of *v*; if *v* is an external node, an error occurs;
  - rightChild(*v*) – returns the right child of *v*; if *v* is an external node, an error occurs;
  - sibling(*v*) – returns the sibling of *v*; an error occurs if *v* is the root;

Binary trees

- Denote all the nodes of a binary tree *T* at the same depth *d* as the level *d* of *T*;
- Level 0 has 1 node (the root), level 1 has at most 2 nodes, etc. In general, level *d* has at most 2^d nodes;
- In a proper binary tree the number of external nodes is 1 more than the number of internal nodes;

Binary trees



Preorder traversal of a binary tree

Algorithm binaryPreorder(*T*,*v*)
  perform the action on the node *v*
  if *v* is an internal node **then**
    call binaryPreorder(*T*,*T*.leftChild(*v*))
    call binaryPreorder(*T*,*T*.rightChild(*v*))

Postorder traversal of a binary tree

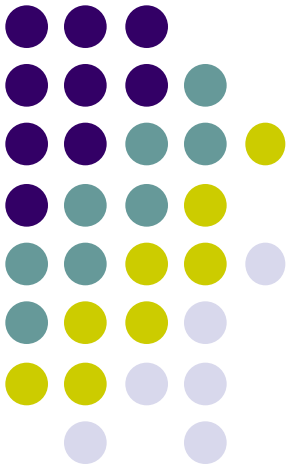
Algorithm binaryPostorder(*T*,*v*)
  if *v* is an internal node **then**
    call binaryPostorder(*T*,*T*.leftChild(*v*))
    call binaryPostorder(*T*,*T*.rightChild(*v*))
  perform the action on the node *v*

Inorder traversal of a binary tree

- In this method the action on a node *v* is performed in between the recursive traversals on its left and right subtrees.
- The inorder traversal of a binary tree can be viewed as visiting the nodes from left to right.

Algorithm binaryInorder(*T,v*)
  if *v* is an internal node **then**
    call binaryInorder(*T,T.*leftChild(*v*))
  perform the action on the node *v*
  if *v* is an internal node **then**
    call binaryInorder(*T,T.*leftChild(*v*))

COMP20010: Algorithms and Imperative Programming

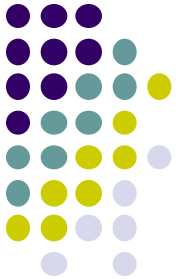


Lecture 2

Data structures for binary trees

Priority queues

Lecture outline



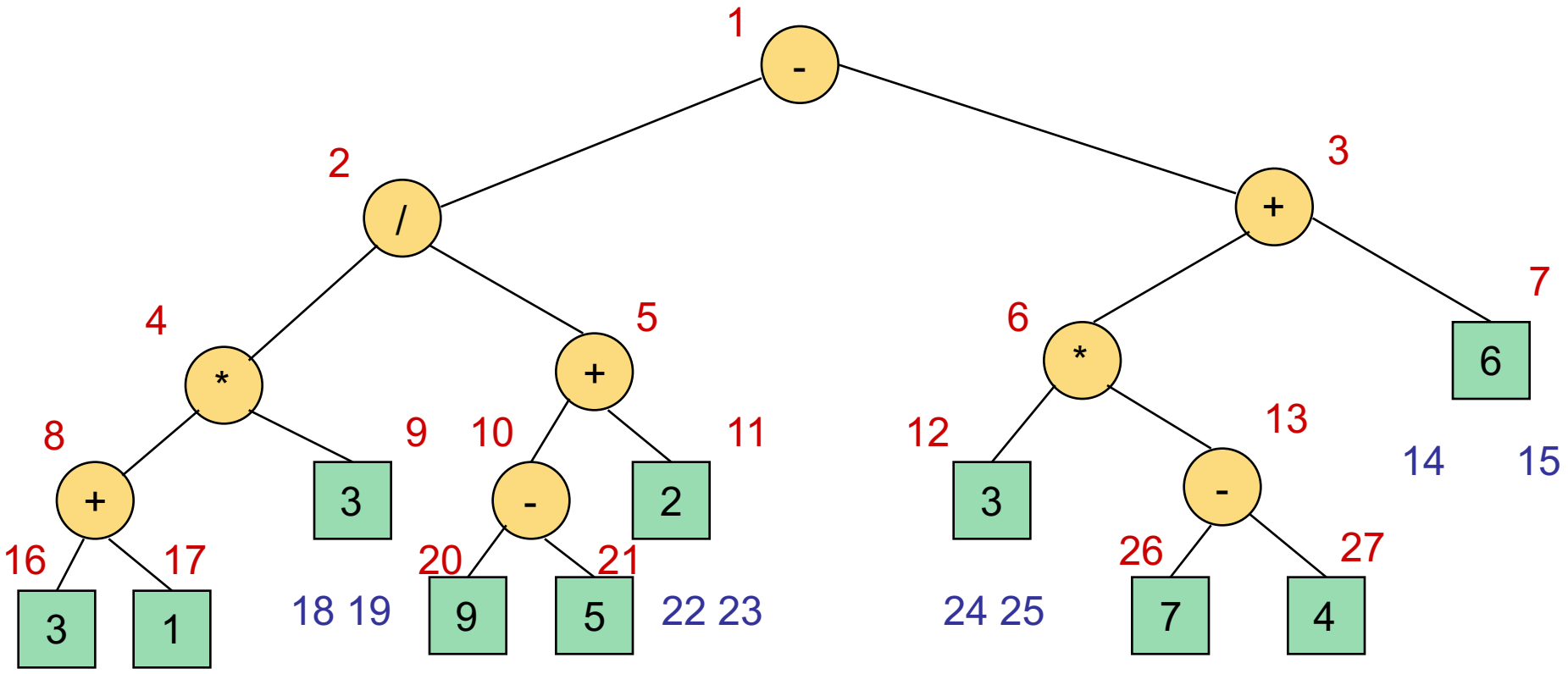
- Different data structures for representing binary trees (vector-based, linked), linked structure for general trees;
- Priority queues (PQs), sorting using PQs;

Data structures for representing trees
A vector-based data structure

- A vector-based structure for binary trees is based on a simple way of numbering the nodes of *T*.
- For every node *v* of *T* define an integer *p(v)*:
  - If *v* is the root, then *p(v)*=1;
  - If *v* is the left child of the node *u*, then *p(v)*=2*p(u)*;
  - If *v* is the right child of the node *u*, then *p(v)*=2*p(u)*+*1*;
- The numbering function *p(.)* is known as a level numbering of the nodes in a binary tree *T*.

Data structures for representing trees
A vector-based data structure

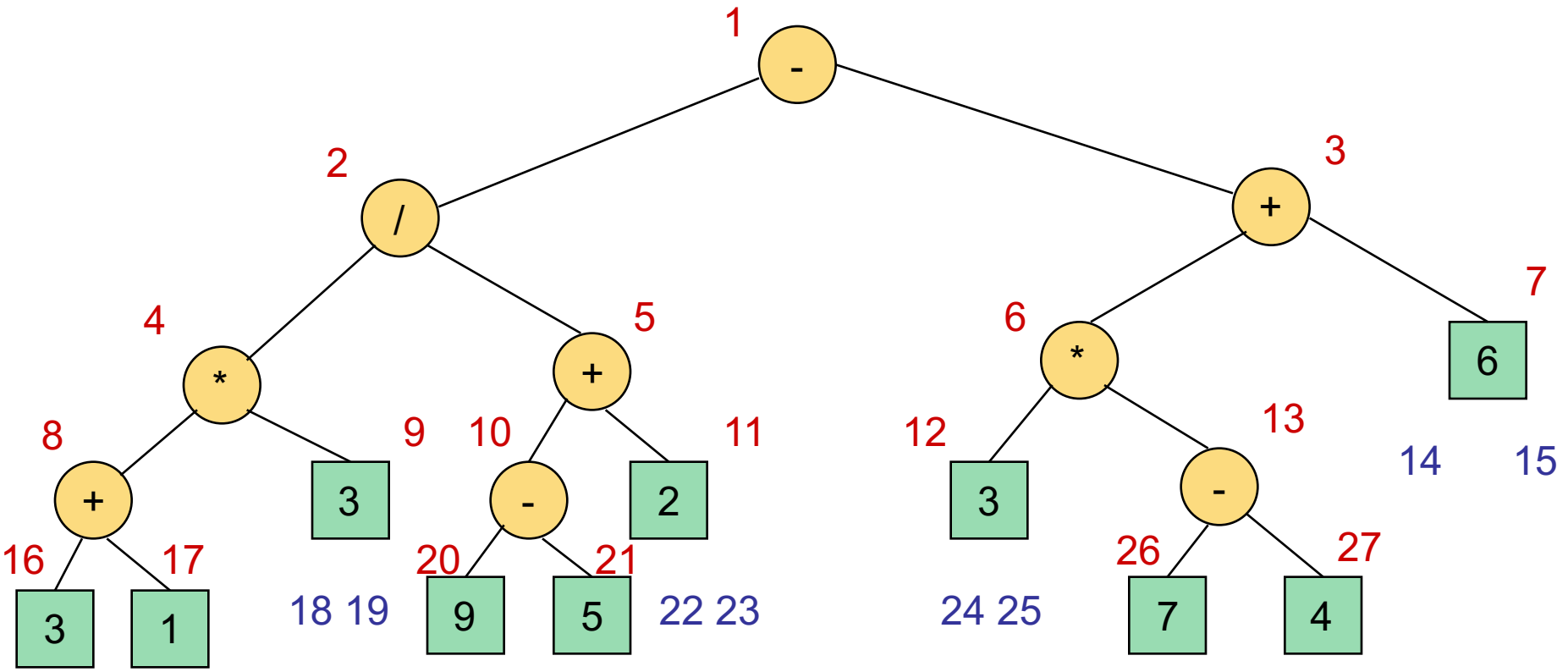
- ((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))



Binary tree level numbering

Data structures for representing trees
A vector-based data structure

- The level numbering suggests a representation of a binary tree *T* by a vector *S*, such that the node *v* from *T* is associated with an element *S*[*p(v)*];



S

-	/	+	*	+	*	6	+	3	-	2	3	-			3	1			9	5					7	4

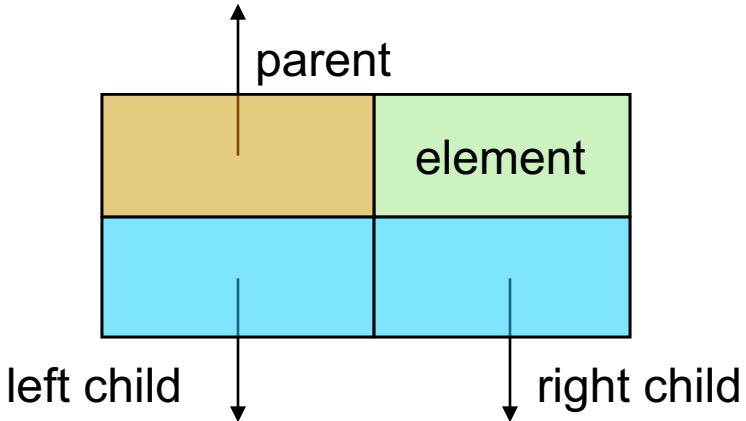
Data structures for representing trees
A vector-based data structure

Operation	Time
positions(), elements()	$O(n)$
swapElements(), replaceElement()	$O(1)$
root(), parent(), children()	$O(1)$
leftChild(), rightChild(), sibling()	$O(1)$
isInternal(), isExternal(), isRoot()	$O(1)$

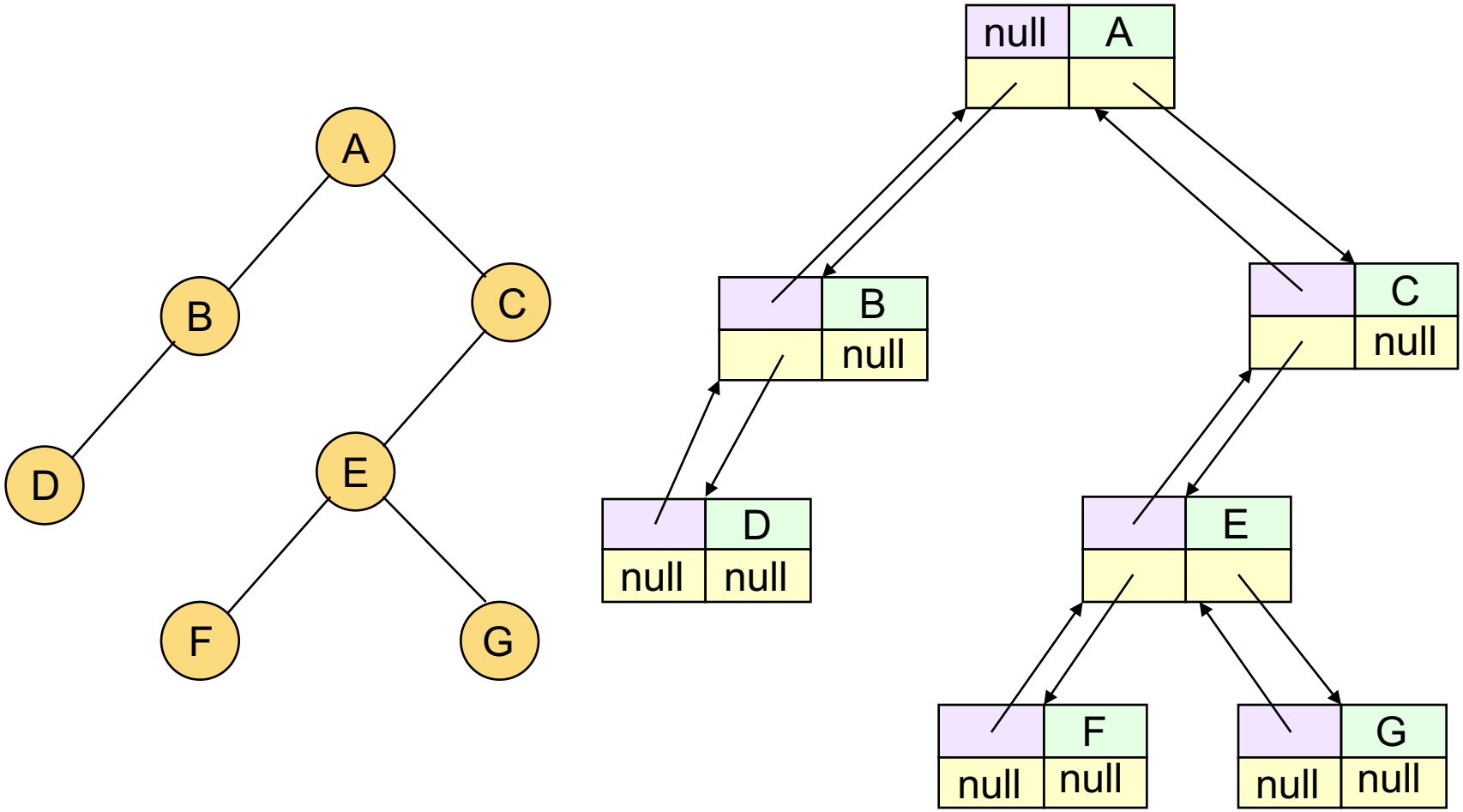
Running times of the methods when a binary tree T is implemented as a vector

Data structures for representing trees
A linked data structure

- The vector implementation of a binary tree is fast and simple, but it may be space inefficient when the tree height is large (why?);
- A natural way of representing a binary tree is to use a linked structure.
- Each node of *T* is represented by an object that references to the element *v* and the positions associated with with its parent and children.

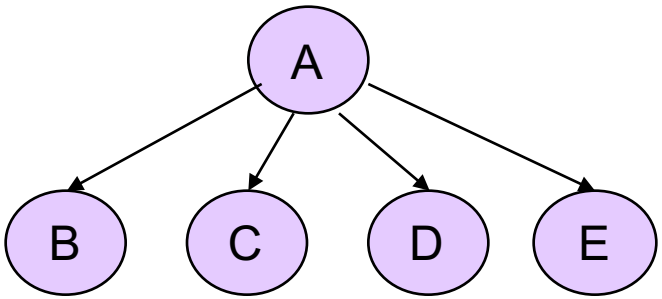


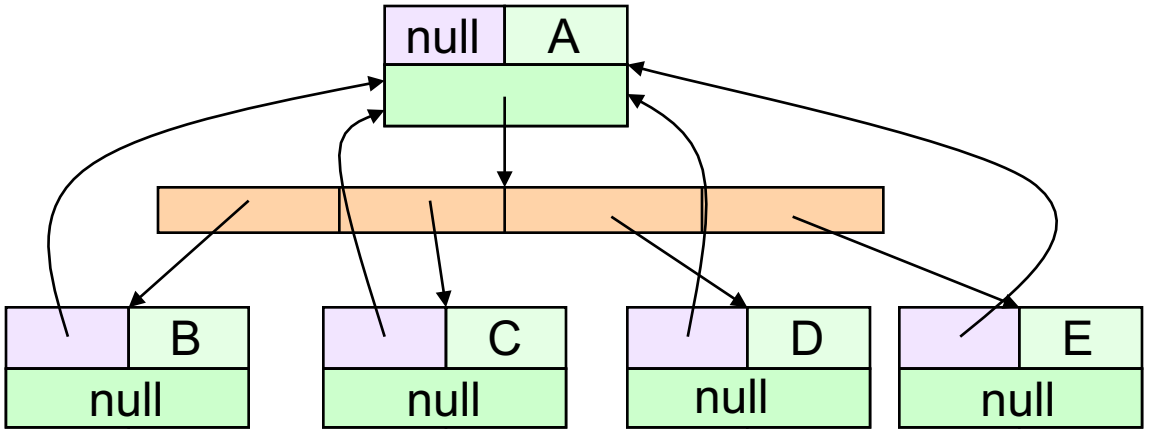
Data structures for representing trees
A linked data structure for binary trees



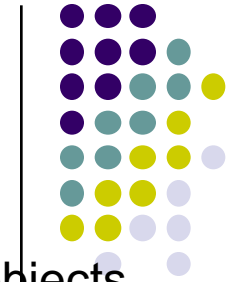
Data structures for representing trees
A linked data structure for general trees

- In order to extend the previous data structure to the case of general trees;
- In order to register a potentially large number of children of a node, we need to use a container (a list or a vector) to store the children, instead of using instance variables;



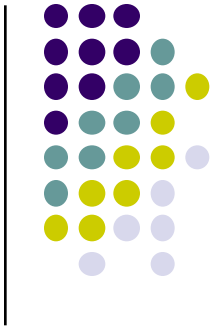


Keys and the total order relation



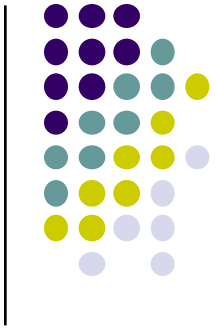
- In various applications it is frequently required to compare and rank objects according to some parameters or properties, called keys that are assigned to each object in a collection.
- A key is an object assigned to an element as a specific attribute that can be used to identify, rank or weight that element.
- A rule for comparing keys needs to be robustly defined (not contradicting).
- We need to define a total order relation, denoted by $\leq$ with the following properties:
  - Reflexive property: $k \leq k$;
  - Antisymmetric property: if $k_1 \leq k_2$ and $k_2 \leq k_1$, then $k_1 = k_2$;
  - Transitive property: if $k_1 \leq k_2$ and $k_2 \leq k_3$, then $k_1 \leq k_3$;
- The comparison rule that satisfies the above properties defines a linear ordering relationship among a set of keys.
- In a finite collection of elements with a defined total order relation we can define the smallest key $k_{\min}$ as the key for which $k_{\min} \leq k$ for any other key *k* in the collection.

Priority queues



- A priority queue *P* is a container of elements with keys associated to them at the time of insertion.
- Two fundamental methods of a priority queue *P* are:
  - insertItem(*k*,*e*) – inserts an element *e* with a key *k* into *P*;
  - removeMin() – returns and removes from *P* an element with a smallest key;
- The priority queue ADT is simpler than that of the sequence ADT. This simplicity originates from the fact that the elements in a PQ are inserted and removed based on their keys, while the elements are inserted and removed from a sequence based on their positions and ranks.

Priority queues

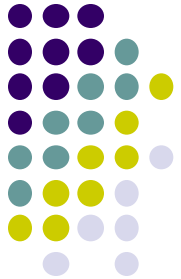


- A comparator is an object that compares two keys. It is associated with a priority queue at the time of construction.
- A comparator method provides the following objects, each taking two keys and comparing them:
  - isLess(k_1, k_2) – true if $k_1 < k_2$;
  - isLessOrEqualTo(k_1, k_2) – true if $k_1 \le k_2$;
  - isEqualTo(k_1, k_2) – true if $k_1 = k_2$;
  - isGreater(k_1, k_2) – true if $k_1 > k_2$;
  - isGreaterOrEqualTo(k_1, k_2) – true if $k_1 \ge k_2$;
  - isComparable(*k*) – true if *k* can be compared;

PQ-Sort

- Sorting problem is to transform a collection C of n elements that can be compared and ordered according to a total order relation.
- Algorithm outline:
  - Given a collection C of n elements;
  - In the first phase we put the elements of C into an initially empty priority queue P by applying n insertItem(c) operations;
  - In the second phase we extract the elements from P in non-decreasing order by applying n removeMin operations, and putting them back into C;

PQ-Sort

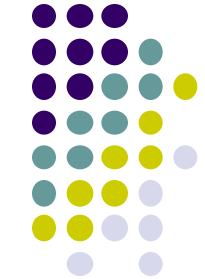


- **Algorithm** PQ-Sort(*C,P*);
  - **Input:** A sequence *C*[1:n] and a priority queue *P* that compares keys (elements of *C*) using a total order relation;
  - **Output:** A sequence *C*[1:n] sorted by the total order relation;

```
while C is not empty do
    e ← C.removeFirst();      {remove an element e from C}
    P.insertItem(e,e);        {the key is the element itself}
while P is not empty do
    e ← P.removeMin()         {remove the smallest element from P}
    C.insertLast(e)           {add the element at the end of C}
```

- This algorithm does not specify how the priority queue *P* is implemented. Depending on that, several popular schemes can be obtained, such as selection-sort, insertion-sort and heap-sort.

Priority queue implemented with an unordered sequence and Selection-Sort



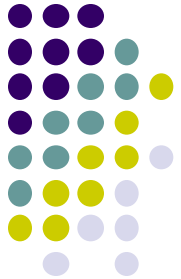
- Assume that the elements of *P* and their keys are stored in a sequence *S*, which is implemented as either an array or a doubly-linked list.
- The elements of *S* are pairs (*k,e*), where *e* is an element of *P* and *k* is the key.
- New element is added to *S* by appending it at the end (executing insertLast*(k,e)*), which means that *S* will be unsorted.
- insertLast() will take *O*(1) time, but finding the element in *S* with a minimal key will take *O*(*n*).

Priority queue implemented with an unordered sequence and Selection-Sort

- The first phase of the algorithm takes $O(n)$ time, assuming that each insertion takes $O(1)$ time.
- Assuming that two keys can be compared in $O(1)$ time, the execution time of each removeMin operation is proportional to the number of elements currently in *P.*
- The main bottleneck of this algorithm is the repeated selection of a minimal element from an unsorted sequence in Phase 2. This is why the algorithm is referred to as selection-sort.
- The bottleneck of the selection-sort algorithm is the second phase. Total time needed for the second phase is

$$O(n) + O(n-1) + \cdots + O(1) = \sum_{i=1}^{n} O(i) = O(n^2)$$

Priority queue implemented with a sorted sequence and Insertion-Sort



- An alternative approach is to sort the elements in the sequence *S* by their key values.
- In this case the method removeMin actually removes the first element from *S*, which takes *O*(1) time.
- However, the method insertItem requires to scan through the sequence *S* for an appropriate position to insert the new element and its key. This takes *O*(*n*) time.
- The main bottleneck of this algorithm is the repeated insertion of elements into a sorted priority queue. This is why the algorithm is referred to as insertion-sort.
- The total execution time of insertion-sort is dominated by the first phase and is $O(n^2)$.

COMP20010: Algorithms and Imperative Programming

Lecture 3

Heaps

Dictionaries and Hash Tables

Lecture outline

- The heap data structure;
- Implementing priority queues as heaps;
- The vector representation of a heap and basic operations (insertion,removal);
- Heap-Sort;
- Dictionaries (the unordered dictionary ADT);
- Hash tables (bucket arrays, hash functions);
- Collision handling schemes;

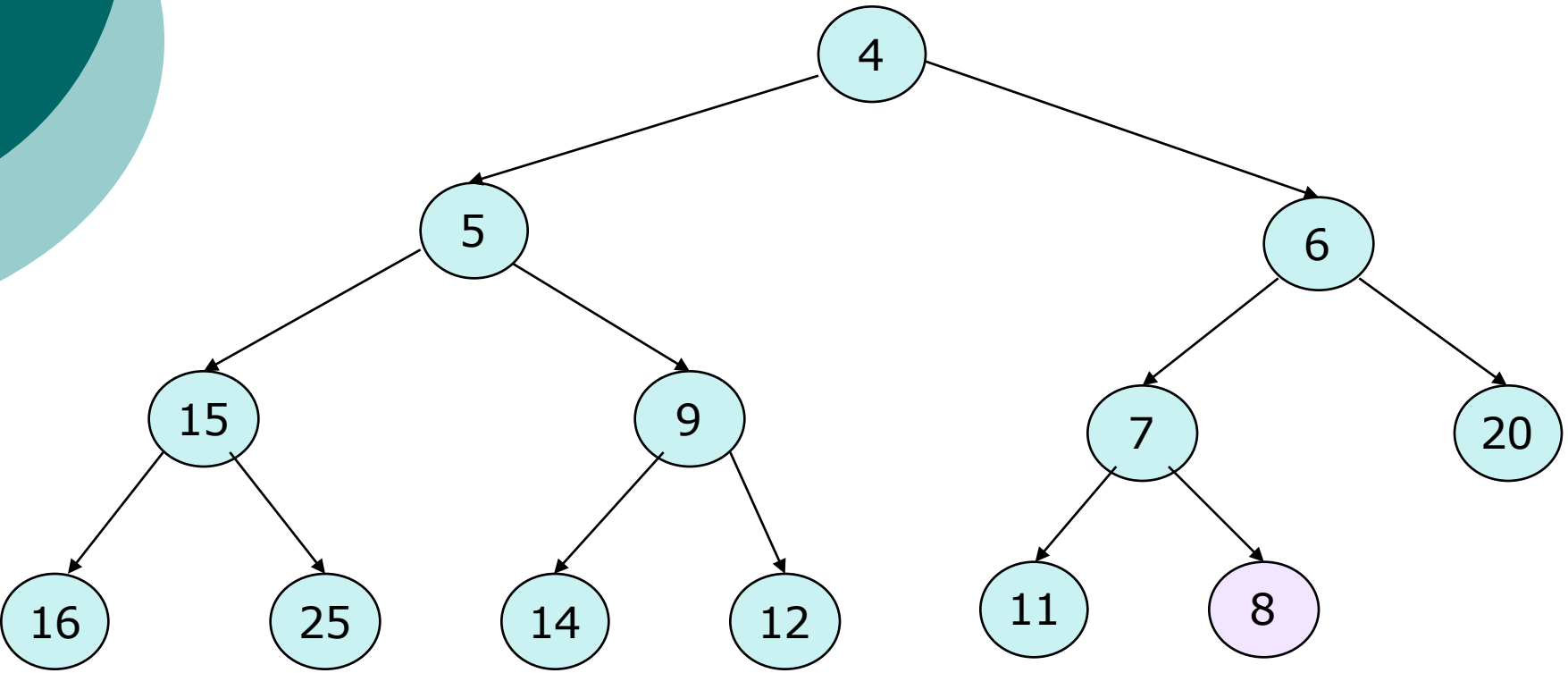
The heap data structure

- The aim is to provide a realisation of a priority queue that is efficient for both insertions and removals.
- This can be accomplished with a data structure called a heap, which enables to perform both insertions and removals in logarithmic time.
- The idea is to store the elements in a binary tree instead of a sequence.

The heap data structure

- A heap is a binary tree that stores a collection of keys at its internal nodes that satisfies two additional properties:
  - A relational property (that affects how the keys are stored);
  - A structural property;
- We assume a total order relationship on the keys.
- Heap-Order property: In a heap *T* for every node *v* other than a root, the key stored in *v* is greater or equal than the key stored at its parent.
- The consequence is that the keys encountered on a path from the root to an external node are in non-decreasing order and that a minimum key is always stored at the root.
- Complete binary tree property: A binary tree *T* with height *h* is complete if the levels 0 to *h*-1 have the maximum number of nodes (level *i* has 2^i nodes for *i*=0,…,h-1) and in the level *h*-1 all internal nodes are to the left of the external nodes.

The heap data structure

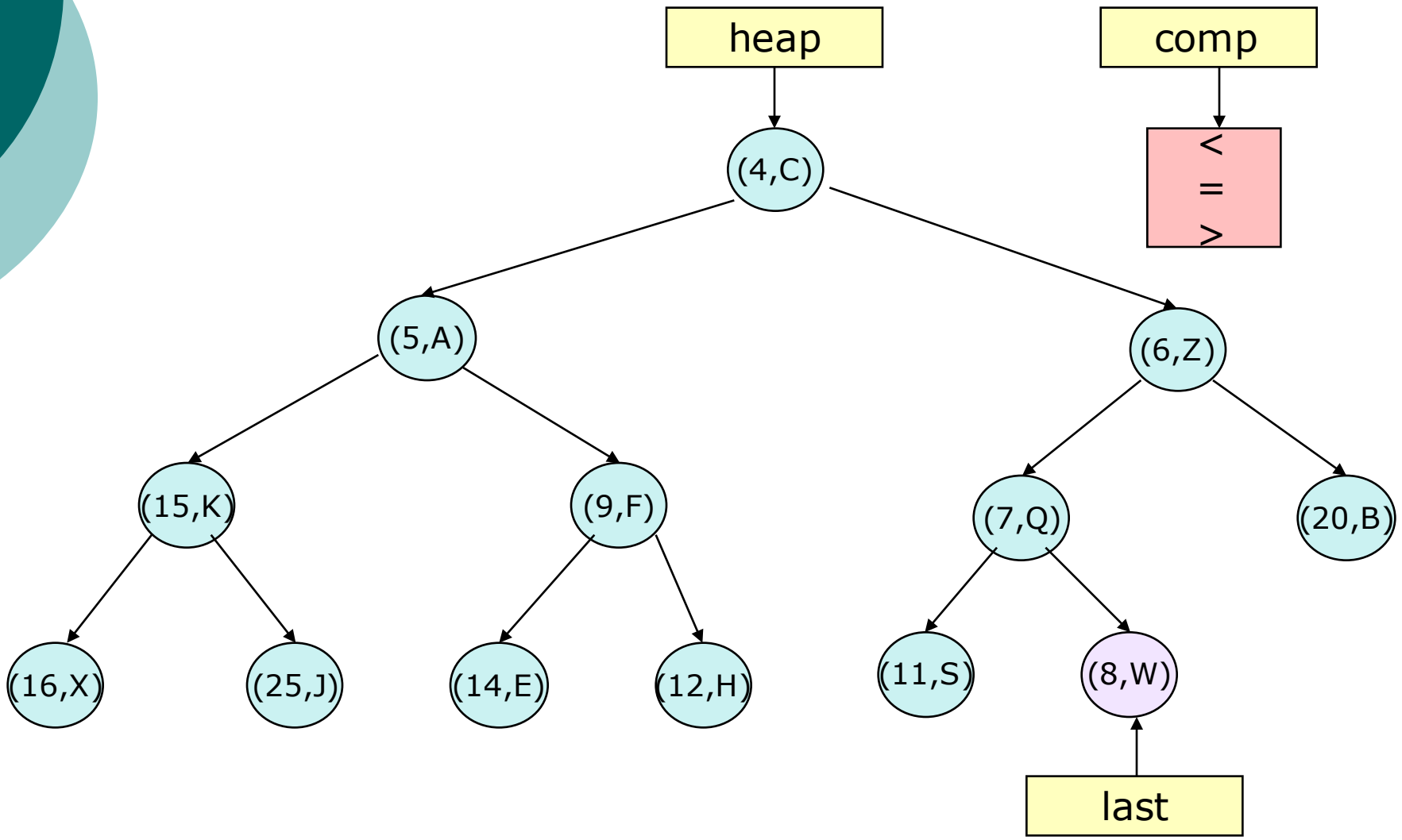


An example of a heap T storing 13 integer keys. The last node (the right-most, deepest internal node of T) is 8 in this case

Implementing a Priority Queue with a Heap

- A heap-based priority queue consists of:
  - **heap:** a complete binary tree with keys that satisfy the heap-order property. The binary tree is implemented as a vector.
  - **last:** A reference to the last node in *T*. For a vector implementation, last is an integer index to the vector element storing the last node of *T*.
  - **comp:** A comparator that defines the total order relation among the keys. The comparator should maintain the minimal element at the root.
- A heap *T* with *n* keys has height $h = \lceil \log(n+1) \rceil$.
- If the update operations on a heap can be performed in time proportional to its height, rather than to the number of its elements, then these operations will have complexity $O(\log n)$.

Implementing a Priority Queue with a Heap



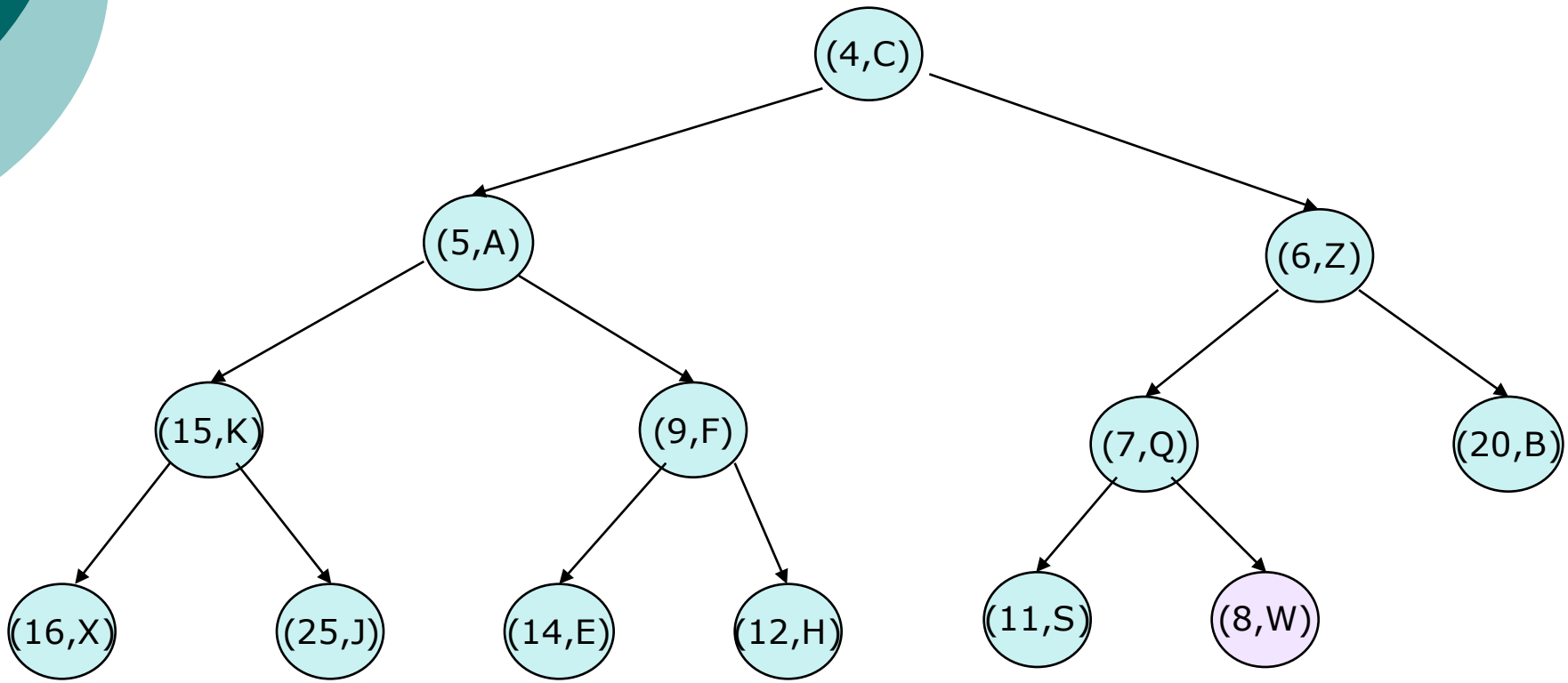
Insertion into the PQ implemented with Heap

- In order to store a new key-element pair (*k*,*e*) into *T*, we need to add a new node to *T*. To keep the complete tree property, the new node must become the last node of *T*.
- If a heap is implemented as a vector, the insertion node is added at index *n*+1, where *n* is the current size of the heap.

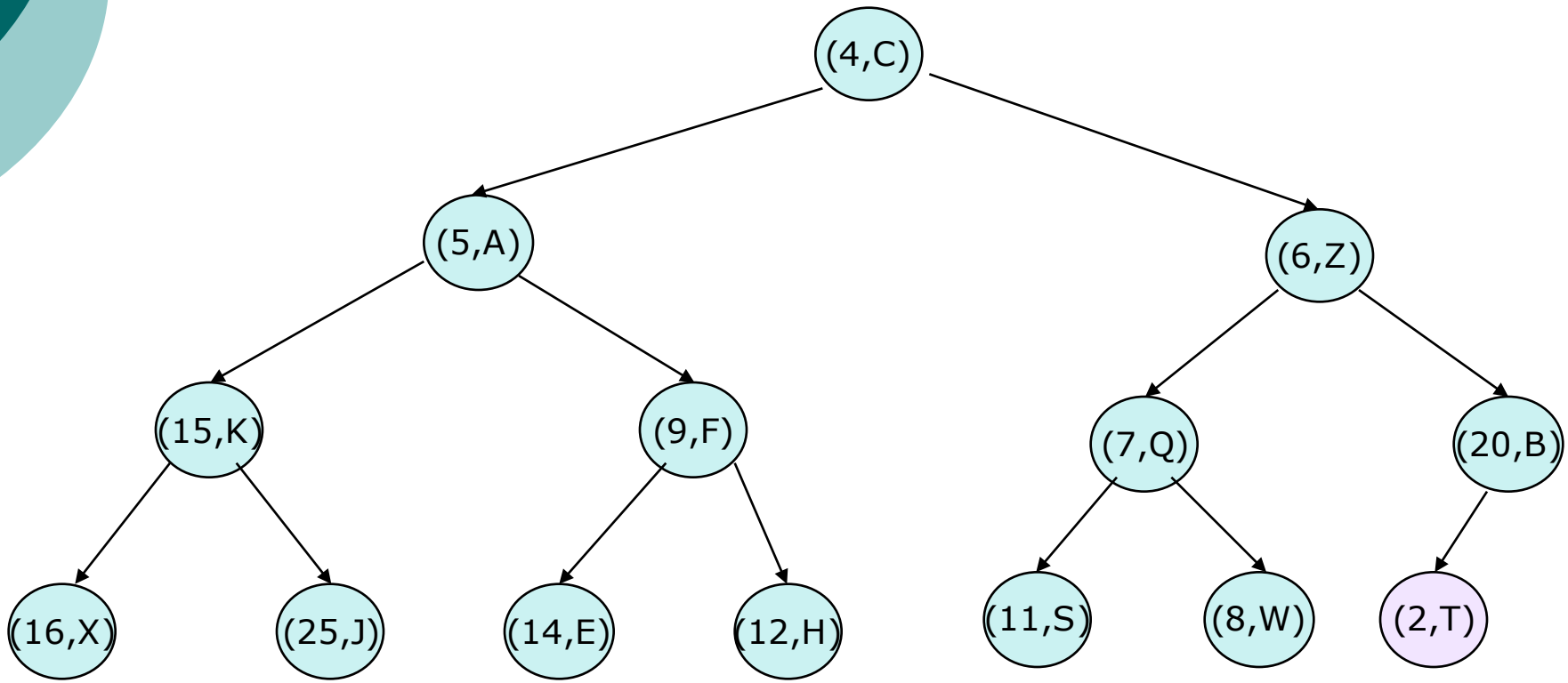
Up-heap bubbling after an insertion

- After the insertion of the element z into the tree T, it remains complete, but the heap-order property may be violated.
- Unless the new node is the root (the PQ was empty prior to the insertion), we compare keys $k(z)$ and $k(u)$ where u is the parent of z. If $k(u)>k(z)$, the heap order property needs to be restored, which can locally be achieved by swapping the pairs $(u,k(u))$ and $(z,k(z))$, making the element pair $(z,k(z))$ to go up one level. This upward movement caused by swaps is referred to as up-heap-bubbling.
- In the worst case the up-heap-bubbling may cause the new element to move all the way to the root.
- Thus, the worst case running time of the method insertItem is proportional to the height of T, i.e. $O(\log n)$, as T is complete.

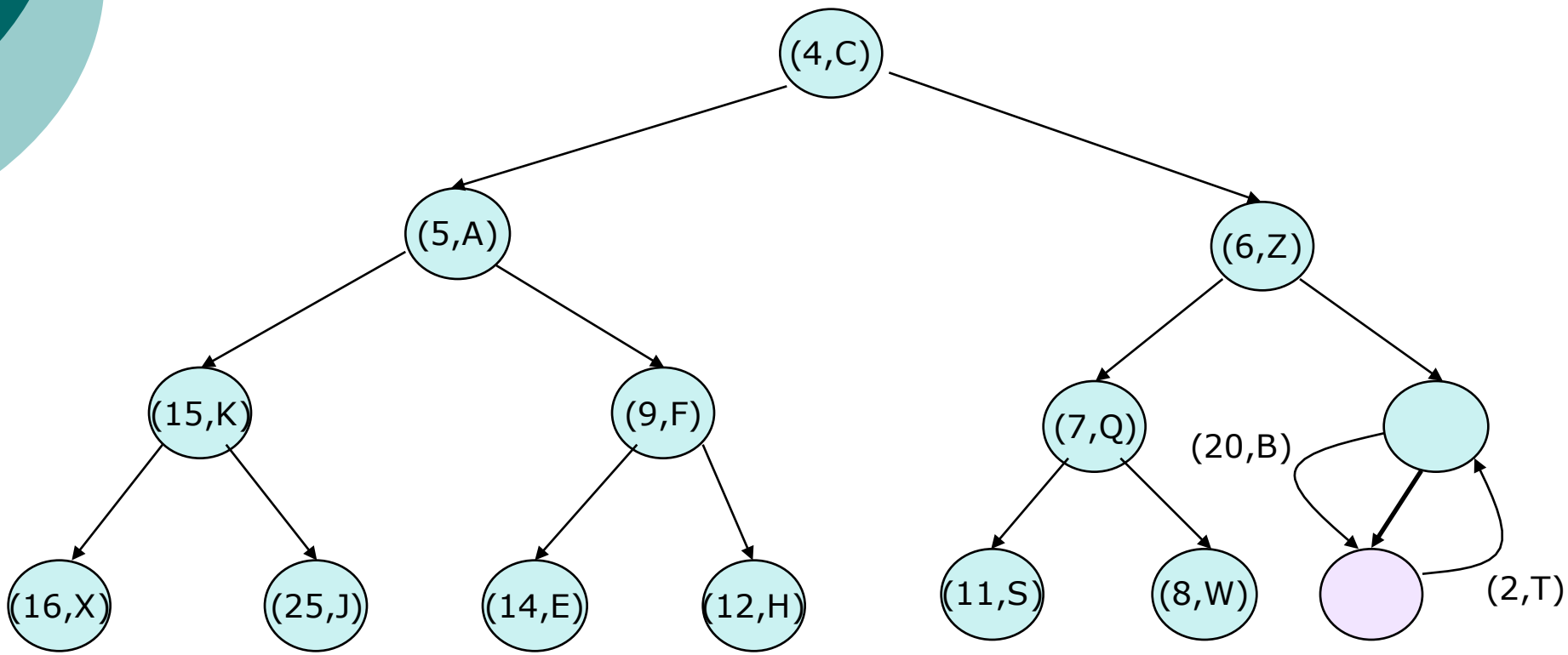
Up-heap bubbling after an insertion (an example)



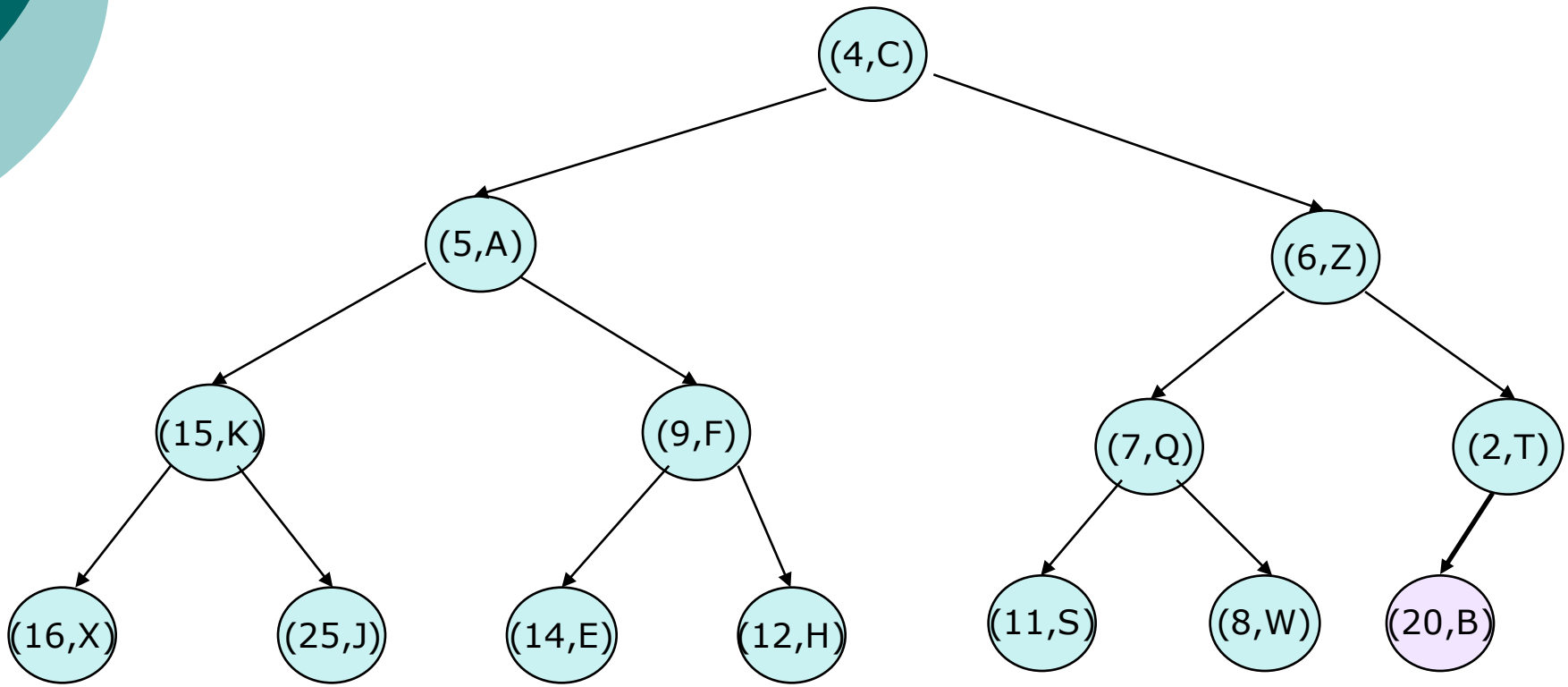
Up-heap bubbling after an insertion (an example)



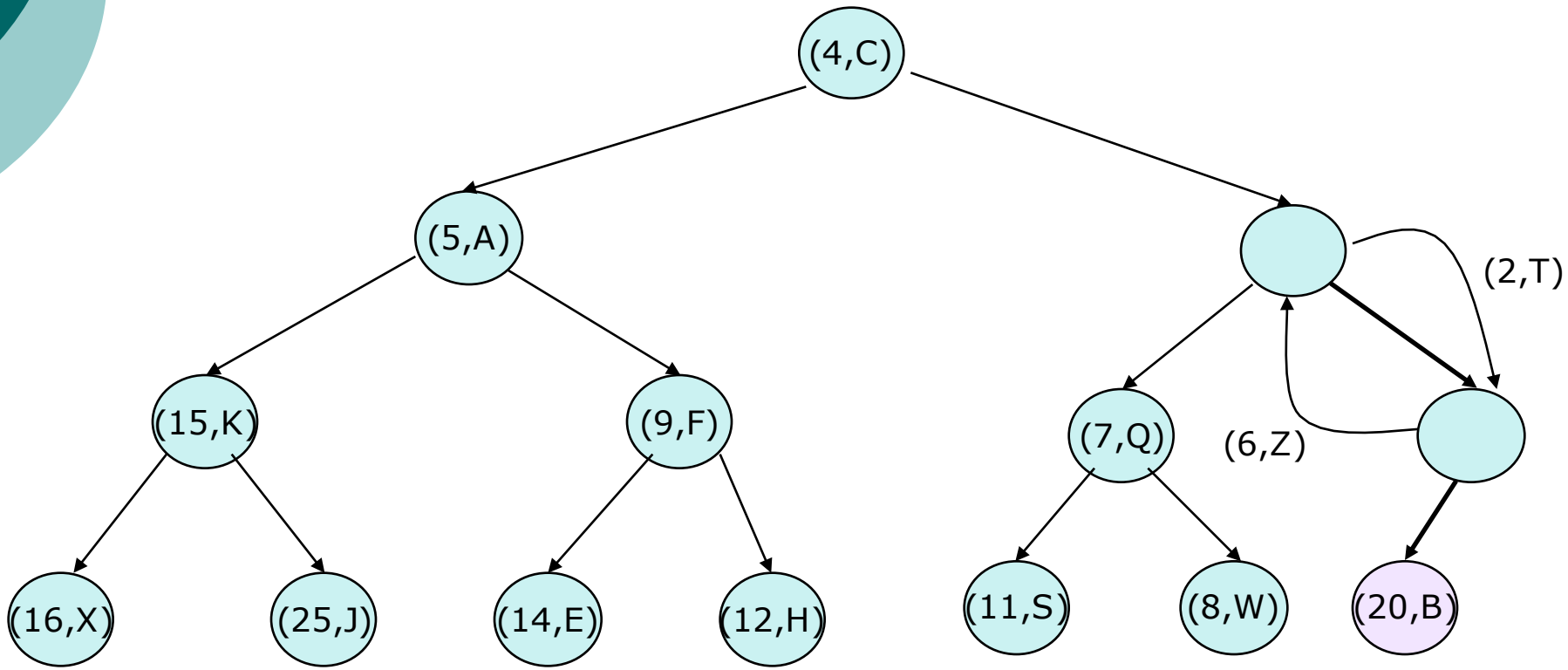
Up-heap bubbling after an insertion (an example)



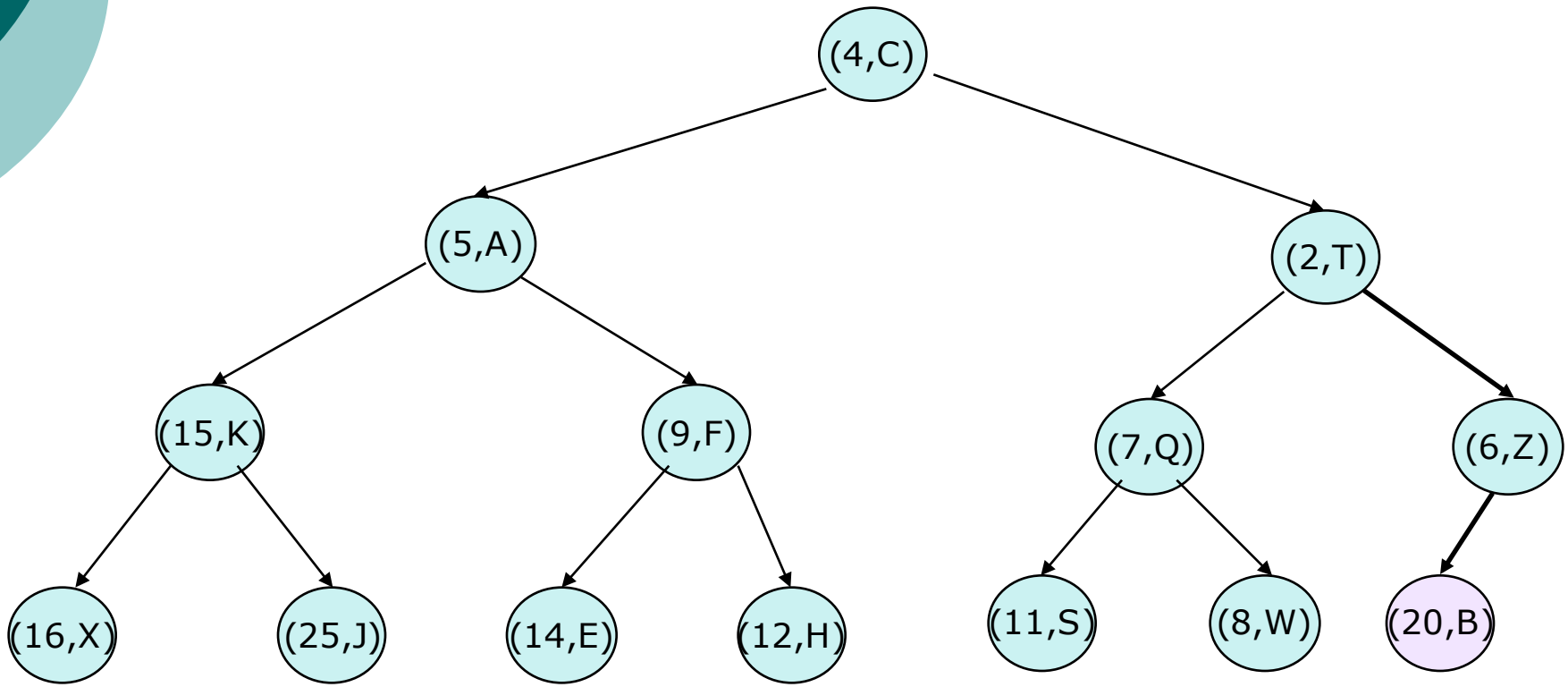
Up-heap bubbling after an insertion (an example)



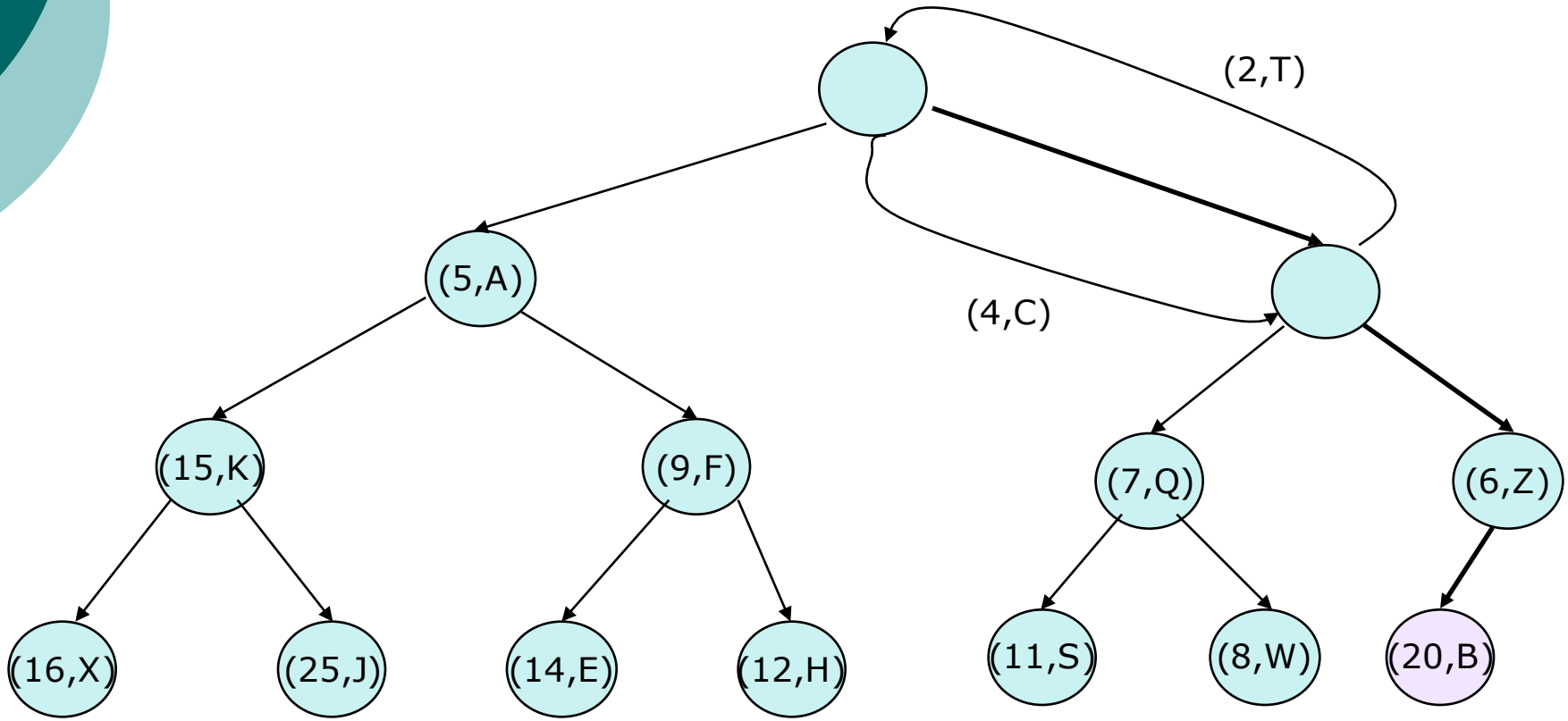
Up-heap bubbling after an insertion (an example)



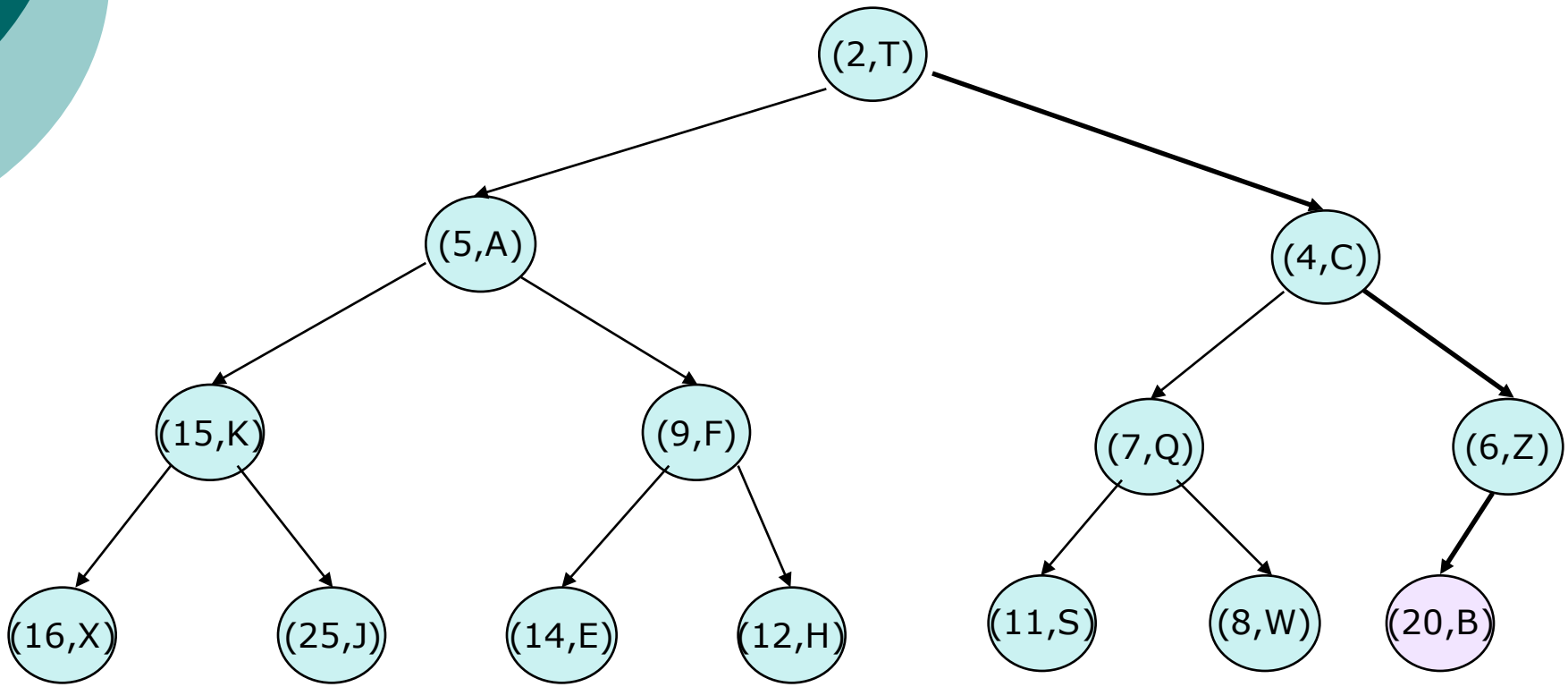
Up-heap bubbling after an insertion (an example)



Up-heap bubbling after an insertion (an example)



Up-heap bubbling after an insertion (an example)



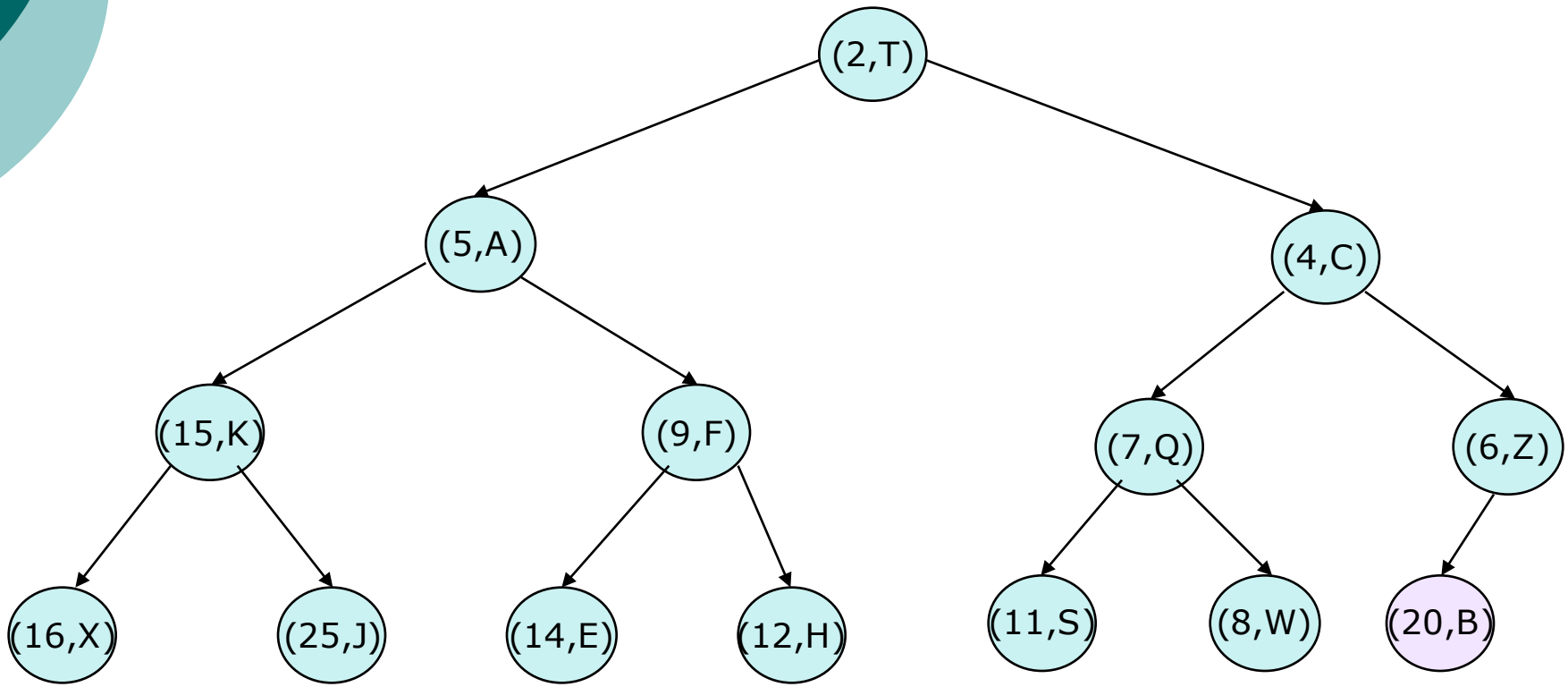
Removal from the PQ implemented as a Heap

- We need to perform the method removeMin from the PQ.
- The element *r* with a smallest key is stored at the root of the heap. A simple deletion of this element would disrupt the binary tree structure.
- We access the last node in the tree, copy it to the root, and delete it. This makes *T* complete.
- However, these operations may violate the heap-order property.

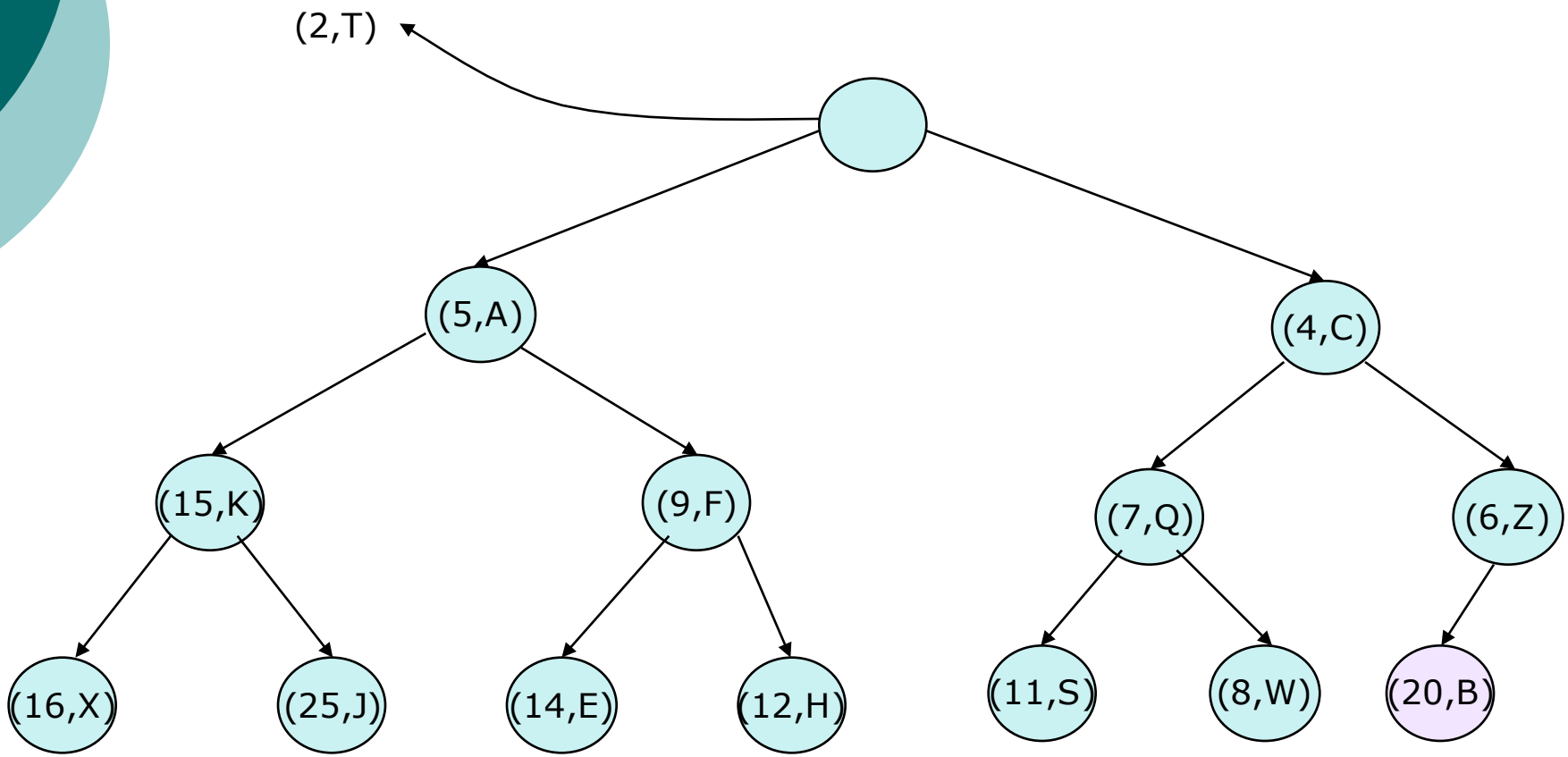
Removal from the PQ implemented as a Heap

- To restore the heap-order property, we examine the root *r* of *T*. If this is the only node, the heap-order property is trivially satisfied. Otherwise, we distinguish two cases:
  - If the root has only the left child, let *s* be the left child;
  - Otherwise, let *s* be the child of *r* with the smallest key;
- If $k(r)>k(s)$, the heap-order property is restored by swapping locally the pairs stored at *r* and *s*.
- We should continue swapping down *T* until no violation of the heap-order property occurs. This downward swapping process is referred to as down-heap bubbling. A single swap either resolves the violation of the heap-order property or propagates it one level down the heap.
- The running time of the method removeMin is thus $O(\log n)$.

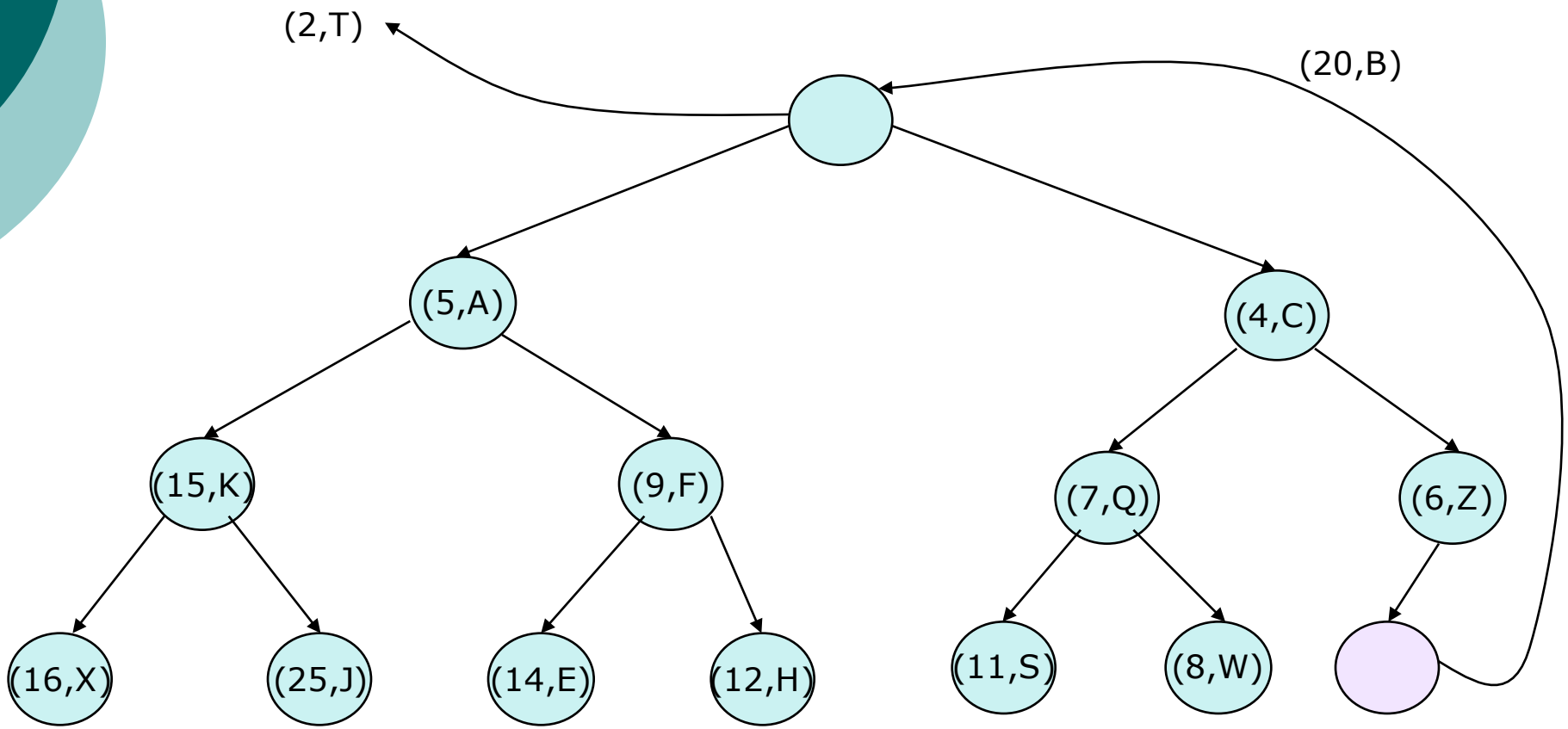
Down-heap bubbling after a removal (an example)



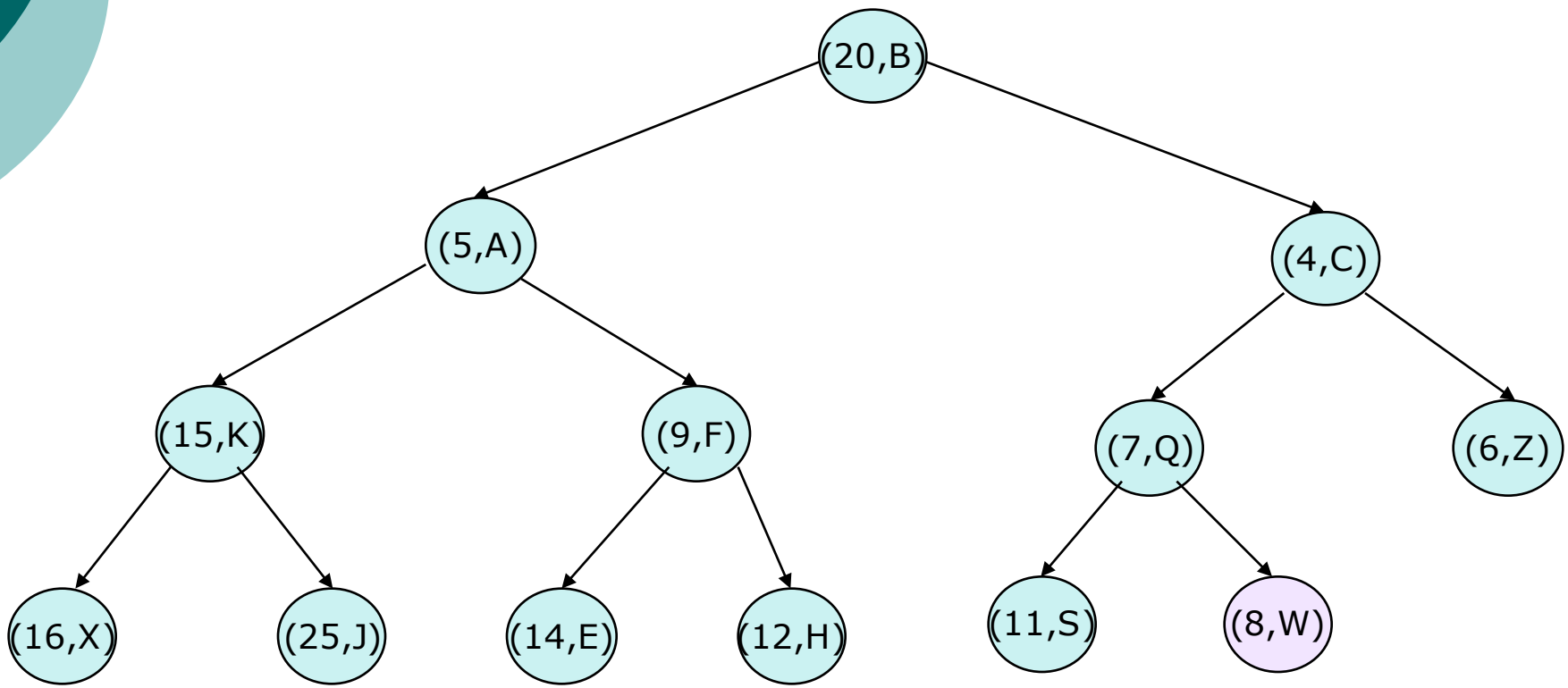
Down-heap bubbling after a removal (an example)



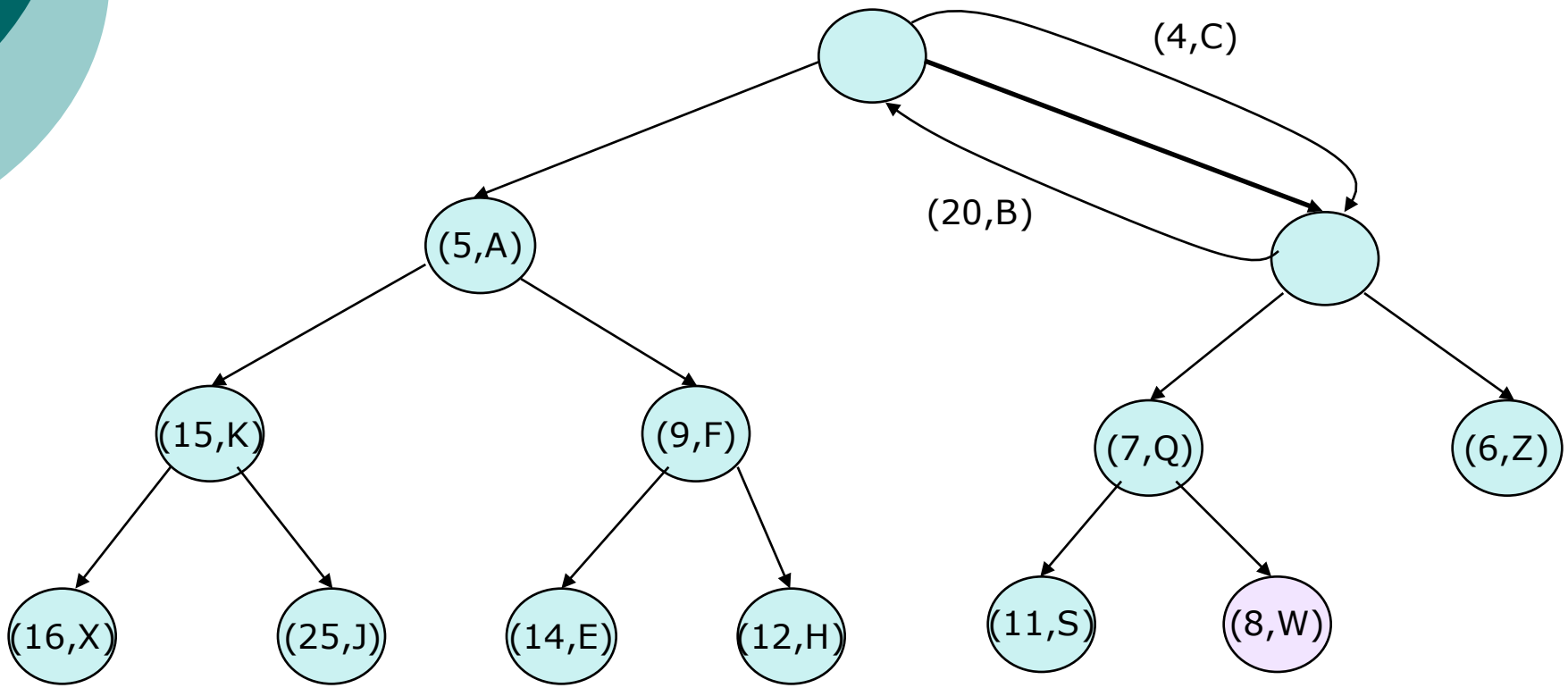
Down-heap bubbling after a removal (an example)



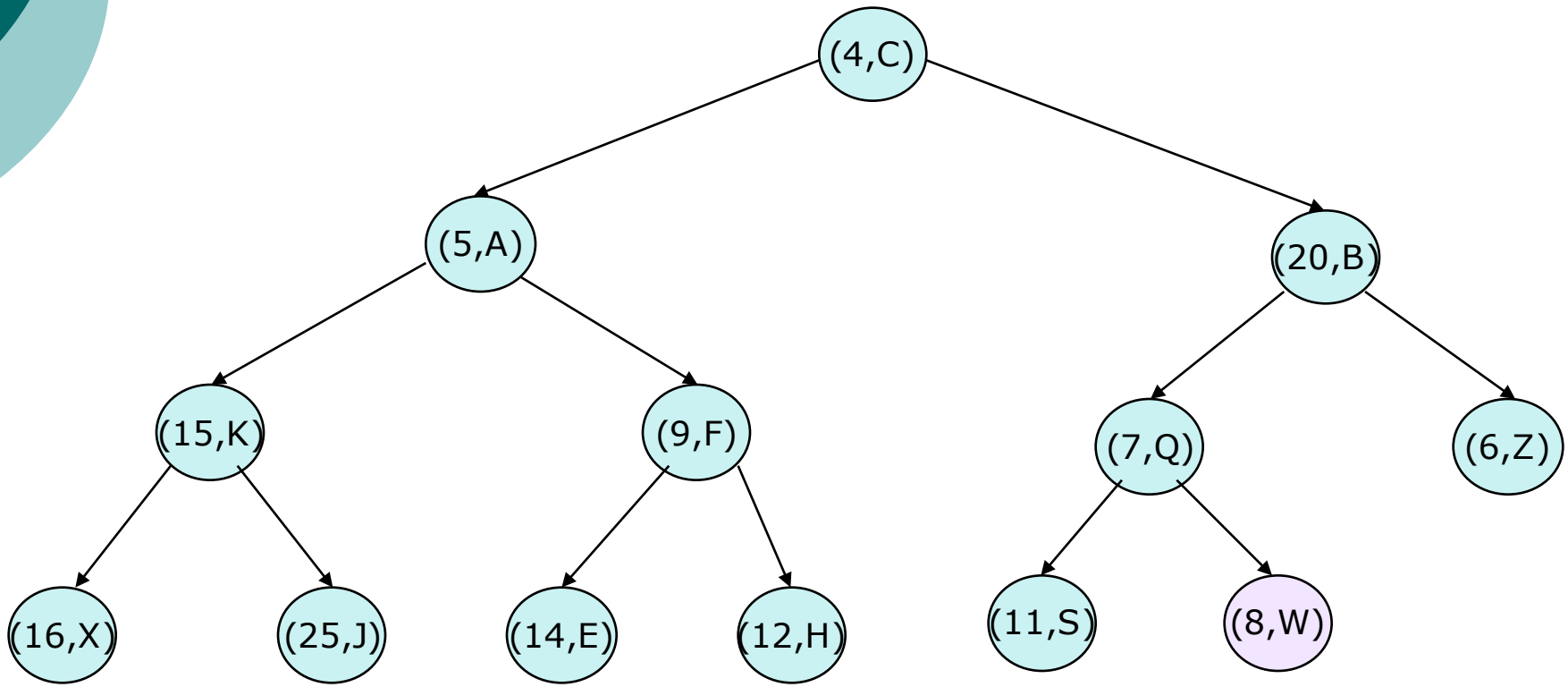
Down-heap bubbling after a removal (an example)



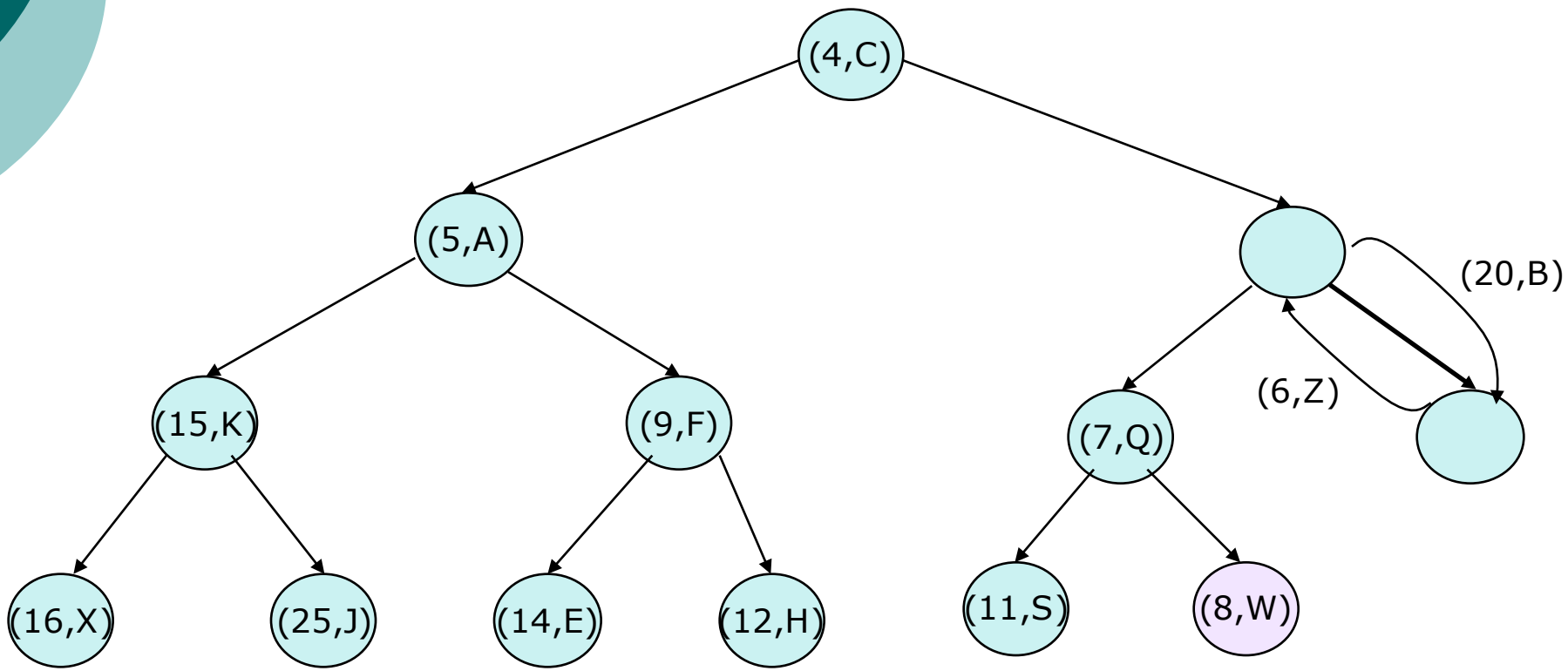
Down-heap bubbling after a removal (an example)



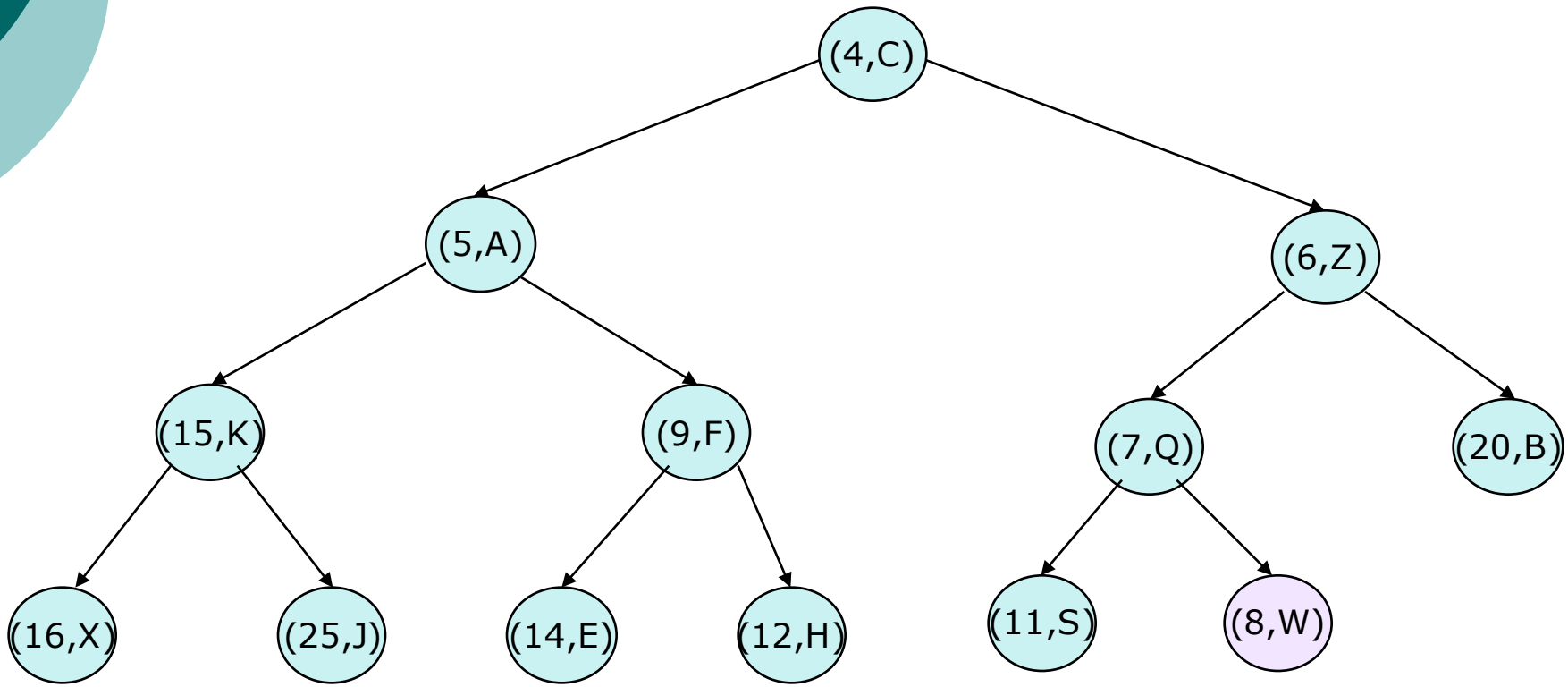
Down-heap bubbling after a removal (an example)



Down-heap bubbling after a removal (an example)



Down-heap bubbling after a removal (an example)



Heap-Sort

- Consider the PQ-Sort scheme that uses PQ to sort a sequence.
- If the PQ is implemented with a heap, during the insertion phase each of *n* insertItem operations takes $O(\log k)$, where *k* is the number of elements currently in the heap.
- During the second phase, each of *n* removeMin operations require $O(\log k)$ time.
- As $k \leq n$ at all times, the worst case run-time for each phase is equal to $O(\log n)$.
- This gives a total run time of a PQ sort algorithm as $O(n \log n)$ if a heap is used to implement a PQ. This sorting algorithm is known as **heap-sort**. This is a considerable improvement from selection-sort and insertion sort that both require $O(n^2)$ time.
- If the sequence *S* to be sorted is implemented as an array, then heap-sort can be implemented **in-place**, using one part of the sequence to store the heap.

Heap-Sort

We use the heap with the largest element on top. During the execution, left part of S ($S[0{:}i\text{-}1]$) is used to store the elements of the heap, and the right portion ($S[i{+}1,n]$). In the heap part of S, the element at the position k is greater or equal to its children at the positions $2k{+}1$ and $2k{+}2$.

- In the first phase of the algorithm, we start with an empty heap, and move the boundary between the heap part and the sequence part from left to right one element at the time (at the step i we expand the heap by adding an element at the rank i-1.
- In the second phase of the algorithm we start with an empty sequence and move the boundary between the heap and the sequence from right to left, one element at the time (at the step i we remove the maximum element from the heap and store it in the sequence part at the rank n-i.

Dictionaries and Hash Tables

- A computer dictionary is a data repository designed to effectively perform the search operation. The user assigns keys to data elements and use them to search or add/remove elements.
- The dictionary ADT has methods for the insertion, removal and searching of elements.
- We store key-element pairs (*k*,*e*) called **items** into a dictionary.
- In a student database (containing student's name, address and course choices, for example) a key can be student's ID.
- There are two types of dictionaries:
  - Unordered dictionaries;
  - Ordered dictionarties;

Unordered dictionaries

- In the most general case we can allow multiple elements to have the same key. However, in some applications this should not be advantageous (e.g. in a student database it would be confusing to have several different students with the same ID).
- In the cases when keys are unique, a key associated to an object can be regarded as an address of this object. Such dictionaries are referred to as **associative stores**.
- As an ADT a dictionary *D* supports the following methods:
  - **findElement**(*k*) – if *D* contains an item with key *k*, return an element *e* of such an item, else return an exception NO_SUCH_KEY.
  - **insertItem**(*k*,*e*) – insert an item with element *e* and key *k* in *D*
  - **RemoveElement**(*k*) – remove from *D* an item with key *k* and return its element, else return an exception NO_SUCH_KEY.

Log Files

- One simple way of realising an unordered dictionary is by an unsorted sequence S, implemented itself by a vector or list that store (such implementation is referred to as a **log file** or **audit trial**).
- This implementation is suitable for storing a small amount of data that do not change much over time.
- With such implementation, the asymptotic execution times for dictionary methods are:
  - insertItem – is implemented via insertLast method on S in $O(1)$ time;
  - findElement – is performed by scanning the entire sequence which takes $O(n)$ in the worst case;
  - removeElement – is performed by searching the whole sequence until the element with the key k is found, again the worst case is $O(n)$;

Hash Tables

- If the keys represent the "addresses" of the elements, an effective way of implementing a dictionary is to use a **hash table**.
- The main components of a hash table are the **bucket arrays** and the **hash functions**.
- A bucket array for a hash table is an array A of size N, where each "element" of A is a container of key-element pairs, and N is the capacity of the array.
- An element e with the key k is inserted into the bucket $A[k]$.
- If keys are not unique, two different elements may be mapped to the same bucket in A (a **collision** has occurred).

Hash Tables

There are two major drawbacks of the proposed concept:

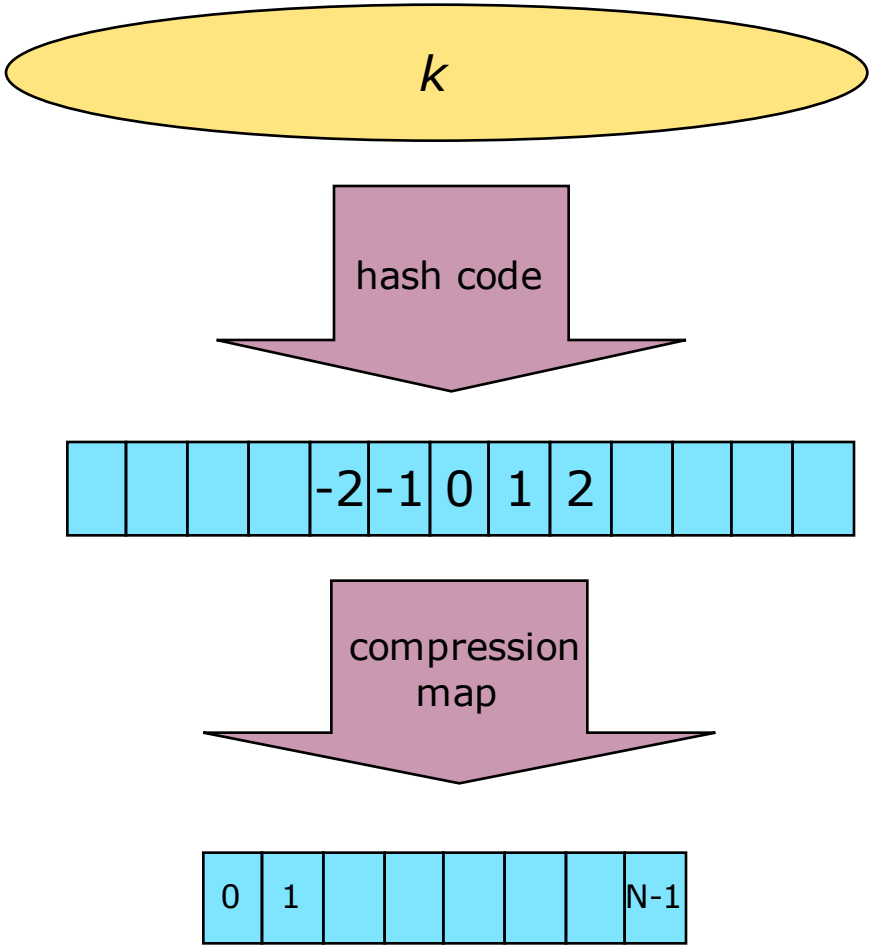
- The space $O(N)$ is not necessarily related to the number of items n (if N is much greater than n, considerable amount of space is unused);
- The bucket array requires the keys to be unique integers in the range $[0,n-1]$, which is not always the case;

- To address these issues, the hash table data structure has to be defined as a bucket array, together with a good mapping from the keys to the range $[0,n-1]$;

Hash Functions

- A **hash function** *h* maps each key *k* from a dictionary to an integer in range [0,*N*-1], where *N* is the bucket capacity.
- Instead of using *k* as an index of the bucket array, we use *h*(*k*), i.e. we store the item (*k*,*e*) in the bucket *A*[*h*(*k*)].
- A hash function is deemed to be "good" if it minimises collisions as much as possible, and at the same time eavaluating *h*(*k*) is not expensive.
- The evaluation of a hash function consists of two phases:
  - Mapping *k* to an integer (the **hash code**);
  - Mapping the hash code to be an integer within the range of indices in a bucket array (the **compression map**);

Hash Functions



Hash Codes

- The first action performed on an (arbitrary) key *k* is to assign to it an integer value (called the **hash code** or **hash value**). This integer does not have to be in the range [0,*N*-1], but the algorithm for assigning it should avoid collisions.
- If the range of values of *k* is larger than that assumed in the hash code (e.g. *k* is a long integer and the hash value is a short integer).
- One way of creating a hash code in such situation is to take only lower or higher half of *k* as a hash code.
- Another option is to sum lower and upper half of *k* (the summation hash code).

Polynomial Hash Codes

- The summation hash code is not a good choice for keys that are character strings or other multiple-length objects of the form $(x_0, x_1, \ldots, x_{k-1})$.
- An example: Assume that x_i 's are ASCII characters and that the hash code is defined as

$$h(k) = \sum_{i=0}^{k-1} \text{ASCII}(x_i)$$

Such approach would produce many unwanted collisions, such as:

temp 01
temp 10

tops
pots
stop
spot

Polynomial Hash Codes

- A better hash code would take into account both the values x_i and their positions *i*. By choosing a constant a $(a \neq 0,1)$, we can define the hash code:

$$h = x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$$

  This is a polynomial in *a* with the coefficients $(x_0, x_1, \ldots, x_{k-1})$.
- Hence, this is the polynomial hash code.
- Experimental studies show that good spreads of hash codes are obtained with certain choices for *a* (for example, 33,37,39,41);
- Tested on the case of 50,000 English words, each of these choices provided fewer than 7 collisions.

Compression Maps

If the range of hash codes generated for the keys exceeds the index range of the bucket array, the attempt to write the element would cause an out-of-bounds exception (either because the index is negative, or out of range [0,*N*-1].

- The process of mapping an arbitrary integer to a range [0,*N*-1] is called compression, and is the second step in evaluating the hash function.
- A compression map that uses:

$$h(k) = |k| \bmod N$$

is called the **division method**. If *N* is a prime number, the division compression map may help spreading out the hashed values. Otherwise, there is a likelihood that patterns in the key distributions are repeated (e.g. the keys {200,205,210,…,600} and *N*=100.

Compression Maps

- A more sophisticated compression function is the **multiply add and divide** (or MAD method), with the compression function:

$$h = |ak + b| \bmod N$$

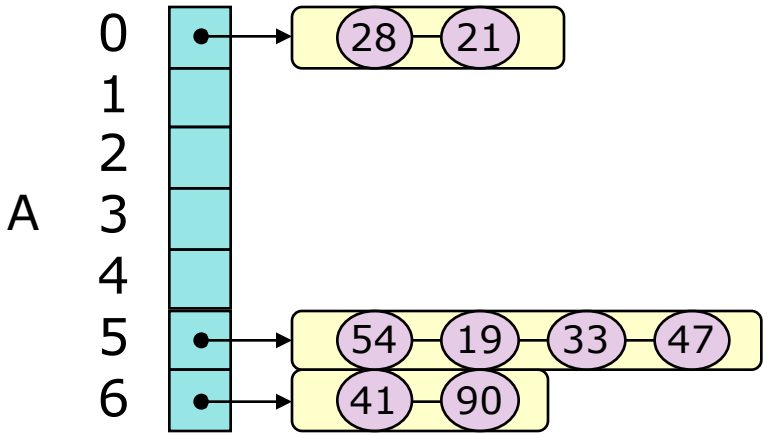
  where *N* is a prime number, and *a,b* are random non-negative integers selected at the time of compression, such that $a \bmod N \neq 0$.
- With this choice of compression function the probability of collision is at most 1/*N*.

Collision Handling Schemes

- The collision occurs when for two distinct keys k_1, k_2 we have $h(k_1) = h(k_2)$. This prevents simple insertion of a new item (k,e) into a bucket $A[h(k)]$.
- A simple way of dealing with collisions is to make each bucket capable of storing more than one item. This requires each bucket $A[i]$ to be implemented as a sequence, list or vector. This collision resolution rule is known as **separate chaining**. Each bucket can be implemented as a miniature dictionary.

(41,28,54,19,33,21,90,47)

$$h(k) = |k| \bmod 7$$



Load Factors and Rehashing

- If a good hashing function is used for storing *n* items of a dictionary in a bucket of size *N*, we expect each bucket to be of size ~*n*/*N*. This parameter (called **load factor**) should be kept small. If this requirement is fulfilled, the running time of methods findElement, insertItem and removeElement is $O(n/N)$
- In order to keep the load factor below a constant the size of bucket array needs to be increased and to change the compression map. Then we need to re-insert all the existing hash table elements into a new bucket array using the new compression map. Such size increase and the hash table rebuild is called **rehashing**.

Open Addressing

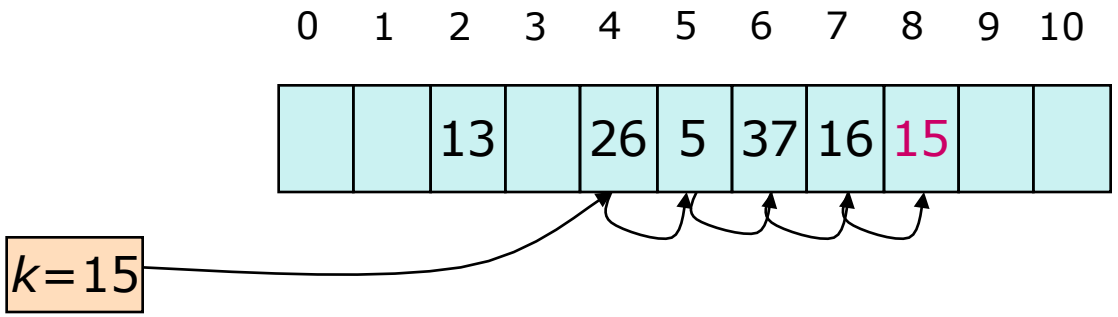
- Separate chaining allows simple implementation of dictionary operations, but requires the use of an auxiliary data structure (a list, vector or sequence).
- An alternative is to store only one element per bucket. This requires a more sophisticated approach of dealing with collisions.
- Several methods for implementing this approach exist referred to as **open addressing**.

Linear Probing

- In this strategy, if we try to insert an item (k,e) into a bucket $A[h(k)]=A[i]$ that is already occupied, we try to insert an element in the positions $A[(i+j) \bmod N]$ for $j=1,2,\ldots$ until a free place is found.
- This approach requires re-implementation of the method findElement(k). To perform search we need to examine consecutive buckets starting from $A[h(k)]$ until we either find an element or an empty bucket.
- The operation removeElement(k) is more difficult to implement. The easiest way to implement it is to introduce a special "deactivated item" object.
- With this object, findElement*(k)* and removeElement(k) should be implemented should skip over deactivated items and continue probing until a desired item or an empty bucket is found.

Linear Probing

- The algorithm for inserItem(*k*,*e*) should stop at the deactivated item and replace it with *e*.
- In conclusion, linear probing saves space, but complicates removals. It also clusters of a dictionary into contiguous runs, which slows down searches.



An insertion into a hash table with linear probing.
The compression map is $h(k) = k \bmod 11$

Quadratic Probing

- Another open addressing strategy (known as **quadratic probing**) involves iterative attempts to buckets $A[(i + j^2) \bmod N]$ for j=0,1,2,… until an empty bucket is found.
- It complicates the removal operation, but avoids clustering patterns characteristic with linear probing.
- However, it creates **secondary clustering** with a set of secondary cells bouncing around a hash table in a fixed pattern.
- If *N* is not a prime number, quadratic probing may fail to find an empty bucket, even if one exists. If the bucket array is over 50% full, this may happen even if *N* is a prime.

Double Hashing

- Double hashing strategy eliminates the data clustering produced by the linear or quadratic probing.
- In this approach a secondary hash function *h′* is used. If the primary function *h* maps a key *k* to a bucket *A*[*h*(*k*)] that is already occupied, the the following buckets are attempted iteratively:

$$A[(i + j \cdot h'(k))], \quad j = 1,2,...$$

- The second hash function must be non-zero. For *N* a prime number, the common choice is

$$h'(k) = q - (k \bmod q)$$

with $q<N$ a prime number.

- If memory size is not an issue, the separate chaining is always competitive to collision handling schemes based on open addressing.

COMP20010: Algorithms and Imperative Programming

Lecture 4

Ordered Dictionaries and Binary Search Trees

AVL Trees

Lecture outline

- Sorted tables;
- Binary search trees
- Searching, inserting and removal with binary search trees;
- Performance of binary search trees;
- AVL trees (update operations, performance);

Ordered dictionaries

- In an ordered dictionary we use a comparator to provide the order relation among the keys. Such ordering allows efficient implementation of the dictionary ADT.
- An ordered dictionary supports the following methods:
  - closestKeyBefore(k) – returns the largest key that is $\leq k$;
  - closestElemBefore(k) – returns e with the largest key that is $\leq k$;
  - closestKeyAfter(k) – returns the smallest key that is $\geq k$;
  - closestElemAfter(k) – returns e with the smallest key that is $\geq k$;
- The ordered nature of the above operations makes the use of a log file or a hash table inappropriate for implementing the dictionary (neither of the two data structures has any ordering among the keys).

Sorted Tables

- If a directory D is ordered, the items can be stored in a vector S by non-decreasing order of the keys.
- The ordering of the keys allows faster searching than in the case of un-ordered sequences (possibly implemented as a linked list).
- The ordered vector implementation of a dictionary D is referred to as the **lookup table**.
- The implementation of insertItem(k,e) in a lookup table takes $O(n)$ time in the worst case, as we need to shift up all the items with keys greater than k to make room for the new item.
- On the other hand, the operation findElement is much faster in a sorted lookup table than in a log file.

Binary Search

- If *S* is an ordered sequence, than the element at the rank (position) *i* has a key that is not smaller than the keys of the items at ranks $0,\ldots,i-1$ and no larger than the keys of the items at ranks $i+1,\ldots.n-1$.
- This ordering allows quick searching of the sequence *S* using a variant of the game "high-low". The algorithm has two parameters: **high** and **low**. All the candidates for a sought element at a current stage of the search are bracketed between these two parameters, i.e. they lie in the interval [low,high].
- The algorithm starts with the values low=0 and high=*n*-1. Then the key *k* of the element we are searching for is compared to a key of the element at a half of *S*, i.e. $\text{mid} = \lfloor(\text{low}+\text{high})/2\rfloor$. Depending on the outcome of this comparison we have 3 possibilities:
  - If *k*=key(mid), the item we are searching for is found and the algorithm terminates returning *e*(mid);
  - If *k*<key(mid), then the element we are searching for is in the lower half of the vector *S*, and we set high=mid-1 and call the algorithm recursively;
  - If *k*>key(mid), the element we are searching for is in the upper half of the vector *S*, and we set low=mid+1 and call the algorithm recursively;

Binary Search

- Operation findElement(*k*) on an *n*-item ordered dictionary implemented with a vector *S* reduces to calling BinarySearch(*S,k,0,n*-1).

  Algorithm BinarySearch(*S,k,*low,high):

  Input: An ordered vector *S* storing *n* items, a search key *k*, and the integers low and high;

  Output: An element of *S* with the key *k*, otherwise an exception;

  if low>high **then**

  return NO_SUCH_KEY

  else

  mid= $\lfloor (\text{low} + \text{high})/2 \rfloor$

  if *k*=key(mid) **then**

  return *e*(mid)

  elseif *k*<key(mid) **then**

  BinarySearch(*S,k,*low,mid-1)

  else

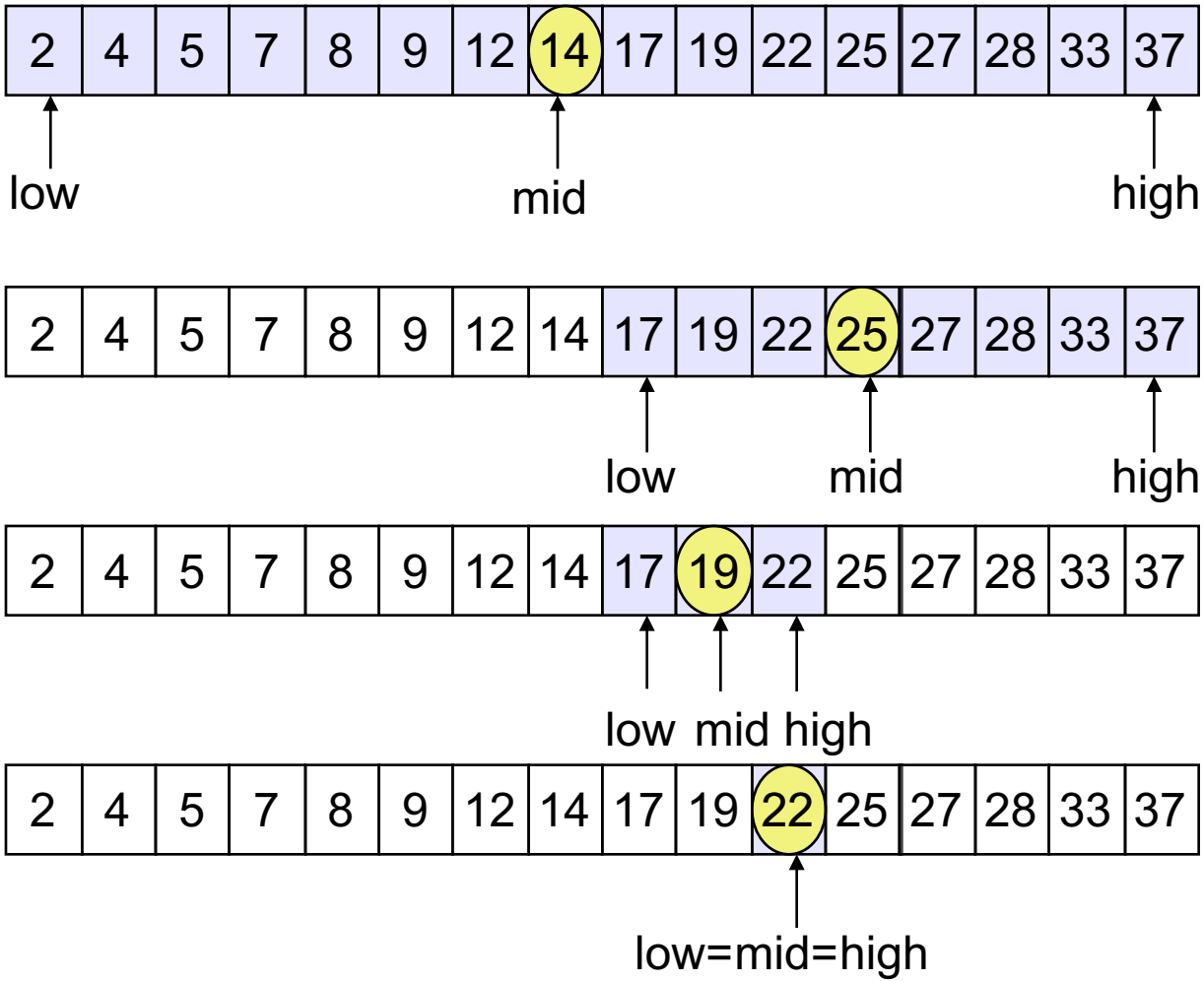
  BinarySearch(*S,k,*mid+1,high)

  endif

  end if

Binary Search

An example of binary search to perform the operation findElement(22)



Binary Search

- Considering the computational cost of binary search, we need to notice first that at each call of the algorithm there is a constant number of operations. Thus the running time is proportional to the number of recursive calls.
- At each recursive call the number of candidates that need to be searched is high-low+1, and it is reduced by at least a half at each recursive call.
- If $T(n)$ is the computational cost of binary search, then:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ T(n/2) + c & \text{otherwise} \end{cases}$$

- In the worst case the search stops when there are no more candidate items. Thus, the maximal number of recursive calls is m, such that

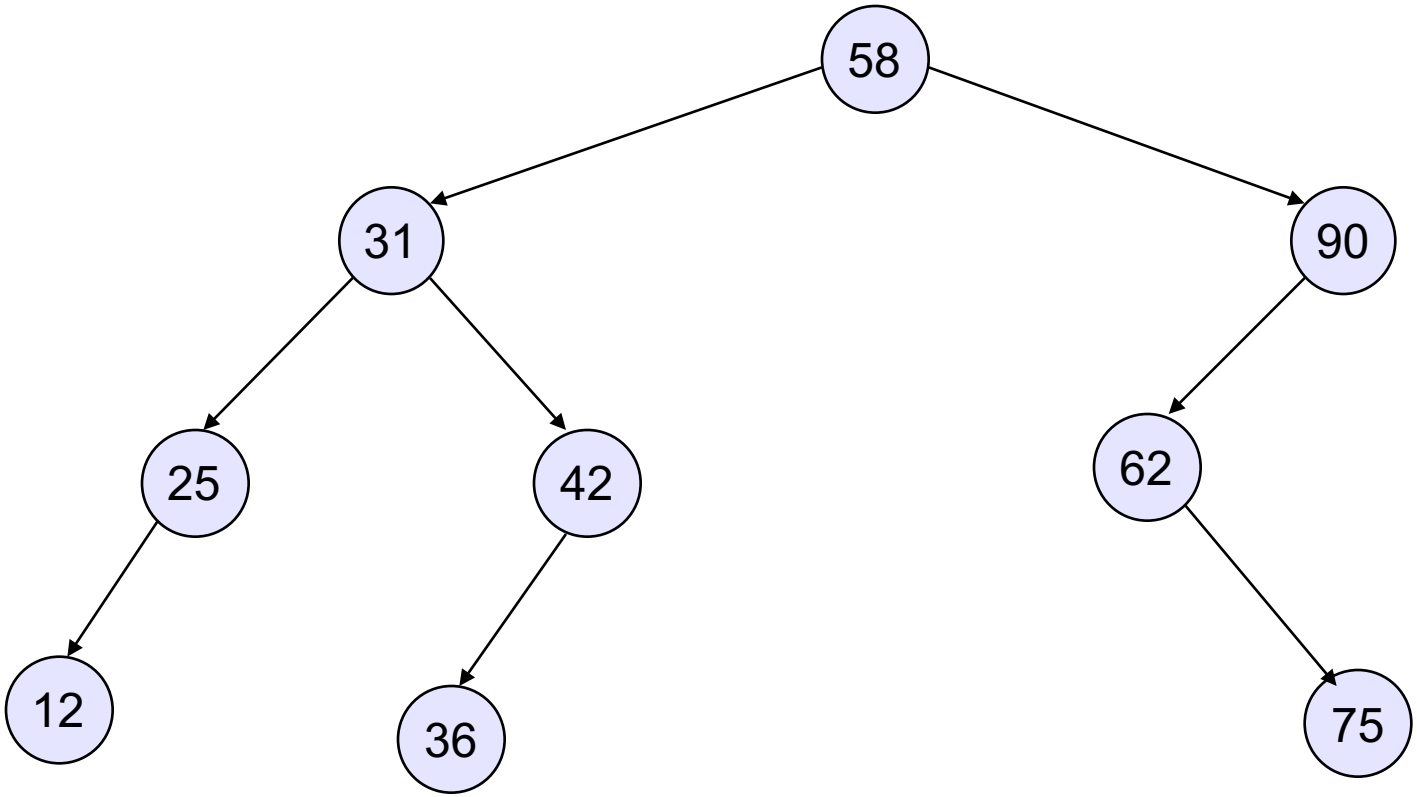
$$n/2^m < 1$$

- This implies that $m = \lfloor \log n \rfloor + 1$, i.e. BinarySearch($S$,$k$,0,$n$-1) runs in $O(\log n)$ time.

Binary Search Trees

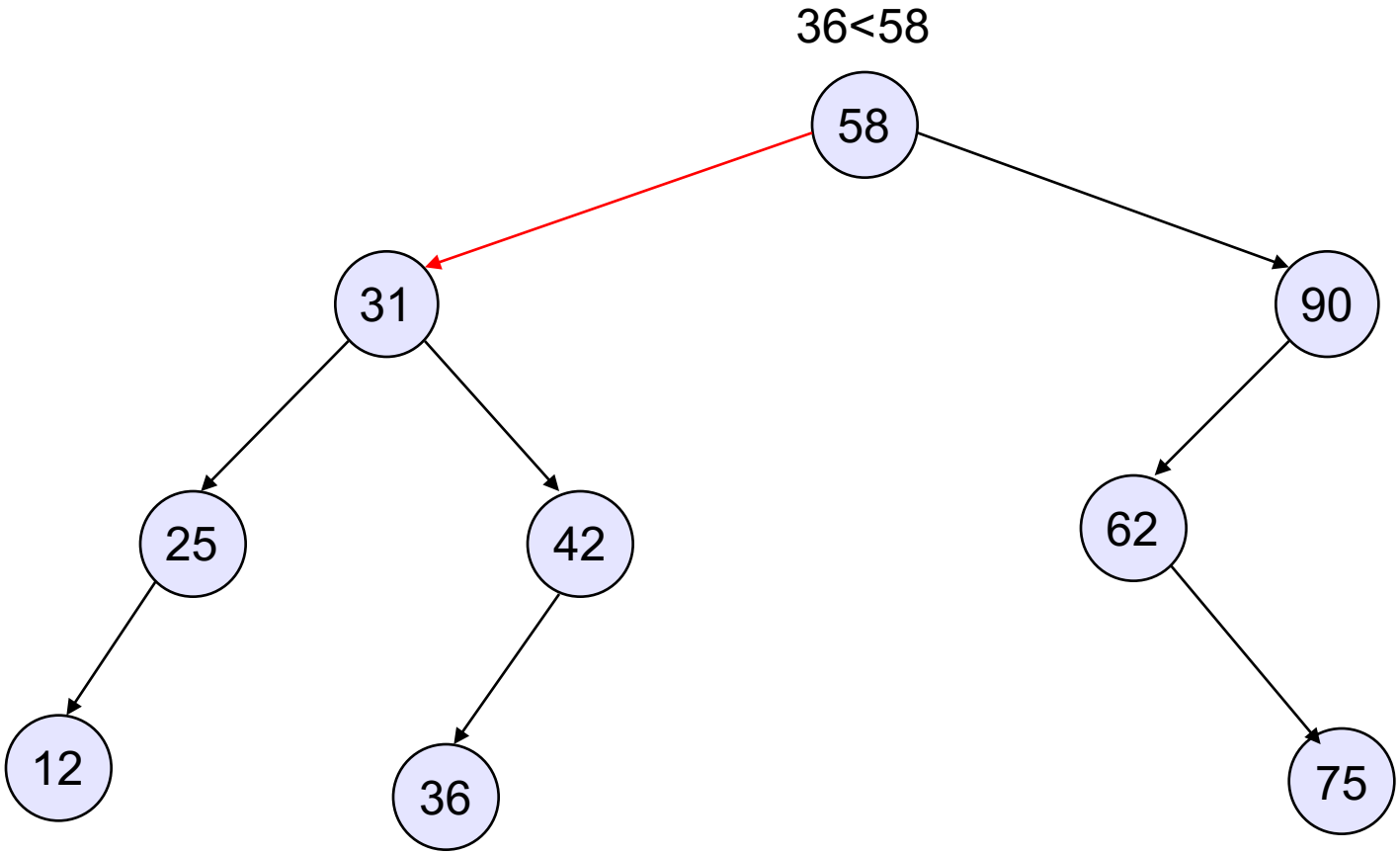
- It is a tree data structure adapted to a binary search algorithm.
- A binary search tree is a binary tree in which each node stores an element *e* and that the elements in the left subtree of that node are smaller or equal to *e*, while the elements in the right subtree of that node are greater or equal to *e*.
- An inorder traversal of a binary search tree visits the elements in a non-decreasing order.
- A binary search tree can be used to search for an element by traversing down the tree. At each node we compare the value we are searching for *x* with *e*. There are 3 outcomes:
  - If $x=e$, the search terminates successfully;
  - If $x<e$, the search continues in the left subtree;
  - If $x>e$, the search continues in the right subtree;
  - If the whole subtree is visited and the element is not found, the search terminates unsuccessfully;

Binary Search Trees



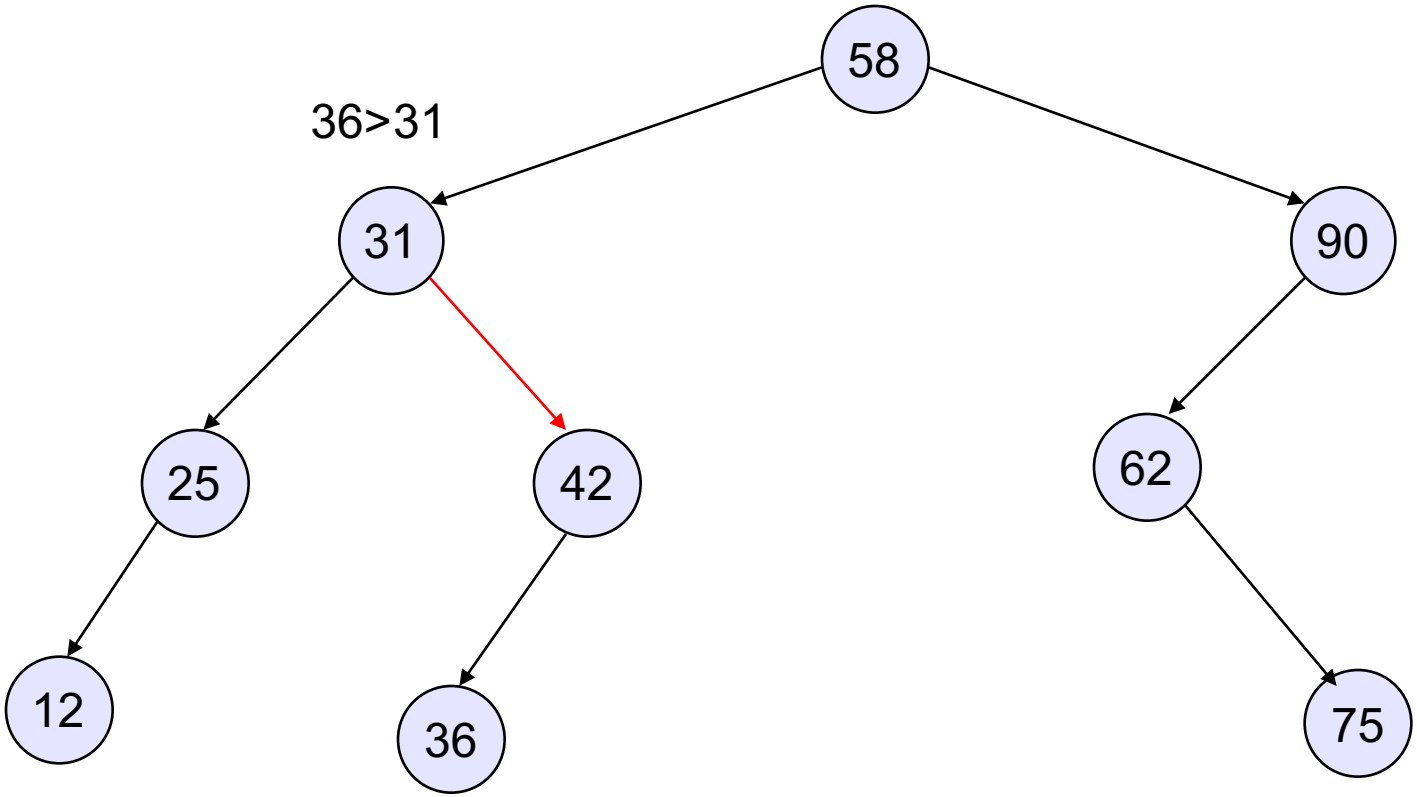
Searching for the element 36

Binary Search Trees



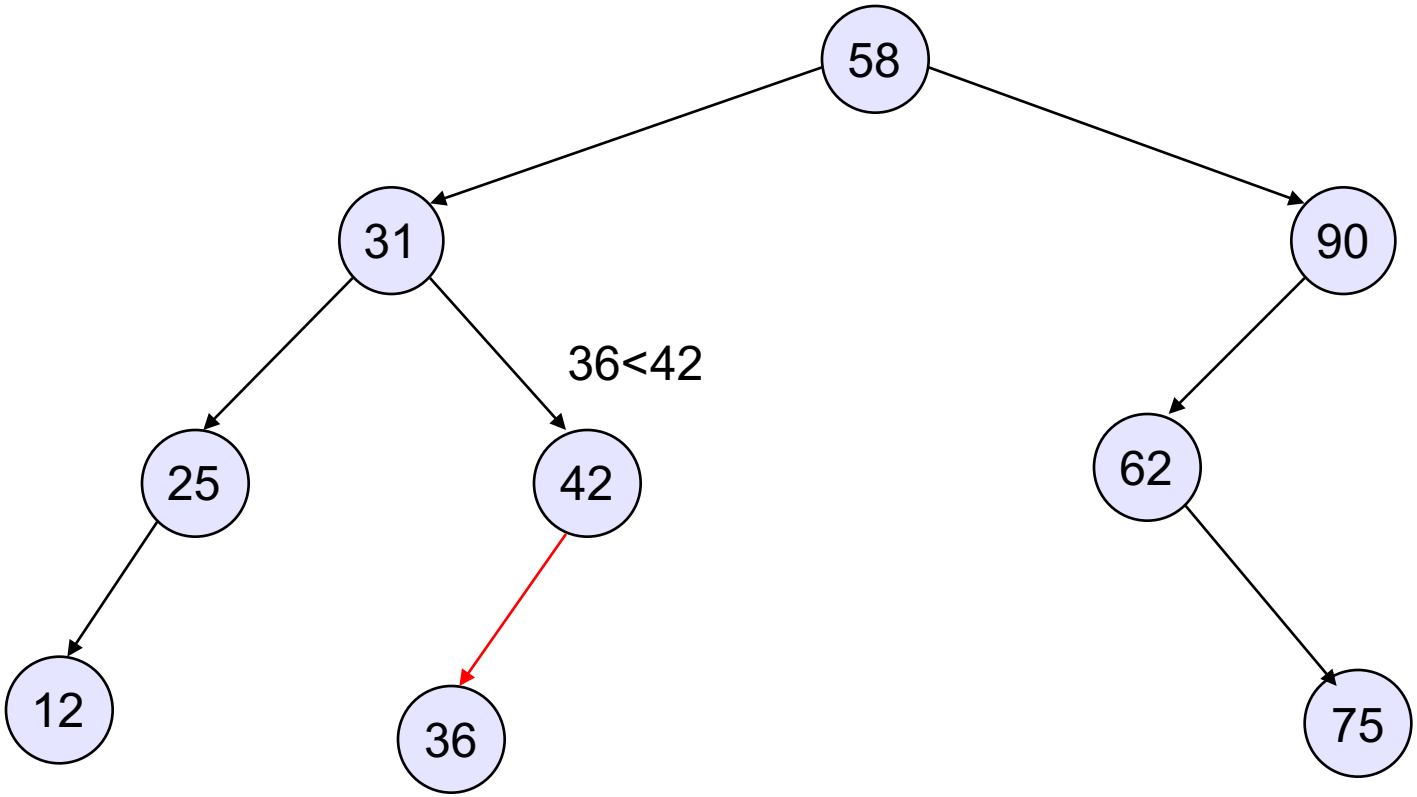
Searching for the element 36

Binary Search Trees



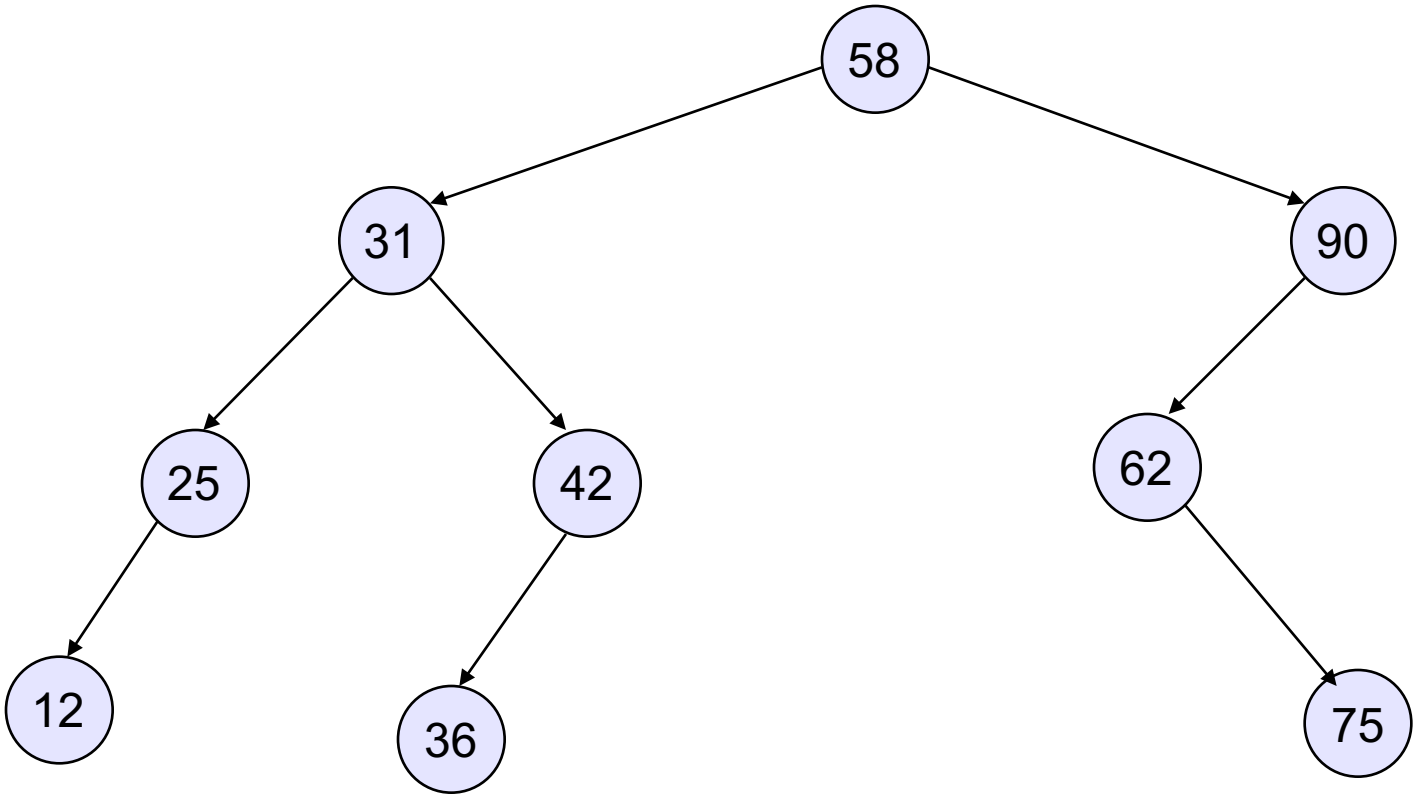
Searching for the element 36

Binary Search Trees



Searching for the element 36

Binary Search Trees



36=36 - success

Searching for the element 36

Computational Cost of Binary Tree Searching

- The binary tree search algorithm executes a constant number of operations for each node during the traversal.
- The binary search algorithm starts from the root and goes down one level at the time.
- The number of levels in a binary search tree is called the height *h*.
- The method findElement runs in O(*h*) time. This can potentially be a problem as *h* can potentially be close to *n*. Thus, it is essential to keep the tree height optimal (as close to $O(\log n)$ as possible). The way to achieve this is to balance a tree after each insertion (AVL trees).

Dictionary Search Using a Binary Search Tree

- The method findElement(*k*) can be performed on a dictionary *D* if we store *D* as a binary search tree and call the method TreeSearch(*k, T.*root()).

Algorithm TreeSearch(*k,v*);

Input: A search key *k* and a node *v* of a binary search tree;

Output: A node *w* of *T* equal to *k*, or an exception;

if *k*=key(*v*) **then return** *v*;

else if *k* is an external node **then return** NO_SUCH_KEY;

else if *k*<key(*v*) **then return** TreeSearch(*k, T.*leftChild(*v*));

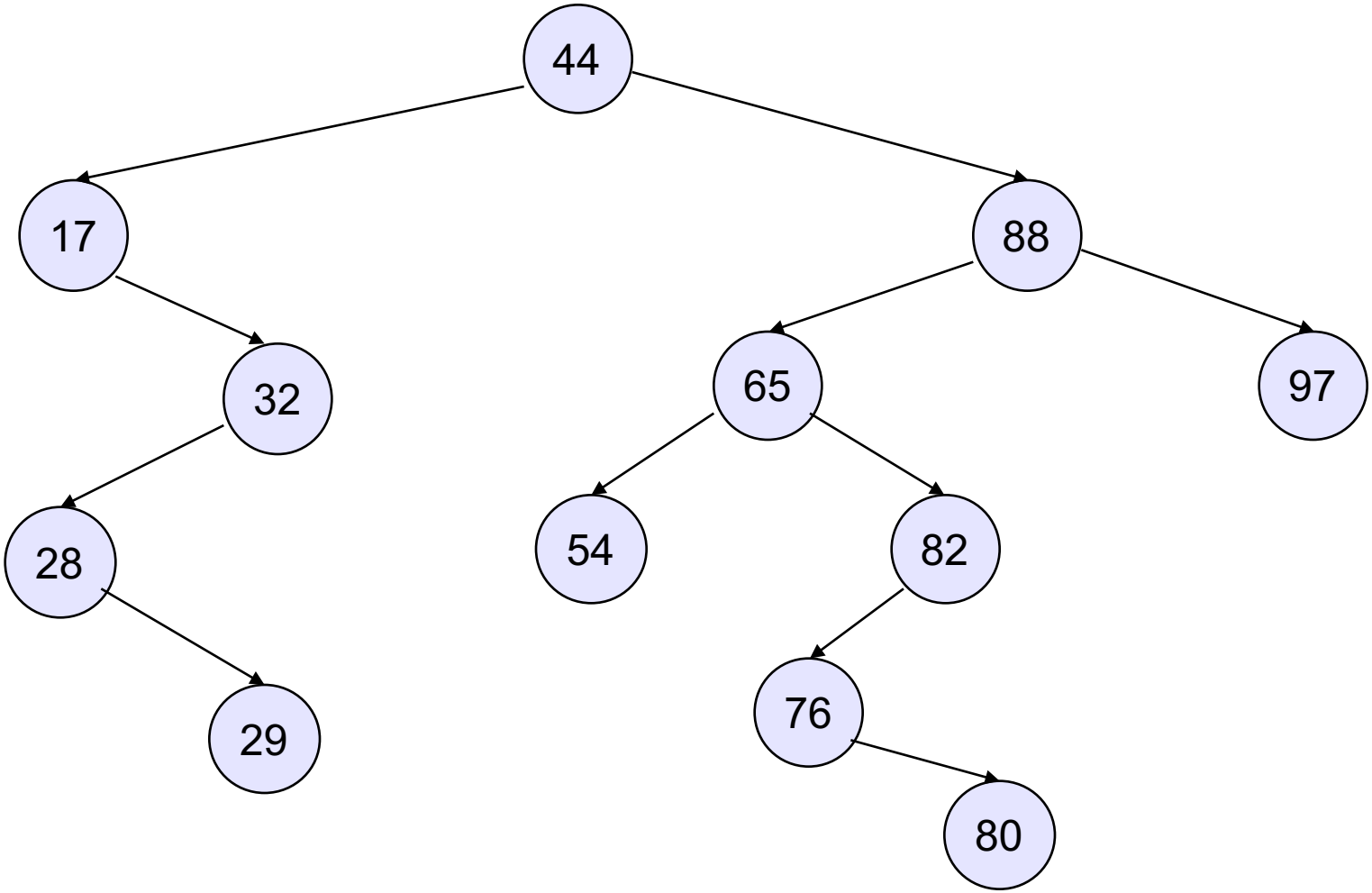
else return TreeSearch(*k, T.*rightChild(*v*));

end if

Insertion into a Binary Search Tree

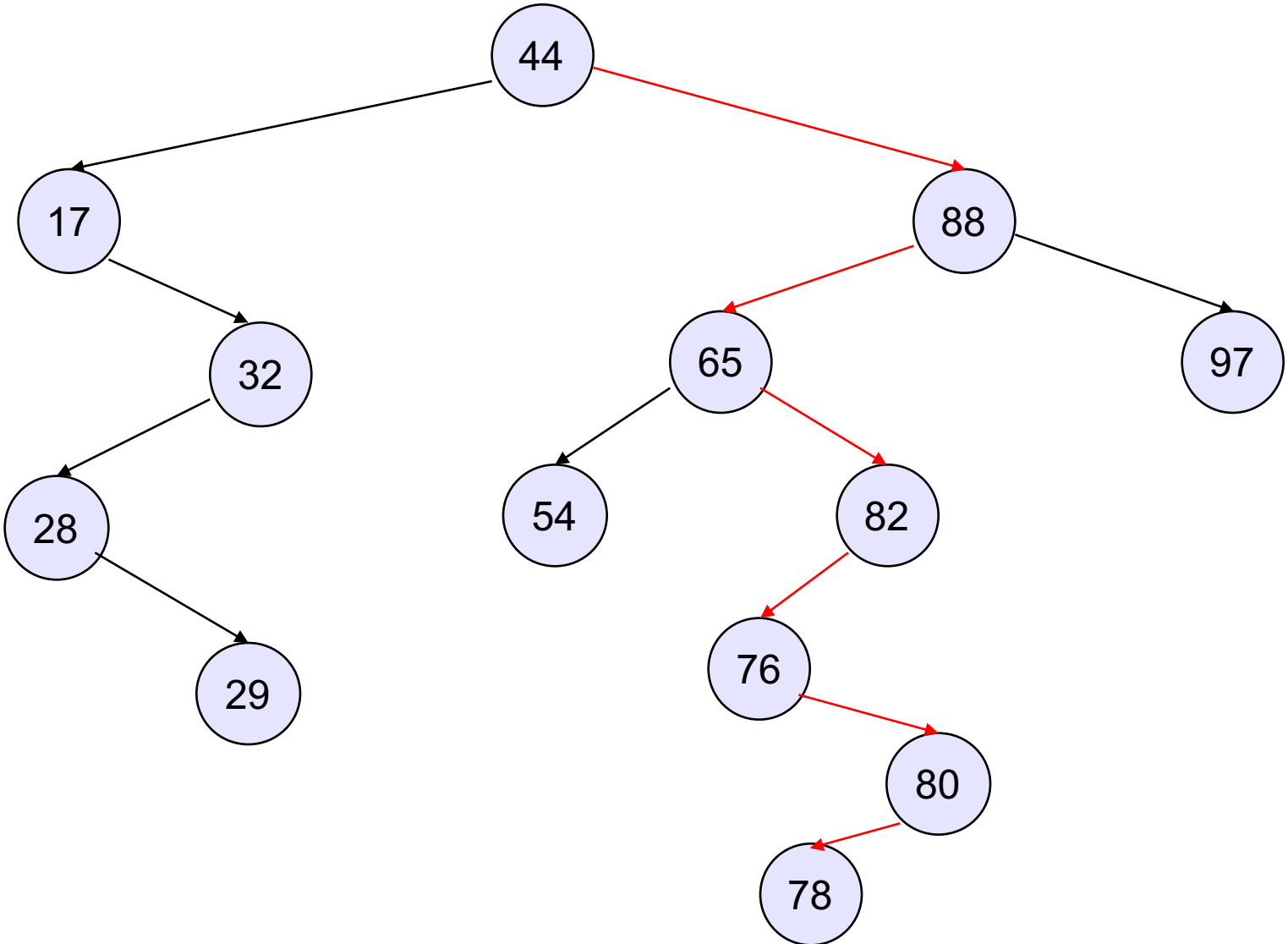
- To perform the operation insertElem(*k,e*) into a dictionary *D* implemented as a binary search tree, we call the method TreeSearch(*k,T.*root()). Suppose that *w* is the node returned by TreeSearch. Then:
  - If besides *w* a flag NO_SUCH_KEY is returned, then compare *e* with *w.* If *e*<*w*, create a new left child and insert the element *e* with the key *k*. Otherwise, create a right child and insert the element *e* with the key *k*.
  - If only the node *w* is returned (there is another item with key *k*), we call the algorithm TreeSearch(*k,T.*leftChild(*w*)) and TreeSearch(*k,T.*rightChild(*w*)) and recursively apply the algorithm returned by the node TreeSearch.

Insertion into a Binary Search Tree



Insertion of an item with the key 78 into a binary search tree

Insertion into a Binary Search Tree



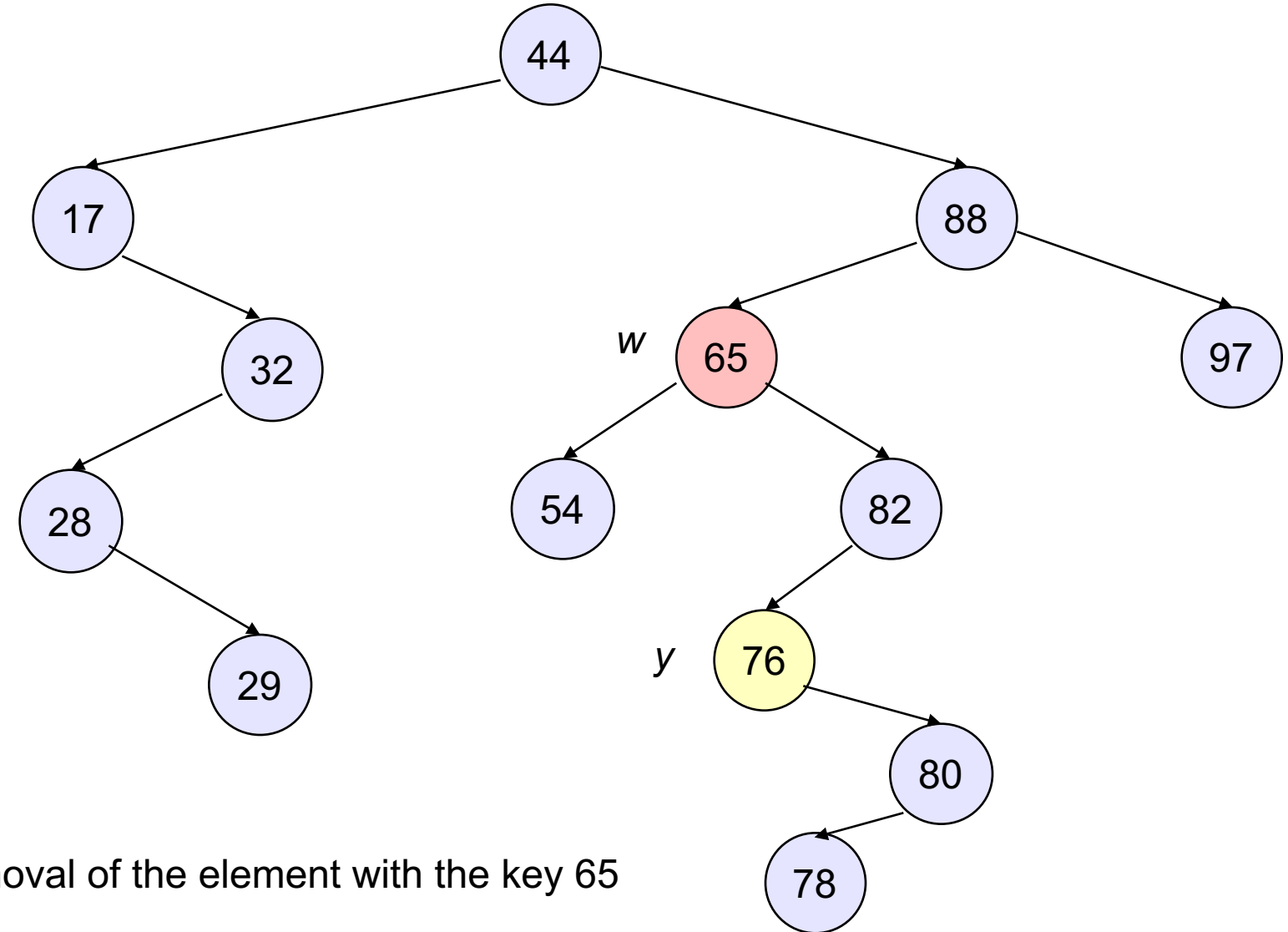
Removal from a Binary Search Tree

- Performing removeElement(*k*) on a dictionary *D* implemented with a binary search tree introduces an additional difficulty that the tree needs to remain connected after the removal.
- We need to execute first TreeSearch(*k,T.*root()) to find a node with a key *k*. If the algorithm returns an exception, there is no such element in *D.* If the key *k* is found in *D*, we distinguish two cases:
  - If the node with the key *k* is the leaf node, the removal operation is simple;
  - If the node with the key *k* is an internal node, its simple removal would create a hole. To avoid this, we need to do the following:

Removal from a Binary Search Tree

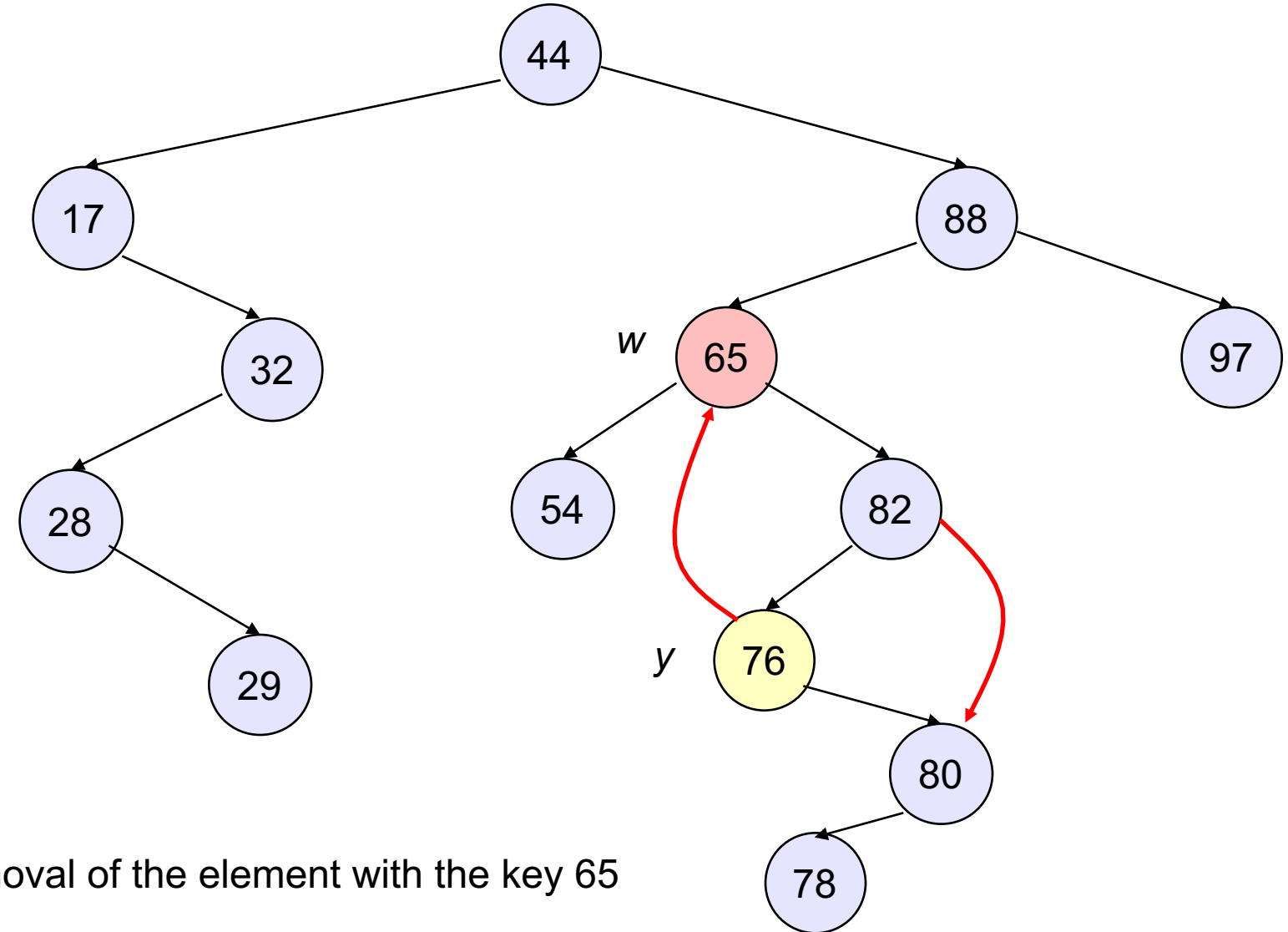
1. Find the first node *y* that follows *w* in an inorder traversal. It is the leftmost internal node in the right subtree of *w* (go right from *w*, and then follow the left children;
2. Save the element stored at *w* into a temporary variable *t*, and move *y* into *w*. This would remove the previously stored element of *w*;
3. Remove the element *y* from *T*;
4. Return the element stored in a temporary variable *t*;

Removal from a Binary Search Tree



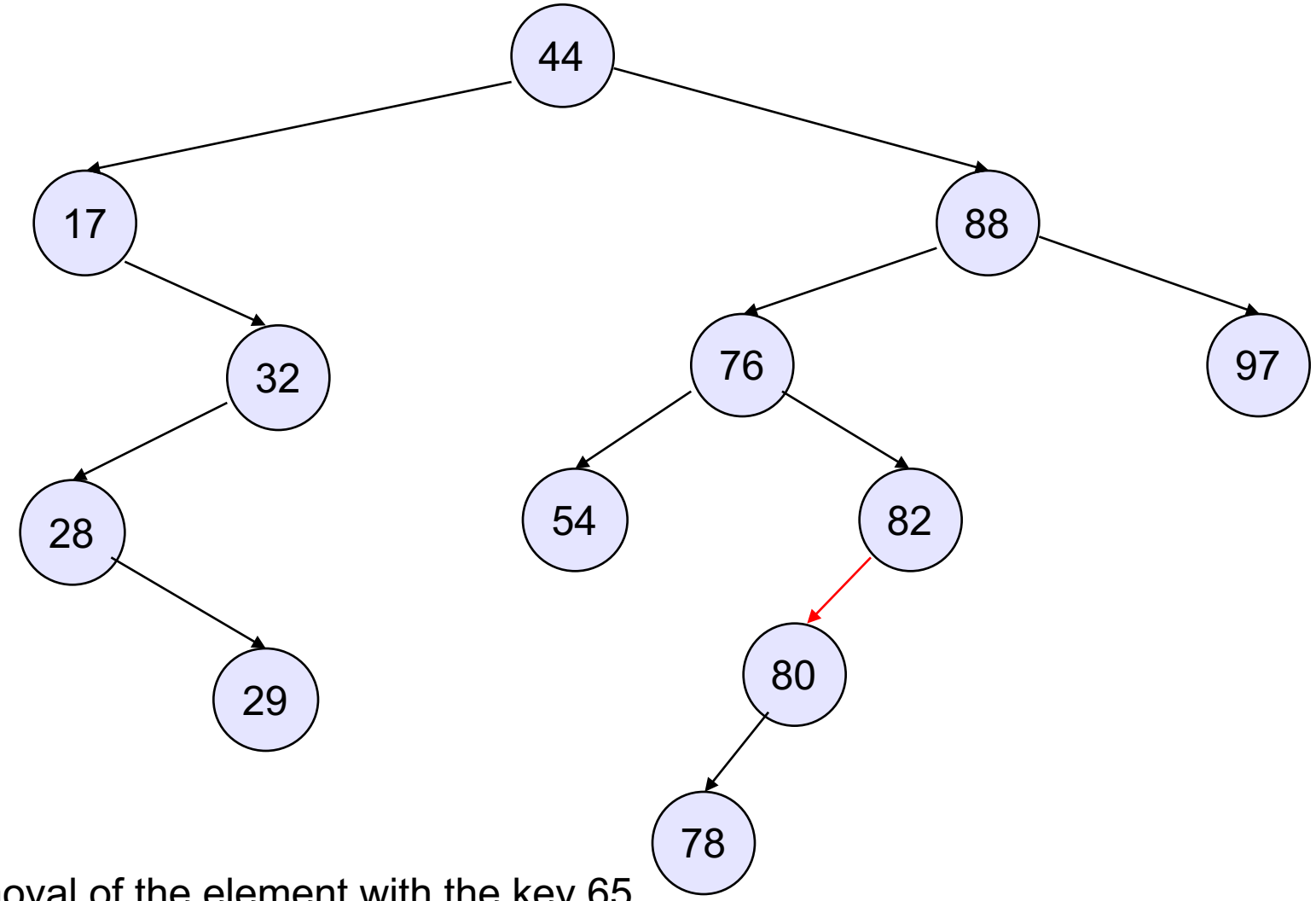
Removal of the element with the key 65

Removal from a Binary Search Tree



Removal of the element with the key 65

Removal from a Binary Search Tree



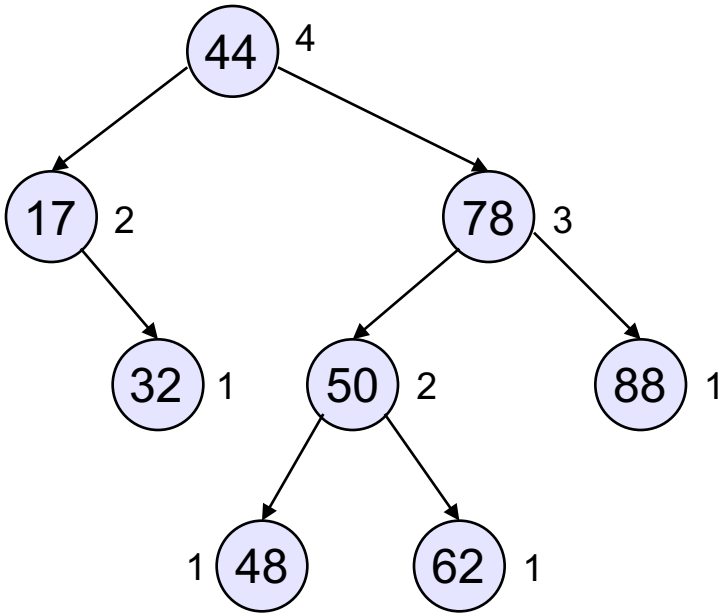
Removal of the element with the key 65

AVL Trees

- The idea behind introducing the AVL trees is to improve the efficiency of the basic operations for a dictionary.
- The main problem is that if the height of a tree that implements a dictionary is close to *n*, the basic operations execute in time that is asymptotically no better than that obtained from the dictionary implementations via log files or lookup tables.
- A simple correction is to have an additional property added to the definition of a binary search tree to keep the logarithmic height of the tree. This is the height balance property:

  For every internal node *v* of the tree *T*, the heights of its children can differ by at most 1.

AVL Trees

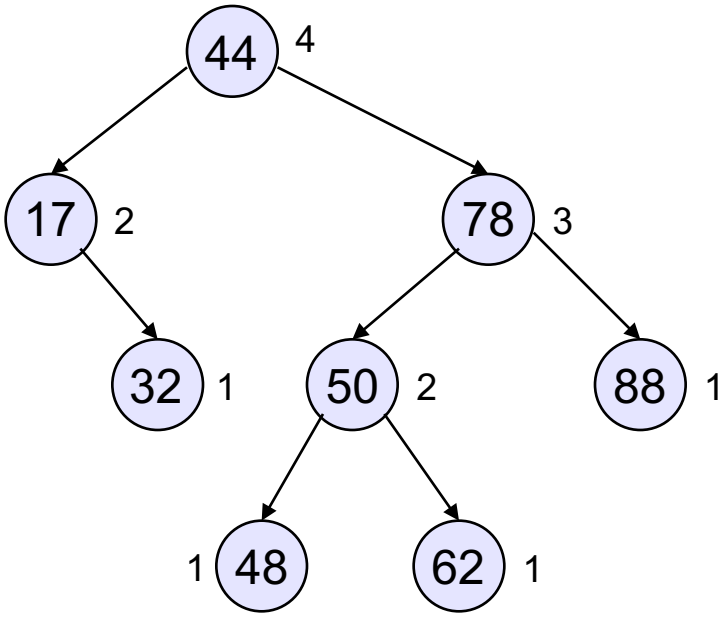


An example of an AVL tree with the node heights

- Any subtree of an AVL tree is an AVL tree itself.
- The height of an AVL tree that stores *n* items is $O(\log n)$.
- This implies that searching for an element in a dictionary implemented as an AVL tree runs in $O(\log n)$ time.

Insertion into an AVL Tree

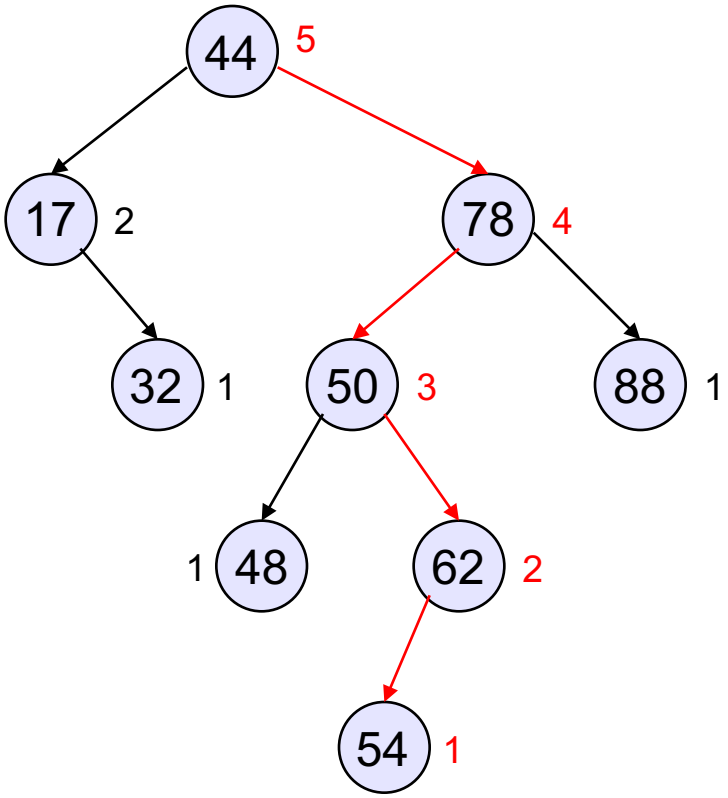
- The first phase of an element insertion into an AVL tree is the same as for any binary tree.



An example of inserting the element with the key 54 into an AVL tree

Insertion into an AVL Tree

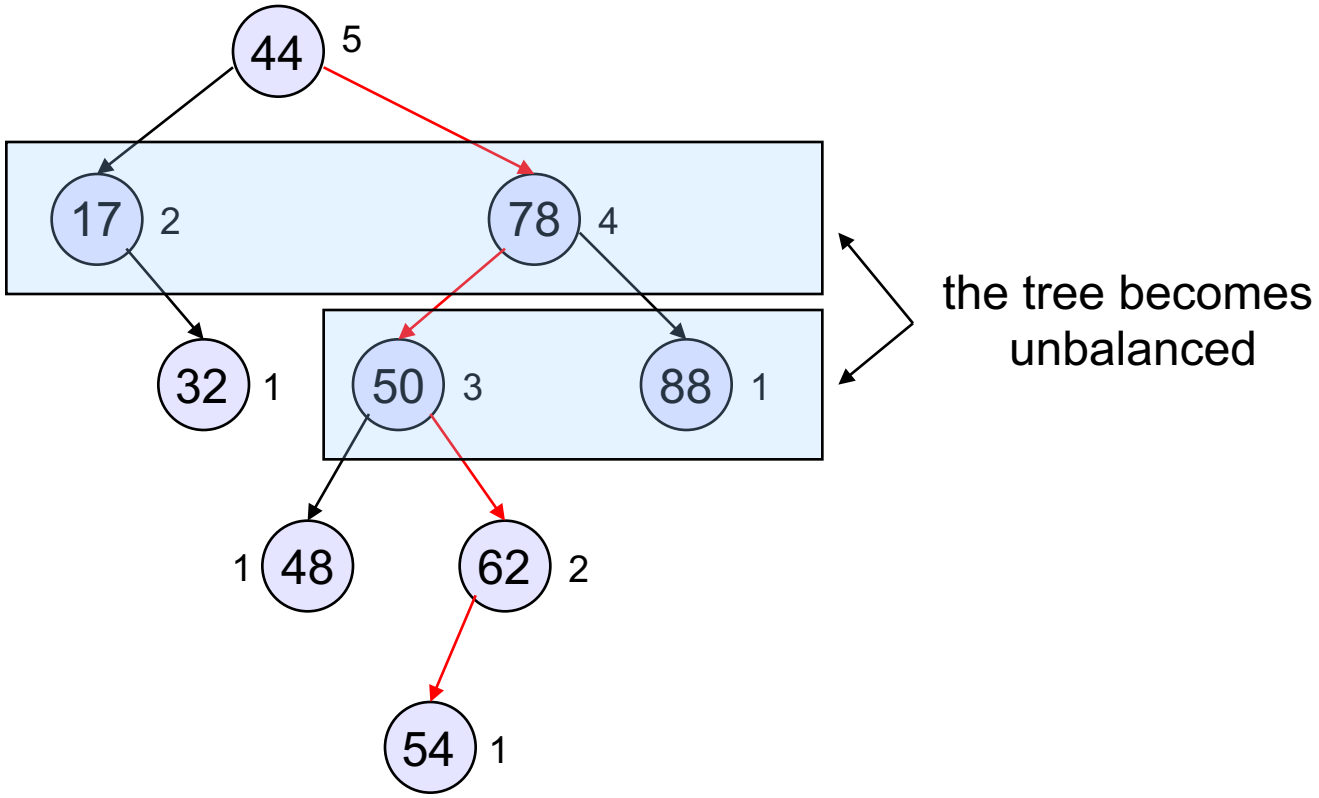
- The first phase of an element insertion into an AVL tree is the same as for any binary tree.



An example of inserting the element with the key 54 into an AVL tree

Insertion into an AVL Tree

- The first phase of an element insertion into an AVL tree is the same as for any binary tree.

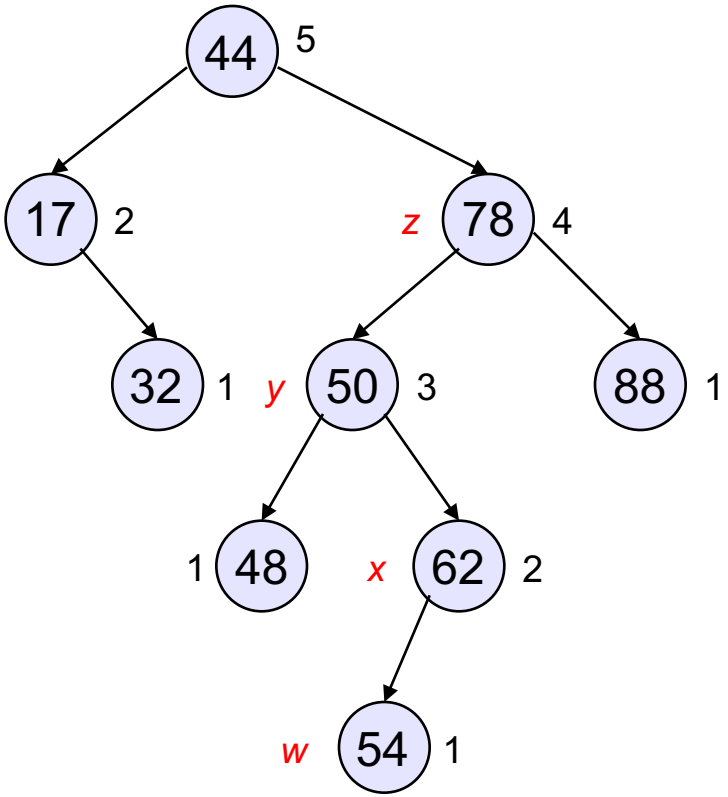


An example of inserting the element with the key 54 into an AVL tree

Insertion into an AVL Tree

- Suppose that the tree satisfies the height-balance property prior to the insertion of a new element *w*. After inserting the node *w*, the heights of all nodes that are on the path from the root to the newly inserted node will increase. Consequently, these are the only nodes that may become unbalanced by the insertion.
- We restore the balance of the nodes in the AVL tree by a search and repair strategy.
- Let *z* be the first node on the path from *w* to root that is unbalanced.
- Denote by *y* the child of *z* with a larger height (if there is a tie, choose *y* to be an ancestor of *z*).
- Denote by *x* the child of *y* with a larger height (if there is a tie, choose *x* to be an ancestor of *z*).

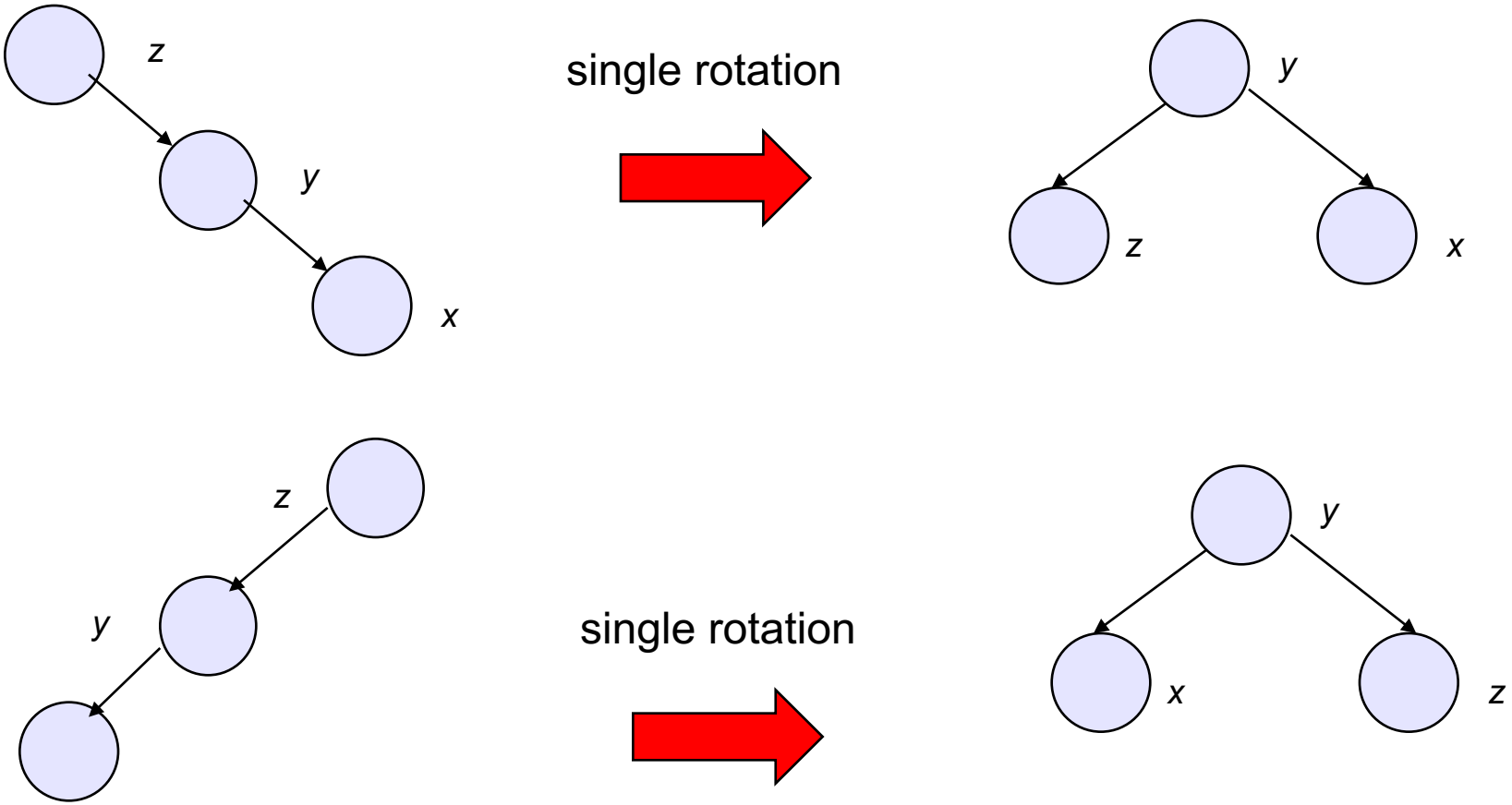
Insertion into an AVL Tree



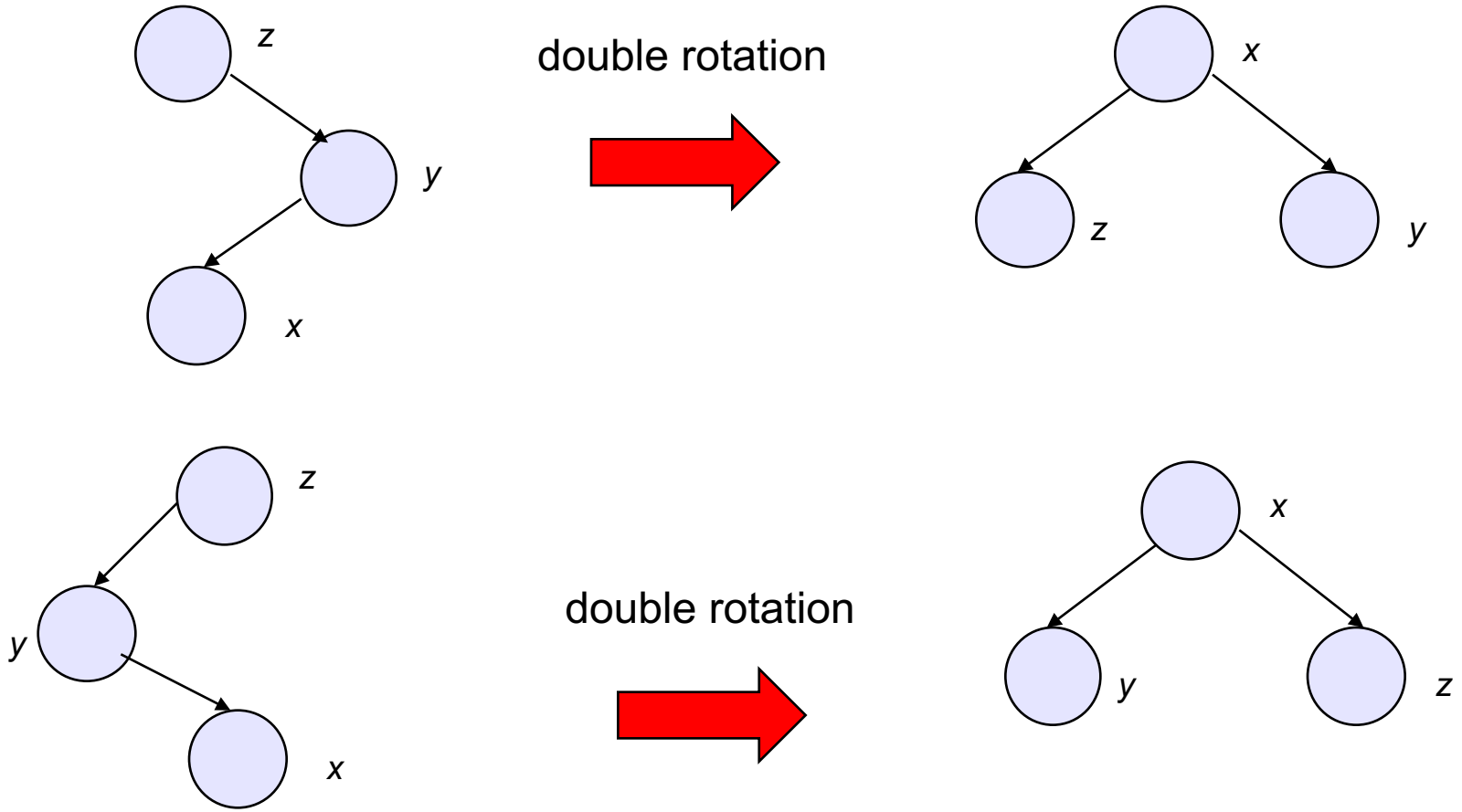
Balancing an AVL Tree

- The node *z* becomes unbalanced because of an insertion into the subtree rooted at its child *y*.
- The subtree rooted at *z* is rebalanced by the trinode restructuring method. There are 4 cases of the restructuring algorithm. The modification of a tree *T* by a trinode restructuring operation is called a rotation. The rotation can be single or double.
- The trinode restructuring methods modify parent-child relationships in O(1) time, while preserving the inorder traversal ordering of all nodes in *T*.

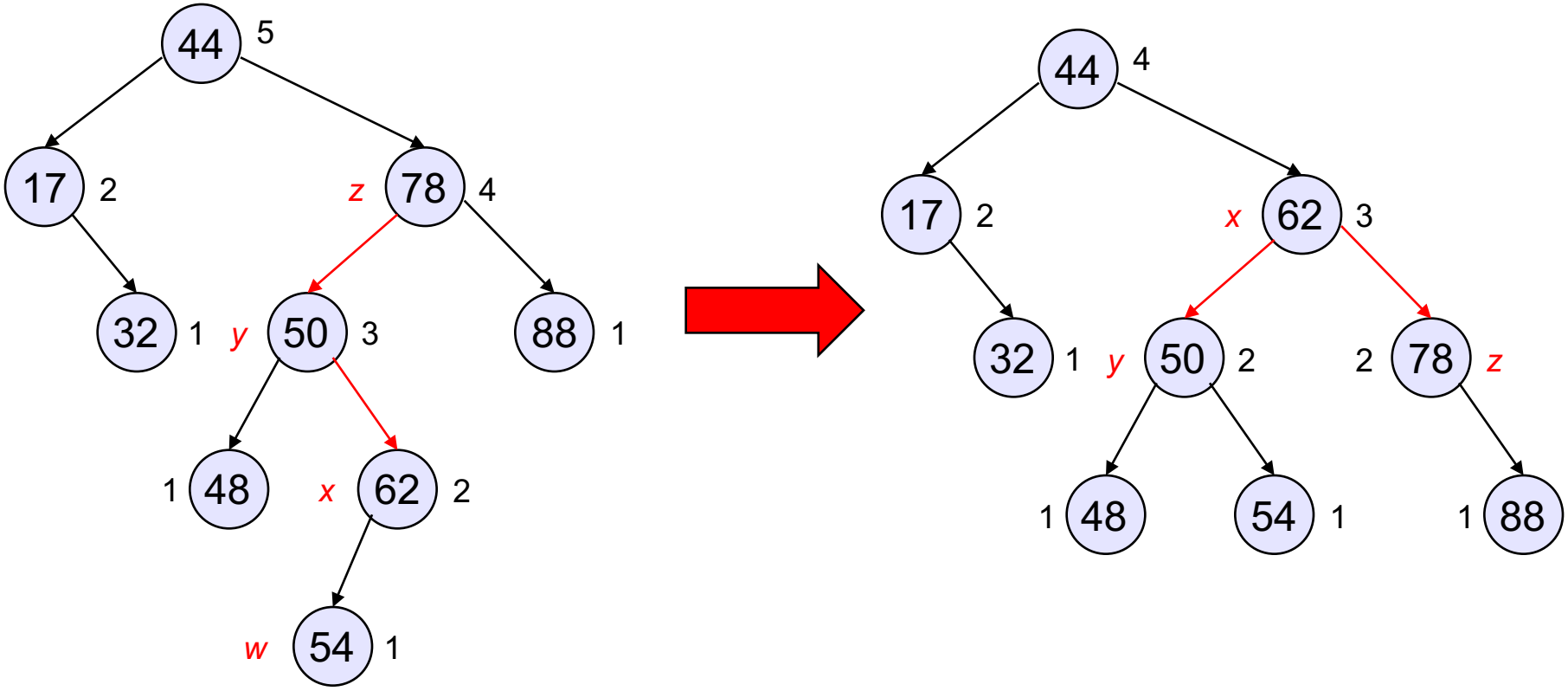
Trinode Restructuring by Single Rotation



Trinode Restructuring by Double Rotation



Balancing an AVL Tree



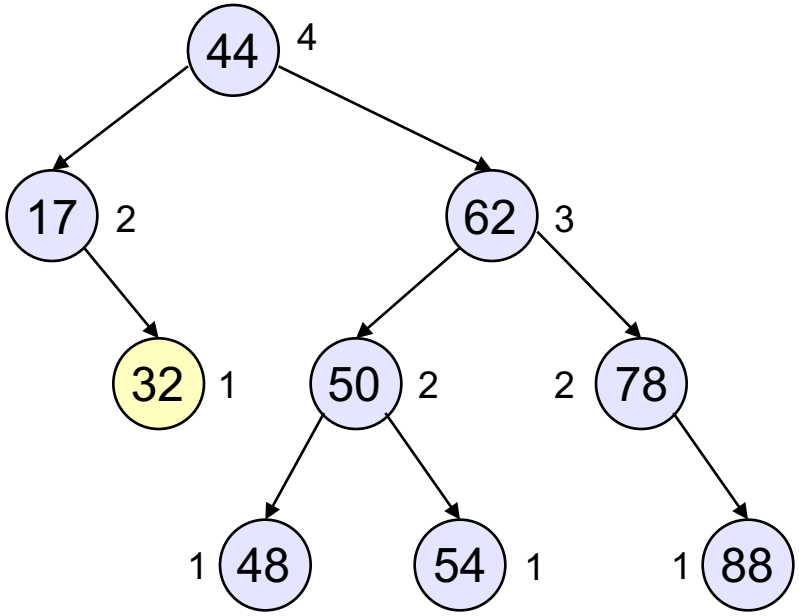
Removal from an AVL Tree

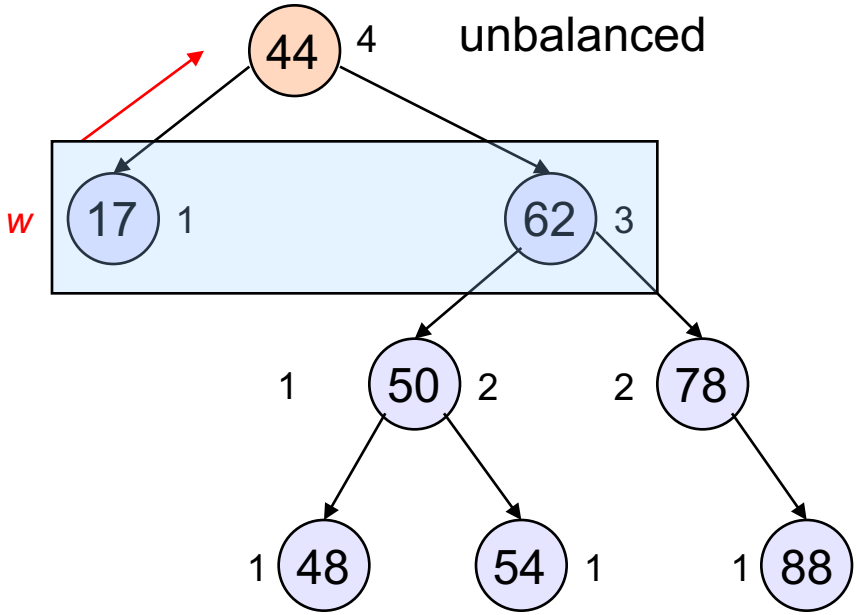
- The first phase of the element removal from an AVL tree is the same as for a regular binary search tree. This can, however, violate the height-balance property of an AVL tree.
- If we remove an external node, the height balance property will be satisfied.
- But, if we remove an internal node and elevating one of its children into its place, an unbalanced node in T may occur. This node will be on the path from the parent w of the previously removed node to the root of T.
- We use the trinode restructuring after the removal to restore the balance.

Removal from an AVL Tree

- Let *z* be the first node encountered going upwards from *w* (the parent of the removed node) towards the root of *T*.
- Let *y* be the child of *z* with a larger height (i.e. it is a child of *z*, but not an ancestor of *w*).
- Let *x* be a child of *y* with a larger height (this choice may not be unique).
- The restructuring operation is then performed locally, by restructuring a subtree rooted in *z.* This may not recover the height balance property, so we need to continue marching up the tree and looking for the nodes with no height balance property.
- The operation complexity of the restructuring is proportional to the height of a tree, hence $O(\log n)$.

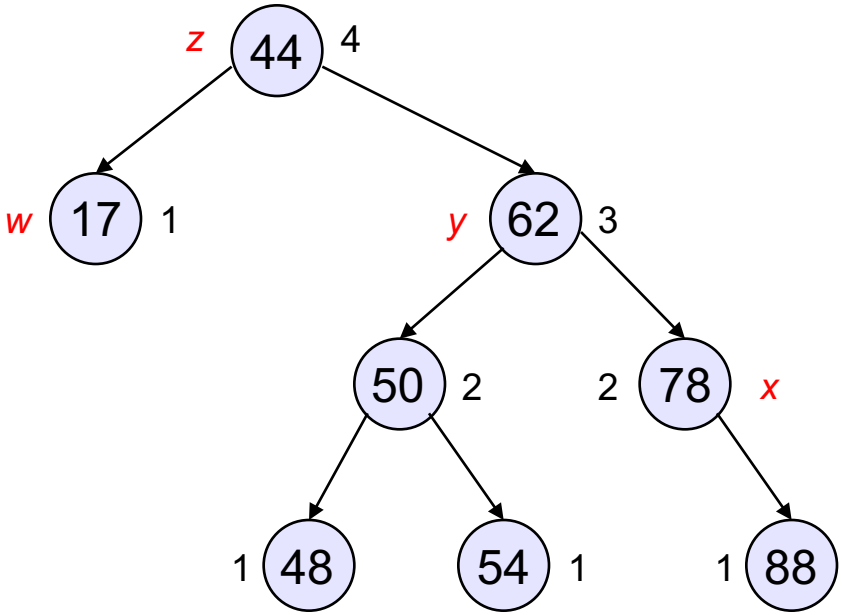
Removal from an AVL Tree

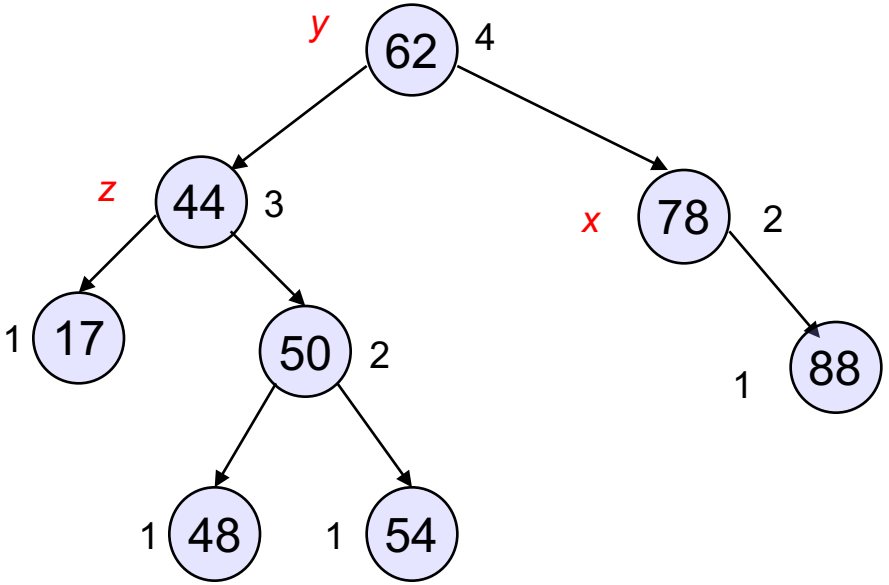




Removal of the element with key 32 from the AVL tree

Removal from an AVL Tree





An Introduction to Algorithmic Problem-solving Techniques

Question: Given a computational task, how do we devise algorithms to solve it?

Algorithmic techniques which may or may not provide solutions: a range of tools for constructing algorithms.

We look briefly at three widely-applicable techniques:

- Divide-and-Conquer
- Greedy Strategies
- Dynamic Programming

This is Chapter 5 of the course textbook.

Divide-and-Conquer - Introduction

Problem: Consider finding both the maximum and minimum of a sequence of integers.

The obvious method is to find the maximum by iterating through the sequence and then doing the same to find the minimum.

Question: What is the time complexity of this? How many integer comparisons do we make for a sequence of length N?

Answer: For the maximum it is $N - 1$ comparisons, and then for the minimum it is $N - 2$ comparisons, so the total is

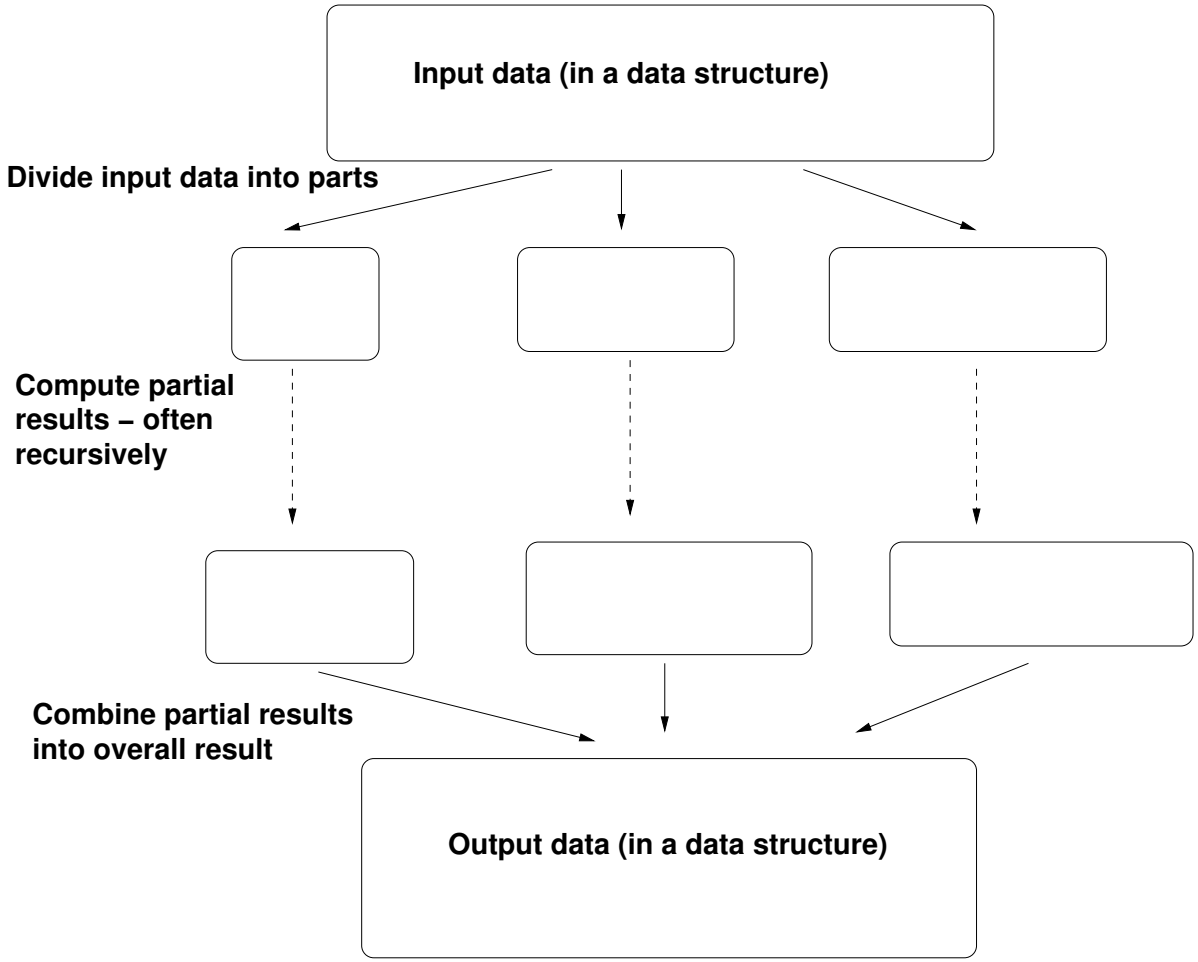
$$2N - 3$$

comparisons.

Question: Can we do better?

Divide-and-Conquer - The Idea

The scheme is: Divide the input into parts, solve the parts (often recursively), then combine the solutions to give the final result.



Divide-and-Conquer - Example

Using divide-and-conquer to find the maximum and minimum elements in a sequence of integers:

Divide the sequence s into s_1 and s_2 (arbitrarily – any elements in either subsequence, but approximately equal sizes for efficiency).

Now recursively calculate the maximum and minimum of subsequence s_1 (call them max_1 and min_1) and of subsequence s_2 (max_2 and min_2).

Can we calculate the maximum and minimum of the whole sequence s?

Yes: the maximum is $max(max_1, max_2)$ and the minimum is $min(min_1, min_2)$.

Divide-and-Conquer - Example (continued)

This gives a recursive algorithm:

```
maxmin(s) =
     if s = [x] then return (x,x);
     if s = [x1,x2]
      then
         if x1>x2 then return (x1,x2) else return (x2,x1);
      else
       (s1,s2) = divide(s);
       (max1,min1) = maxmin(s1);
       (max2,min2) = maxmin(s2);
       return (max(max1,max2),min(min1,min2))
```

Divide-and-Conquer - Example (continued)

Is this any better? How many integer comparisons are required for a sequence of length N? This is hard to say!

Let C_N be the number of integer comparisons needed on a sequence of length N. Then

$$C_N = 2 \times C_{\frac{N}{2}} + 2$$

and $C_1 = 0$ and $C_2 = 1$.
Why? – Count the operations at each stage of the algorithm.

The solution of this is

$$C_N = \frac{3N}{2} - 2.$$

This is much smaller than the previous method, which required $2N - 3$. Why are fewer comparisons needed? Notice that both methods are linear, $O(N)$.

Divide-and-Conquer - Applications

Divide-and-conquer is a technique of very wide application. A few examples:

- Efficient sorting algorithms - both Mergesort and Quicksort are divide-and-conquer algorithms and have average time complexity of $O(N \times \log(N))$, whereas most simple general sorting algorithms are much slower, $O(N^2)$.
- Fast integer multiplication: Integer multiplication by long multiplication is $O(N^2)$, but there are fast $O(n \times \log(N))$ divide-and-conquer algorithms.
- Fast matrix multiplication: Standard matrix multiplication is $O(N^3)$, divide-and-conquer algorithms produce algorithms $O(N^{2.808...})$ (and even down to $O(N^{2.376...})$).
- Nearest neighbour problems: Given a set of points in 2D or 3D (or N dimensions), find two nearest points. Divide-and-conquer algorithm is $O(N \times \log(N))$.
- Other problems in Computational Geometry: For example, the convex hull of a set of points.

Greedy Methods - An Example

Problem: Suppose we have a set of coins of various denominations (values) and we wish to pay for an item of cost V with a minimum number of coins.

How do we select the coins to do this?

Suppose (as in the UK) we have coins of values 1, 5, 10 and 20 pence, and we wish to pay for an item costing 37 pence.

We need to choose a minimum number of coins to do this. How?

Answer: Choose the biggest value coins that we can at each stage: choose a 20, then a 10, then a 5, then a 1, then a 1, and then we are done! 5 coins are needed and this is a minimum.

This is called a greedy strategy.

Greedy Methods - An Example (continued)

Does a greedy strategy always work? For these coin values, it always works. Why?

Consider coin values 1, 10 and 6 and we have item costing 12.

Then the greedy strategy fails. We choose a 10 and then two 1s, to give 3 coins. But we could have chosen 2 coins of value 6.

Greedy Methods - Optimisation Problems

An Optimisation Problem requires us not simply to solve the problem, but to produce a 'best' solution.

'Best' is in terms of some evaluation of the quality of the solution. It may mean the largest or smallest, the closest or furthest, the shortest or longest, etc. In general, we talk of maximum and minimum solutions.

A greedy strategy attempts to find a maximum (or minimum) solution, by maximising (or minimising) the choices at intermediate stages.

Greedy Methods - Applications

Applications:

- For some optimisation problems a greedy strategy is always successful.
- For some optimisation problems, a greedy strategy solves some instances but not others (as in the coins example above).
- For some optimisation problems, a greedy strategy may not give optimal solutions but may lead to good approximations to optimal solutions.
- For some optimisation problems, a greedy strategy is not applicable – no useful solutions are produced.

Greedy Methods - Applications

Some standard problems with greedy solutions:

- Many path-finding algorithms - eg shortest paths using Dijkstra's algorithm.
- Some job scheduling problems admit greedy solutions, others don't.
- Spanning trees of a graph: Given an edge-weighted graph, choose a subset of the edges which form a tree and which include all nodes . We seek such a tree with minimum combined edge-weights.
- Knapsack problems: The fractional knapsack problem admits a greedy solution, but for the 0/1 problem, greedy solutions are not necessarily optimal.

Dynamic Programming - A Simple Example

How do we solve the general case of the coin problem? Greedy solutions fail in general.

Dynamic programming always produces an optimal solution to this problem.

Suppose we have coin types $1, \ldots, N$ and the value of coin type i is v_i.

Suppose that we have enough coins of each type (if we have a limited number of coins of each type, we can modify the idea below).

How do we make a sum with a minimum number of coins?

Dynamic Programming - A Simple Example (continued)

Obvious property:

Let $c(i, s)$ be the minimum number of coins from types 1 through to i required to make sum s. Consider what happens if we add another coin type $i + 1$:

$$
\begin{aligned}
c(i+1, s) = min(\quad & c(i, s) \\
& c(i, s - v_{i+1}) + 1 \\
& \vdots \\
& c(i, s - k \times v_{i+1}) + k \ ) \quad \text{where } (k+1)v_{i+1} > s \\
c(i, s) = \infty \quad & \text{if value } s \text{ cannot be made with coins } 1 \ldots i.
\end{aligned}
$$

This says: if we have solved the problem for coins of types $1 \ldots i$ and now we consider coins of type $i + 1$, then an optimal solution may be to use no coins of type $i + 1$ or one such coin combined with an optimal solution of a smaller problem using coin types $1 \ldots i$, or two coins of type $i + 1$...

Example: there are 4 types of coins, with values 9, 1, 5 and 6. We wish to make sum 11.

Difficulty: Each subproblem that we encounter, using the above recursive relation, may be needed several times to solve the problem - we do not wish to recalculate these results. So... store the subproblem results.

This is typical of dynamic programming - we construct an array of solutions to subproblems:

sum =	1	2	3	4	5	6	7	8	9	10	11
coin type 1	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
coin types 1,2	1	2	3	4	5	6	7	8	1	2	3
coin types 1,2,3	1	2	3	4	1	2	3	4	1	2	3
coin types 1,2,3,4	1	2	3	4	1	1	2	3	1	2	2

The answer, using all 4 coin types, is that 2 coins are needed to make a sum of 11.

Dynamic programming is a bottom-up method – we solve all smaller problems first then combine them to solve the given problem.

How efficient is this dynamic programming solution? What is its time complexity?

The main factor is the size of the table of subproblem results (each entry may require a different number of operations, but this is not usually significant). Thus for N coin types, and a value required of V, the table size is $N \times V$ (notice that this is in terms if N and V, where it is natural to ask for the dependence on V only).

Notice that some subproblem results are not required for the final solution, so that some entries need not be computed - but this is not easy to use as a reduction strategy: it is difficlut to predict what might be needed.

Dynamic Programming - Applications

Many optimisation problems admit a dynamic programming solution. A few examples:

- Some path-finding algorithms use dynamic programming, for example Floyd's algorithm for the all-nodes shortest path problem.
- Some text similarity tests: For example, longest common subsequence.
- Knapsack problems: The 0/1 Knapsack problem can be solved using dynamic programming.
- Constructing optimal search trees.
- The travelling salesperson problem has a dynamic programming solution.

Algorithmic problem-solving:
Tractable vs Intractable problems

Algorithmic techniques

Question: Given a computational task, what sort of algorithms are available for it? What time complexities can we expect?

We have already looked at a few problem solving techniques:

1. Divide-and-Conquer
2. Greedy Strategies
3. Dynamic Programming

These are some of the important techniques, but there are others.

Techniques (1) and (2) tend to give fast (polynomial-time) algorithms. (3) can give fast algorithms, but can also be used when no fast algorithms are available.

Polynomial-time vs exponential algorithms

An algorithm is said to be polynomial-time if its worst-case time complexity is amongst:

$$O(1), O(N), O(N^2), O(N^3), \ldots$$

Whilst $O(1)$ is (very!) fast, $O(N)$ is fast, $O(N^{100})$ is slow, but we still consider these as fast algorithms and as tractable solutions to a problem.

Exponential algorithms: Algorithms that behave in the worst-case as 2^N or N^N or $N!$ (N factorial) are very slow and are considered as intractable algorithms in general.

Note: Stirling's formula (with $e = 2.71828\ldots$ the base of natural logarithms):

$$N! \approx \sqrt{2\pi N}(N/e)^N$$

Thus factorial is not only exponential, but badly so!

Why are exponential algorithms intractable?

Exponential algorithms run very slowly as the size of the input N increases and on fairly small inputs require impossible computational resources:

N	2^N	$N!$	N^N
10	10^3	4×10^6	10^{10}
20	10^6	2×10^{18}	10^{26}
100	10^{30}	9×10^{157}	10^{200}
$1,000$	10^{301}	10^{2672}	10^{3000}
$10,000$	10^{3000}	10^{36701}	10^{40000}

The age of the universe is approximately 4×10^{26} nanoseconds.

It is thought that there are approximately 10^{80} protons in the universe (the Eddington number).

What algorithms are available?

Finding how difficult a computational task is, is difficult!

Problems which appear very similar, or even variants of each other, can have very widely different time complexities.

Remember the coins problem? Some collections of coin values have fast greedy algorithms, other collections of values don't have any fast algorithms.

Where do exponential algorithms come from?

Exponential algorithms arise from various algorithmic techniques:

- Exhaustive enumeration - listing all possibilities and checking each for the solution requirement, or
- Systematic search of the space of (partial) solutions using unbounded backtracking.

An algorithmic task: Graph colouring

A graph consists of a set of nodes linked by (undirected) edges.

A colouring of a graph with k colours is an allocation of the colours to the nodes of the graph, such that each node has just one colour and nodes linked by an edge have different colours.

The minimum number of colours required to colour a graph is its chromatic number.

Applications: Graphs are usually used where 'connectedness' is being recorded. Colouring is about the opposite: edges are present when nodes need to be separated.

An exponential-time colouring algorithm - by enumeration

Exhaustive enumeration is the simplest approach to colouring a graph with k colours:

1. Enumerate all allocations of k colours to the nodes of the graph,
2. Check each allocation to see if it is a valid colouring.

For a graph with N nodes and E edges, there are k^N allocations of k colours to the nodes.

Checking each allocation to see if it is a valid colouring, is $O(E)$.

So this algorithm has worst-case time complexity $O(E \times k^N)$.

A graph colouring algorithm by systematic search

Here is an algorithm for k-colourability using exhaustive unbounded back-tracking:

Number the nodes $1, \ldots, N$. For each node encountered, it successively tries the colours. If all fail, it undoes the colour of the previous node and tries new colours for it. If it succeeds, it then tries to colour the next node.

```
n := 1;
while n =< N do
  { attempt to colour node n with
    next colour not tried for n }
  if there is no such colour
    then
       if n>1 then n := n-1 else fail
    else if n=N then print colouring else n := n+1
```

Time complexity: This is exponential in worst-case. Why?

Exponential algorithms: continued

In fact, all known algorithms for graph colouring (for $k \geq 3$) are exponential!

Graph colouring is an example of what is known as an NP-complete problem. The only algorithms known for NP-complete problems are exponential.

Whether NP-complete problems admit polynomial-time algorithms is one of the most important open problems in Computer Science.

There is one million US dollars available to anyone who can come up with a polynomial algorithm for graph colouring!

More on this topic (and NP-completeness) will be found in the Advanced Algorithms course in the third year.

Intractable problems: Examples

Many common computational problems are NP-complete and therefore all known algorithms are exponential:

For example, the celebrated...

Travelling salesperson problem: Find a tour of N cities in a country (assuming all cities are reachable). The tour should (a) visit every city just once, (b) return to the starting point and (c) be of minimum distance.

This is an optimisation problem - not just find a solution but find 'the best' solution.

Travelling salesperson: Solutions

Solution by exhaustive enumeration:

1. Starting at any city, enumerate all possible permutations of cities to visit,
2. Find the total distance of each permutation and choose one of minimum distance.

There are $(N-1)!$ permutations for N cities (the starting city is arbitrary), so this is an exponential solution.

There are many other approaches to the travelling salesperson problem, for example using dynamic programming. All are exponential in worst-case (but there are faster approximate algorithms).

Knapsack problems

A Knapsack problem requires us to fit a number of items of different sizes into a container ('knapsack'). We are asked to do this optimally: i.e. fill the container as much as possible (or, equivalently, minimise the unused space).

More generally, the items have values as well as size/weight and we are asked to maximise the total value in the knapsack.

This is the next laboratory exercise.

Solutions: Some Knapsack problems have polynomial (indeed linear) solutions, others have only exponential solutions.

Job scheduling, rostering and timetabling

General problem: A collection of tasks are to be undertaken by allocating them to slots. Slots may be time periods, machines, rooms, people's availability etc. or combinations of these.

Usually, the allocation is subject to constraints:

1. Compatibility constraints: which tasks fit which slots.
2. Scheduling constraints e.g. temporal orders: some task may have to be performed before others, some may or may not be allowed to be performed simultaneously, etc.
3. Resource constraints: are resources unlimited or limited?
4. Optimisation requirements: We may require solutions that maximise or minimise given measures. For example, we may require that all tasks are completed in a minimum time, or that the allocation distributes tasks as evenly as possible.

The problem is to allocate tasks to slots satisfying the constraints.

Scheduling: Examples

(1) Scheduling algorithms in multi-tasking operating systems by which threads, processes or data flows are given access to system resources. Constraints include effective load balancing and achieving a required quality of service.

(2) Industrial processing: Combining jobs and their scheduling to achieve required outputs.

(3) Rostering: For example, constructing a roster of buses and drivers to provide a bus service: Your Software Engineering laboratory exercise.

(4) Timetabling: Allocating students, staff, classes, rooms and times to provide a timetable.

Algorithms for scheduling

The wide variety of such tasks is matched by the variety of algorithmic solutions.

- Some have linear or polynomial solutions, often using simple allocation methods: For example, first-come-first-served allocations, or greedy methods.
  The simple form of your SE laboratory exercise has such a solution.
- However, most scheduling problems have only exponential solutions in general.
  The general case of timetabling is such a problem: exhaustive enumeration is one approach.
  Graph colouring is a method of solving some job scheduling problems: Suppose the nodes are tasks. Two jobs are linked by an edge if they conflict i.e. cannot occupy the same slot. A colouring with k colours schedules the jobs into k slots without conflict... and this problem has only exponential algorithms.

Heuristics

Heuristics: Heuristics are rules that can be used in decision making (for example, in exploring a space for solutions). They are intended to reach optimal solutions, but are not guaranteed to do so.

Heuristics are useful when exhaustive searches are exponential-time, and may reduce the search to polynomial-time, but achieve only approximate solutions in general.

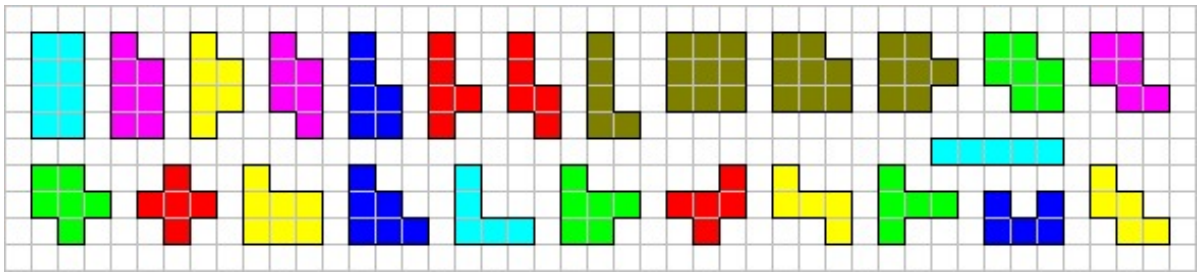
Puzzles

Most puzzles have only exponential algorithms to solve them.

That is what makes puzzles interesting!

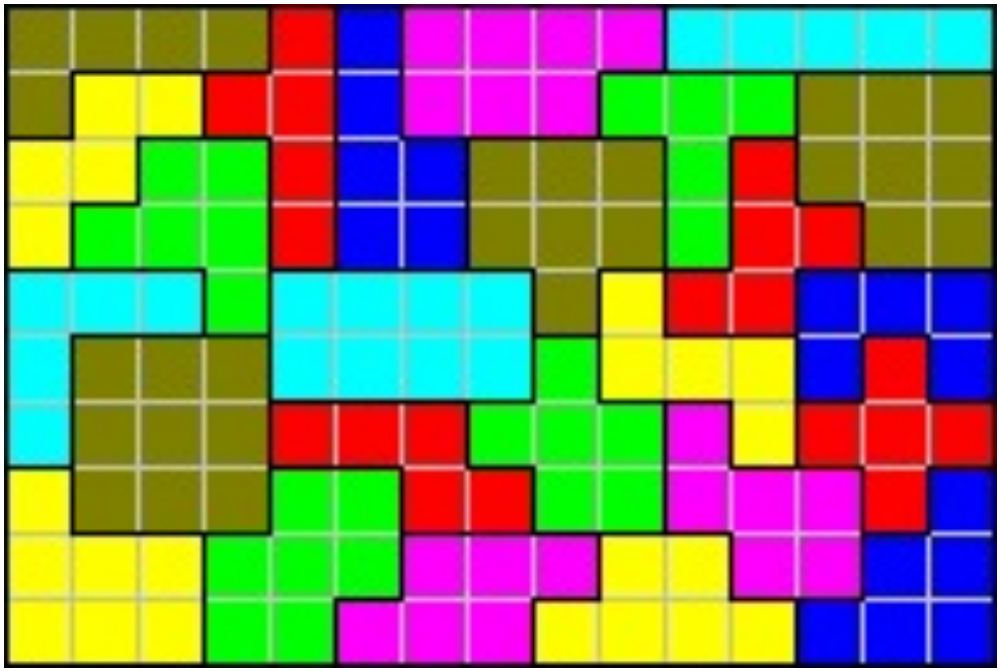
In general, the only solutions are by exhaustive search based on unbounded backtracking.

Example: Polyominoes



Problem: to fit these into a 10×15 rectangle.

One solution



[Website: http://www.iread.it/lz/polyedges.html]

Graphs and Graph Algorithms

In this part of the course unit, we shall

- learn about one of the most widely applicable structures in computing: graphs,
- consider how to use them in applications,
- understand how to represent graphs in programming languages, and
- consider some of the wide variety of algorithms that are available for computing with graphs.

This is an introduction to the material in Part II (Chapters 6, 7 and 8) of the course textbook .

Graphs

Graphs consist of a set of nodes and a set of edges which link some of the nodes.

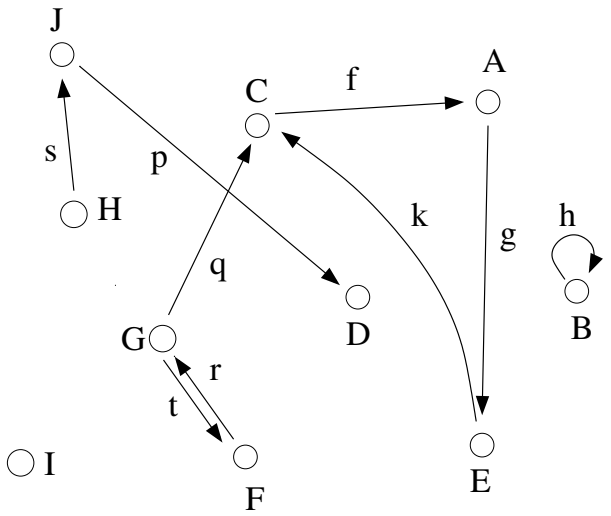


Figure : A directed graph (digraph).

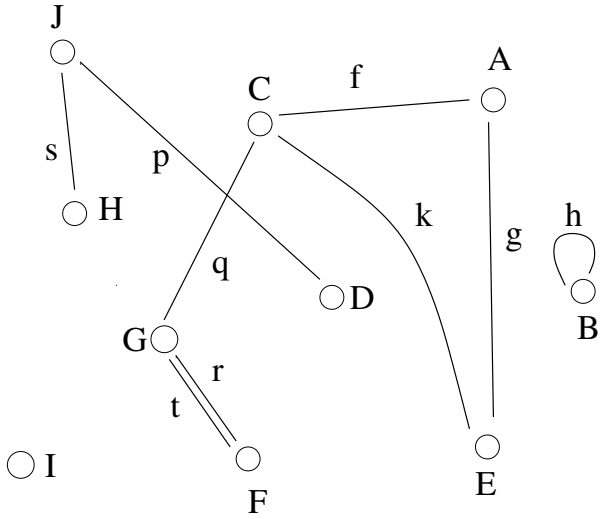


Figure : An undirected graph (a multigraph - more than one edge possible between pairs of nodes).

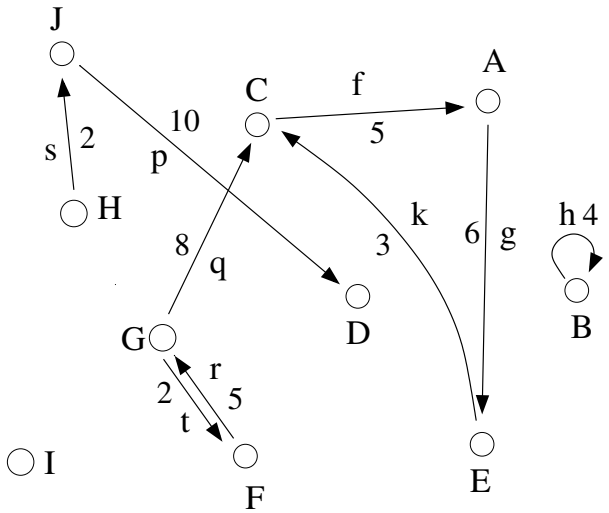


Figure : An edge-weighted directed graph.

Applications

Application	Nodes	Edges	Type of graph
Wired Networks	Computers	Cables	Undirected multigraph
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....
.....	.....	.....	.....

Terminology

Nodes - sometimes called vertices, points, etc.
Edges - sometimes called arcs, lines, etc.

Two nodes linked by an edge are adjacent or, simply linked.

For directed graphs, an edge e from node A to node B, is said to have A as its source node (or start or origin node) and B as its target node (or end or finish or destination node).

An edge is incident on a node if it has the node as source or target (directed) or if it links the node to another (undirected).

For an undirected graphs, the number of edges incident at a node is the degree of the node. For directed graphs, the number of edges whose source is a node, is the out-degree of the node, likewise targets and in-degree .

More terminology: Paths

A path is a sequence (possibly empty) of adjacent nodes and the linking edges, in the case of undirected graphs.

In the case of directed graphs, a path is a sequence (possibly empty) of adjacent nodes and linking edges, but with all edges in the same direction.

If there is a path from node A to node B, we say B is reachable from A.

A path from a node to itself is a cycle.

A loop is an edge from a node to itself.

More terminology: Connected graphs

Two nodes in an undirected graph are connected if there is a path between them.

For directed graphs, the notion of 'connected' is different (warning: the textbook doesn't make this clear):

Two nodes in a directed graph are connected if there is a sequence of adjacent nodes between them. (Notice that the edges can be in any direction between nodes in the sequence.)

A graph is connected if all pairs of nodes are connected.

More terminology: Components

A subgraph of a graph G, is a graph whose nodes and edges are a subset of those of G and whose edges have the same source and target as those of G.

A connected component (sometimes, just a 'component') of a graph G, is a largest connected subgraph (i.e. one that cannot be expanded with additional nodes without becoming disconnected).

Every graph can be partitioned into disjoint connected components.

Question: How do we compute the components of a graph efficiently? Is there a quadratic algorithm $O(N^2)$ or even a linear algorithm $O(N)$, where N is the number of nodes - what about the number of edges?

More about graphs

More about graphs:

- representing graphs in programming languages,
- traversal techniques for trees and graphs.

This is material from Chapter 6 of the course textbook.

Representing graphs

How do we represent graphs using the data types commonly provided in programming languages: arrays, linked lists, vectors, 2-D arrays etc?

There are several representations in widespread use. We deal with the two commonest.

Which to choose depends on

- properties of the graph (e.g. 'sparseness'),
- the algorithms we wish to implement,
- the programming language and how data structures are implemented, and
- application of the code.

Representing graphs: Adjacency lists

An adjacency list representation of a graph consists of a list of all nodes, and with each node n a list of all adjacent nodes (for directed graphs, these are the nodes that are the target of edges with source n).

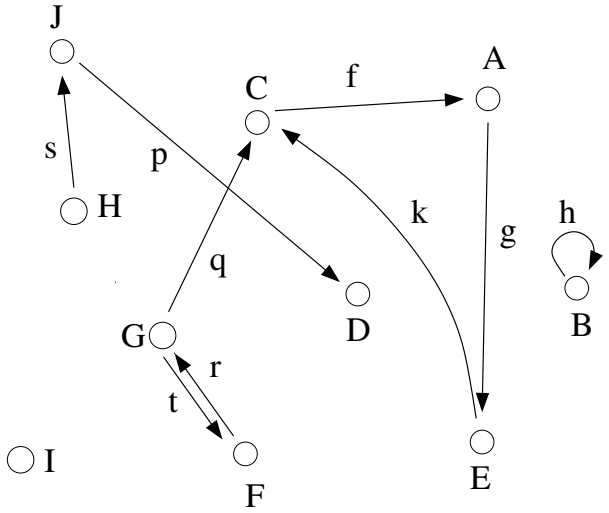


Figure : A directed graph.

Node	Adjacent nodes
A	E
B	B
C	A
D	
E	C
F	G
G	F, C
H	J
I	
J	D

Figure : Adjacency list representation of above graph

Representing graphs: Adjacency matrices

An adjacency matrix representation of a graph consists of a 2-dimensional array (or matrix), each dimension indexed by the nodes of the graph. The entries in the matrix are:

- 1 at index (m, n) if there is an edge from m to n,
- 0 at index (m, n) if there is no edge from m to n.

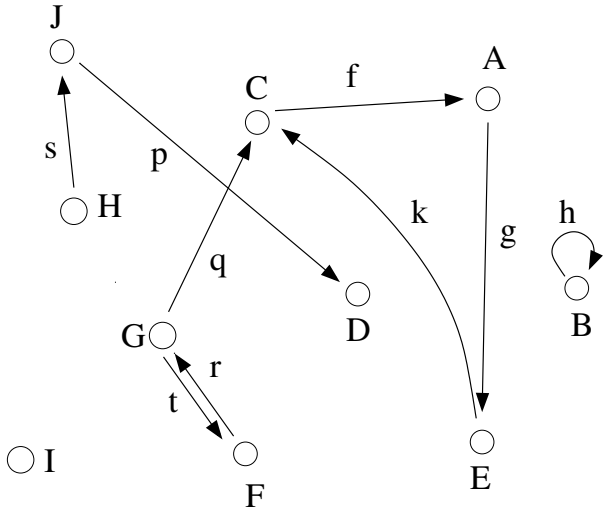


Figure : A directed graph.

	A	B	C	D	E	F	G	H	I	J
A	0	0	0	0	1	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0
E	0	0	1	0	0	0	0	0	0	0
F	0	0	0	0	0	0	1	0	0	0
G	0	0	1	0	0	1	0	0	0	0
H	0	0	0	0	0	0	0	0	0	1
I	0	0	0	0	0	0	0	0	0	0
J	0	0	0	1	0	0	0	0	0	0

Figure : Adjacency matrix representation of above graph (source nodes on the left)

Representing graphs: Notes

- These are both tabular representations. The exact choice of types to represent them depends on the application.
  For example, an adjacency list may be an array of linked lists, if we wish to have fast (random) access to the lists of adjacent nodes, but to iterate through these lists.
- Notice how sparse the adjacency matrix is: most entries are zero as there are few edges in the graph.
- The representations need modification to include various data. For example, for a multigraph, we may record the edges as well as the adjacent nodes in a list representation, or the number of edges in a matrix representation. Also for edge-weighted graphs, we include the weights in the representation.
- Notice that for undirected graphs, the matrix is symmetrical.

- Adjacency matrices allow us to do arithmetic! We may add and multiply matrices, take determinants etc. These determine transformations of graphs and values derived from graphs.
- Different representations for different tasks: We choose which representation to use by considering the efficiency of an algorithm for the task.
  Example: Consider the task of finding whether there is a path of length 2 from a given node S to another node T. For adjacency lists, we consider the list of nodes adjacent to S - in the worst case this is of length E (the number of edges of the graph). For each node in this list we see whether T is in its list. Thus the worst-case time complexity is $E \times E$. Using adjacency matrices, this complexity is N (the number of nodes) - why?
  In general, for path finding algorithms, adjacency lists are sometimes more efficient, adjacency matrices for other algorithms.

Traversals: Trees

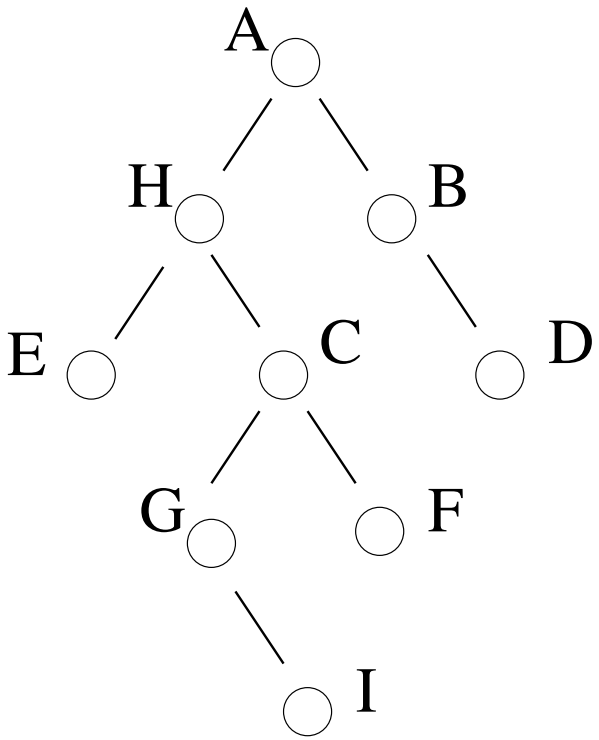
A traversal of a tree or graph is a means of visiting the nodes using the edges and revisits of nodes. Usually, there are rules to determine possible next nodes to visit or revisit.

There are many techniques, the most widely used are Depth-First Search and Breadth-First Search.

For trees, we start at the root and:

- For Depth-First Search (DFS), visit all descendants of a node, before visiting sibling nodes;
- For Breadth-First Search (BFS), visit all children of a node, then all grandchildren, etc;

Traversals of trees: Examples



A Depth-First Search from the root A (in terms of order of visiting nodes): A,H,C,F,G,I,E,B,D.

Another DFS (left-to-right): A,H,E,C,G,I,F,B,D.

For DFS, the revisiting of nodes takes place through backtracking.

A Breadth-First Search (right-to-left): A,B,H,D,C,E,F,G,I.

Priority search

Now let trees have a numerical priority assigned to each node.

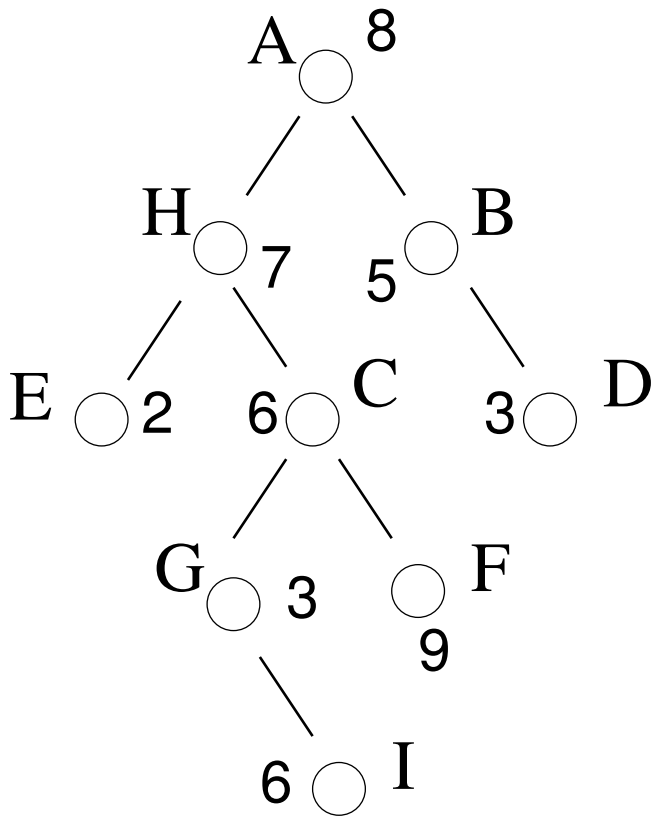
A priority search is:

1. Visit the root.
2. At each step, visit a node that has highest priority amongst unvisited children of visited nodes.

Priority searches are often used to implement heuristic search methods where the priority is calculated at each node to provide an indication of whether a route through this node is likely to reach a required goal.

Priority search: Example

As an example consider the tree:



One priority search of this tree is: A, H, C, F, B, G, I, D, E.

Generic search routine for trees

We now show how we can code all the above search techniques (traversals) of tree with the same program.

The program uses an auxiliary date structure with operations push, pop, top, empty.

- When the data structure is a stack the traversal is DFS,
- When the data structure is a queue the traversal is BFS,
- When the data structure is a priority queue the traversal is a priority search.

A priority queue has the operations of a queue (push, pop, top, empty), but top returns an item of highest priority in the queue and pop removes this item.

Generic search routine for trees - program

Idea:

1. Start by pushing root node of the tree onto the structure.
2. Then visit the top element on the structure, pop it and push its children onto the structure.

The structure therefore stores the unvisited children of visited nodes in the order in which they are encountered.

Generic search routine for trees - pseudocode

The pseudocode labels each node u with search-num(u) giving the order the nodes are encountered:

```
u <- rootnode;
s <- empty;
i <-  0;
push(u,s);
while s not empty do
 { i <-  i+1;
   u <- top(s);
   search-num(u) <- i;
   pop(s);
   forall v children of u
     push(v,s) }
```

Exercise: Hand run examples above.

Traversals: From trees to graphs

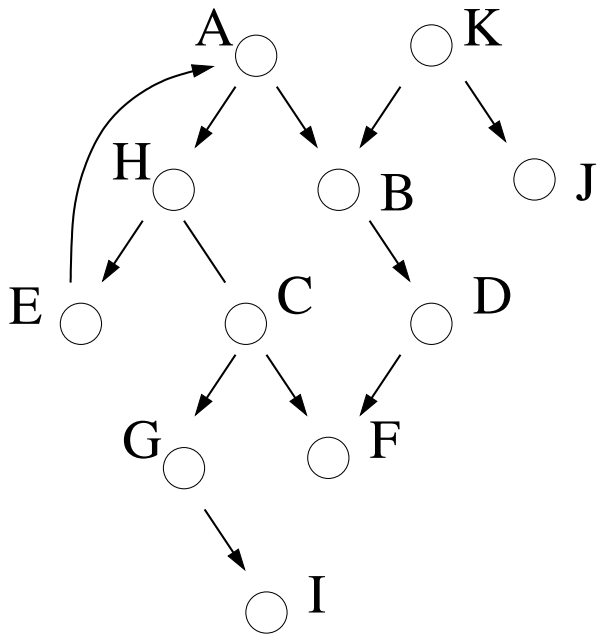
A tree is a directed graph such that:

> *There is a distinguished node (the root) such that there is a unique path from the root to any node in the graph.*

To modify tree traversal for graphs:

1. We may revisit nodes: so mark nodes as visited/unvisited and only continue traversal from unvisited nodes.
2. There may not be one node from which all others are reachable: so choose node, perform traversal, then start traversal again from any unvisited nodes.

Traversals of Graphs: Examples



A Depth-First Search: A,H,E,C,F,G,I,B,D,K,J.

A Breadth-First Search: A,H,B,E,C,D,G,F,I,K,J.

Depth-First Search for Graphs: Recursive Algorithm

Allocates a number dfsnum(u) to each node u in a graph, giving the order of encountering nodes in a depth-first search on a graph. Notice how we use this numbering to determine whether or not we have visited a node before (Zero = No, Non-zero = Yes).

```
forall nodes u do dfsnum(u) <- 0 end;
i <- 0;

visit(u) =
   { i <- i+1;
     dfsnum(u) <- i;
     forall nodes v adjacent to u do
       if dfsnum(v) = 0 then visit(v) end };

forall nodes u do
   if dfsnum(u) = 0 then visit(u) end;
```

The complexity of DFS

For a graph with N nodes and E edges:

- For the adjacency list representation, the complexity is linear $O(N + E)$,
- For the adjacency matrix representation, the complexity is quadratic $O(N^2)$.

Why?

Traversals of graphs: Notes

- The actual order determined by a depth-first search is that of a stack-based discipline, that is when we backtrack we visit the 'most recent' unvisited branch first, before backtracking to more remote ancestors.
- Breadth-First Search does not have such a recursive formulation.
- The generic code for trees above extends to graphs to provide DFS, BFS and priority search for graphs using the same auxiliary data structures.

Survey of graph algorithms

Many graph algorithms are based on traversals.

There are efficient (usually linear) algorithms based on Depth-First Search for:

- finding the connected components of a graph (how? - consider the case of undirected graphs),
- detecting cycles in a graph,
- finding 'strong components' of a graph,
- planarity testing and embedding (see below),
- articulation points and blocks of undirected graphs, 'orientability' and 'reducibility' of graphs, etc.

Survey of graph algorithms: Path finding

There are numerous path-finding problems in graphs and a variety of algorithms:

- To find all paths between all pairs of nodes ('transitive closure'), or
- To find all paths between a fixed pair of nodes, or
- To find all paths from one node to all others (the 'single source problem').

When the edges are labelled with numerical values, then we can ask for shortest paths, by which we mean a path of minimum length, where the length of a path is the sum of its edge labels.

Each problem above yields a shortest path problem - an example of an optimization problem.

In fact, the second and third problems are equivalent.

Survey of graph algorithms: Path finding (continued)

There is an optimization property concerning shortest paths:

> *If p is a shortest path from node u to node v via node w, then the portions of p from u to w and from w to v are both shortest paths.*

Proof: Evident!

Such optimization properties mean that there are direct methods of computing shortest paths: To accumulate shortest paths we need combine only other shortest paths (and not consider any other paths).

This is the basis of both Floyd's Algorithm and Dijkstra's Algorithm.

Survey of graph algorithms: Planarity

Planarity is about depicting graphs in 2-D space: how do we 'draw' graphs and can we do so without edges crossing.

Definition: An embedding of a (directed or undirected) graph in the plane is an allocation of distinct points in the plane to the nodes and distinct continuous lines (not necessarily straight - that is another problem) to the edges, so that no two lines intersect.

There may be more than one 'way' of embedding a graph in the plane. Can all graphs be embedded in the plane? No!

A graph that can be embedded in the plane is called a planar graph.

Hopcroft and Tarjan introduced a linear-time planarity algorithm (1974) based on Depth-First Search

Survey of graph algorithms: Graph colouring

A colouring of a graph with k colours is an allocation of the colours to the nodes of the graph, such that each node has just one colour and nodes linked by an edge have different colours.

To determine whether a graph can be coloured with k ($k \geq 3$) colours is an NP-complete problem.

Thus the only algorithms that exist in general for colourability are exhaustive unlimited back-tracking algorithms, and hence are exponential-time.

Survey of graph algorithms: Finale

The 4-colouring of planar graphs... is a celebrated problem.

Proposition Every planar graph can be coloured with just 4 colours.

- Discussed as a possibility in the 1850s,
- 1879: Kempe publishes a 'proof' of the 4-colour theorem,
- 1890: Heawood finds error in Kempe's proof, but shows Kempe's proof establishes the 5-colourability of planar graphs,
- Since then finding a proof of 4-colourability has been a catalyst for much combinatorial mathematics,
- 1977: Computer-aided proof of 4-colourability: reduced the graphs required to be coloured to a finite number (over 1000) and then used a computer to generate colourings.