COMP20010: Algorithms and Imperative Programming

Lecture 2

Data structures for binary trees

Priority queues

Lecture outline

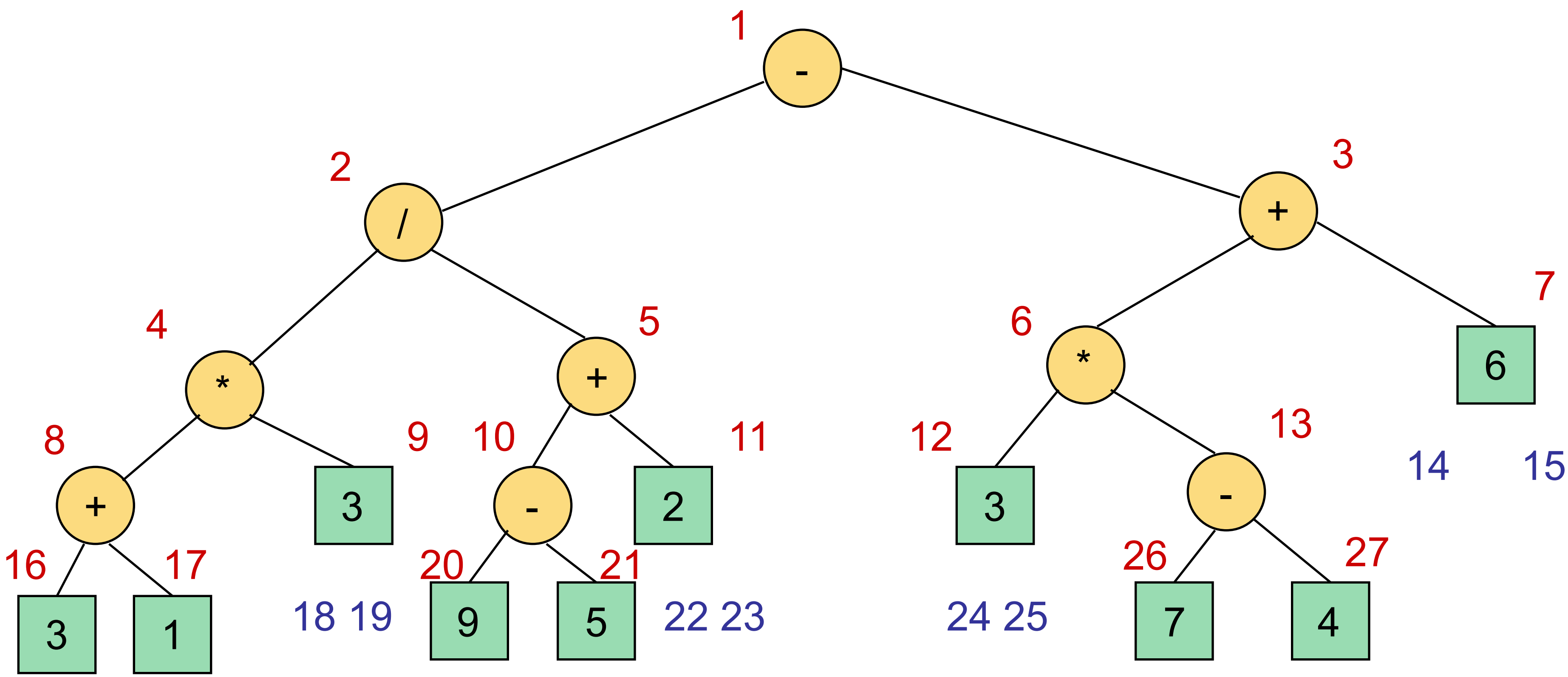
- Different data structures for representing binary trees (vector-based, linked), linked structure for general trees;
- Priority queues (PQs), sorting using PQs;

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 $p(v)=2p(u)$;
  - If v is the right child of the node u, then $p(v)=2p(u)+1$;
- 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

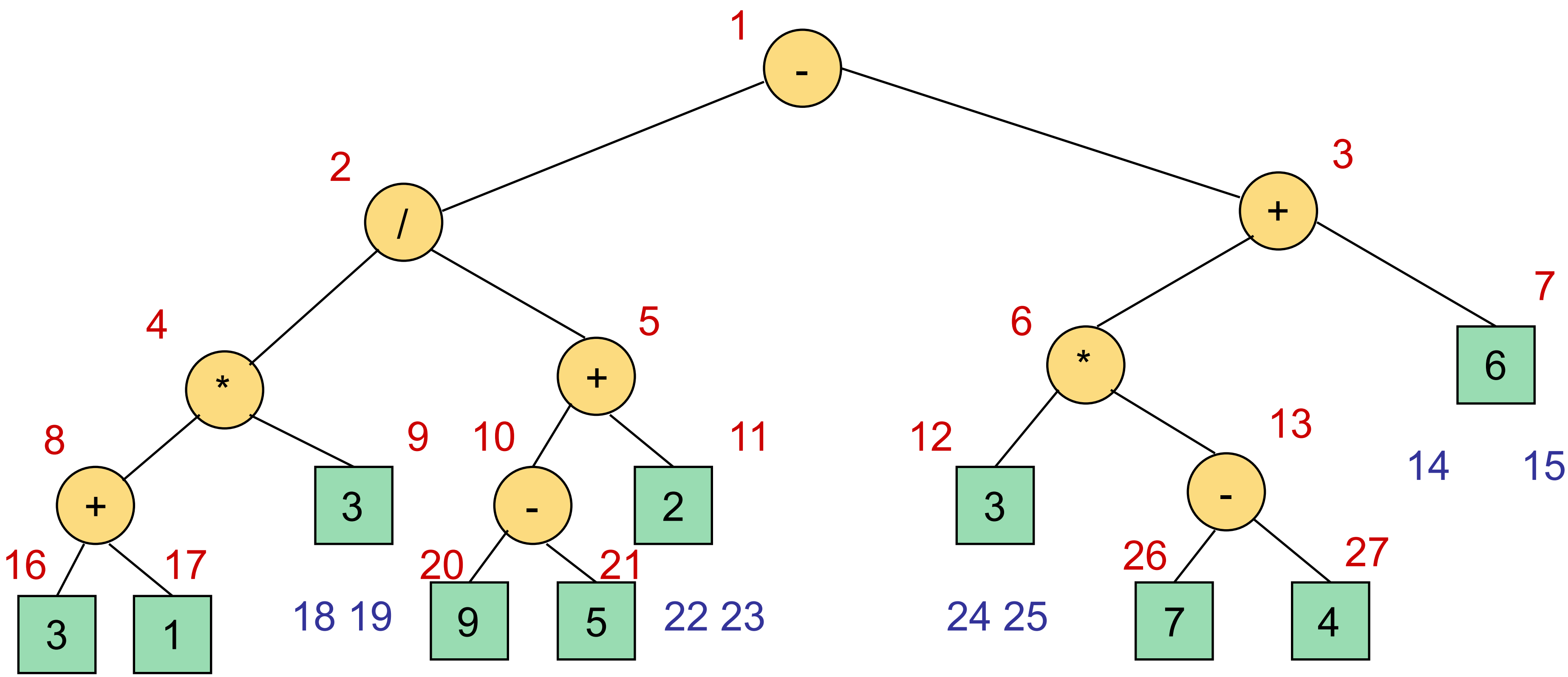
- ((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))



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

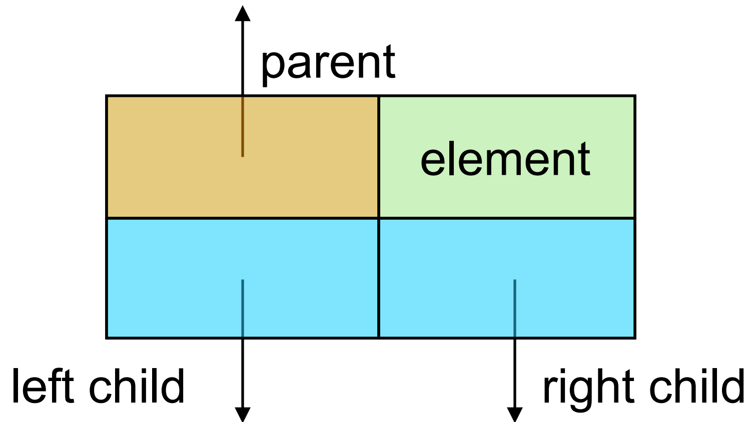
Data structures for representing trees
A vector-based data structure

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

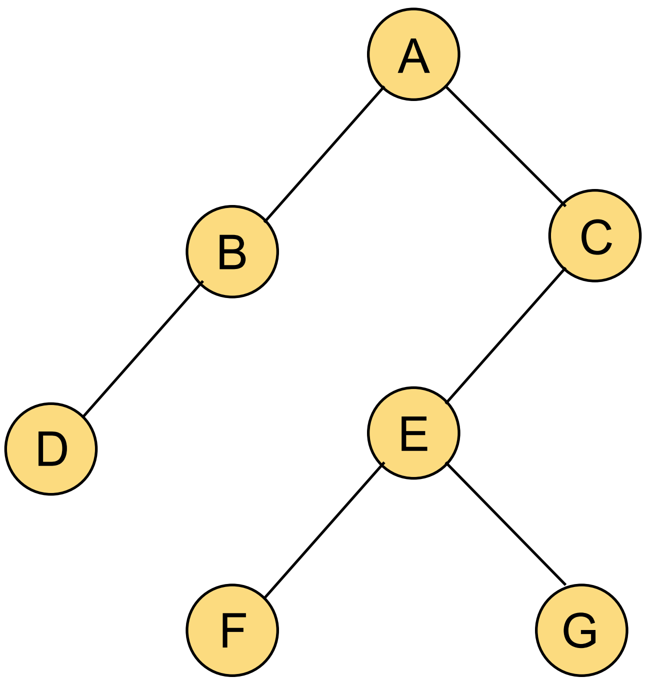
Running times of the methods when a binary tree T is implemented as a vector

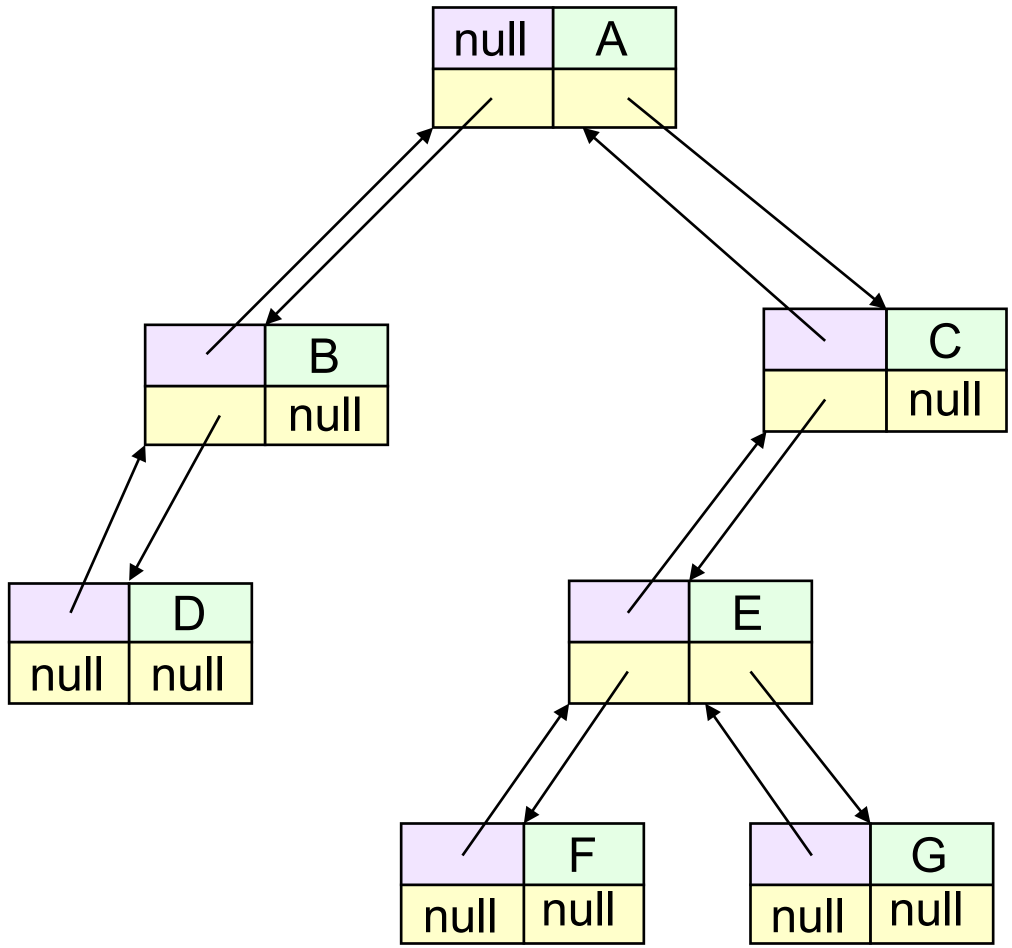
Data structures for representing trees
A linked data structure

- 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.



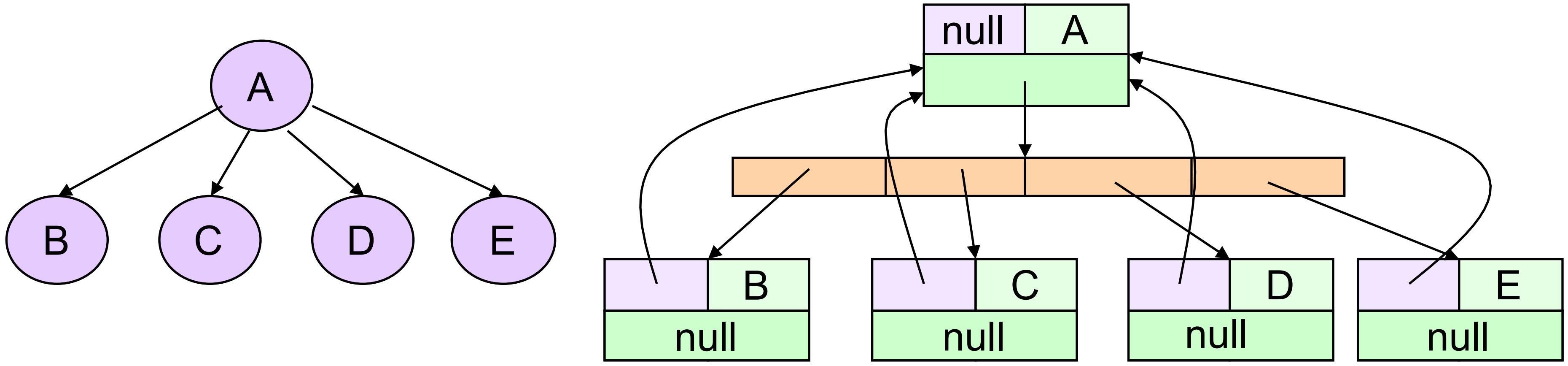
Data structures for representing trees
A linked data structure for binary trees





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 $O(n^2)$.