COMP 6651 - Design and Analysis of Algorithms

Brigitte Jaumard

Winter 2022

Lecture 9

Efficient implementation of a branch-and-bound method

Branch-and-bound Method

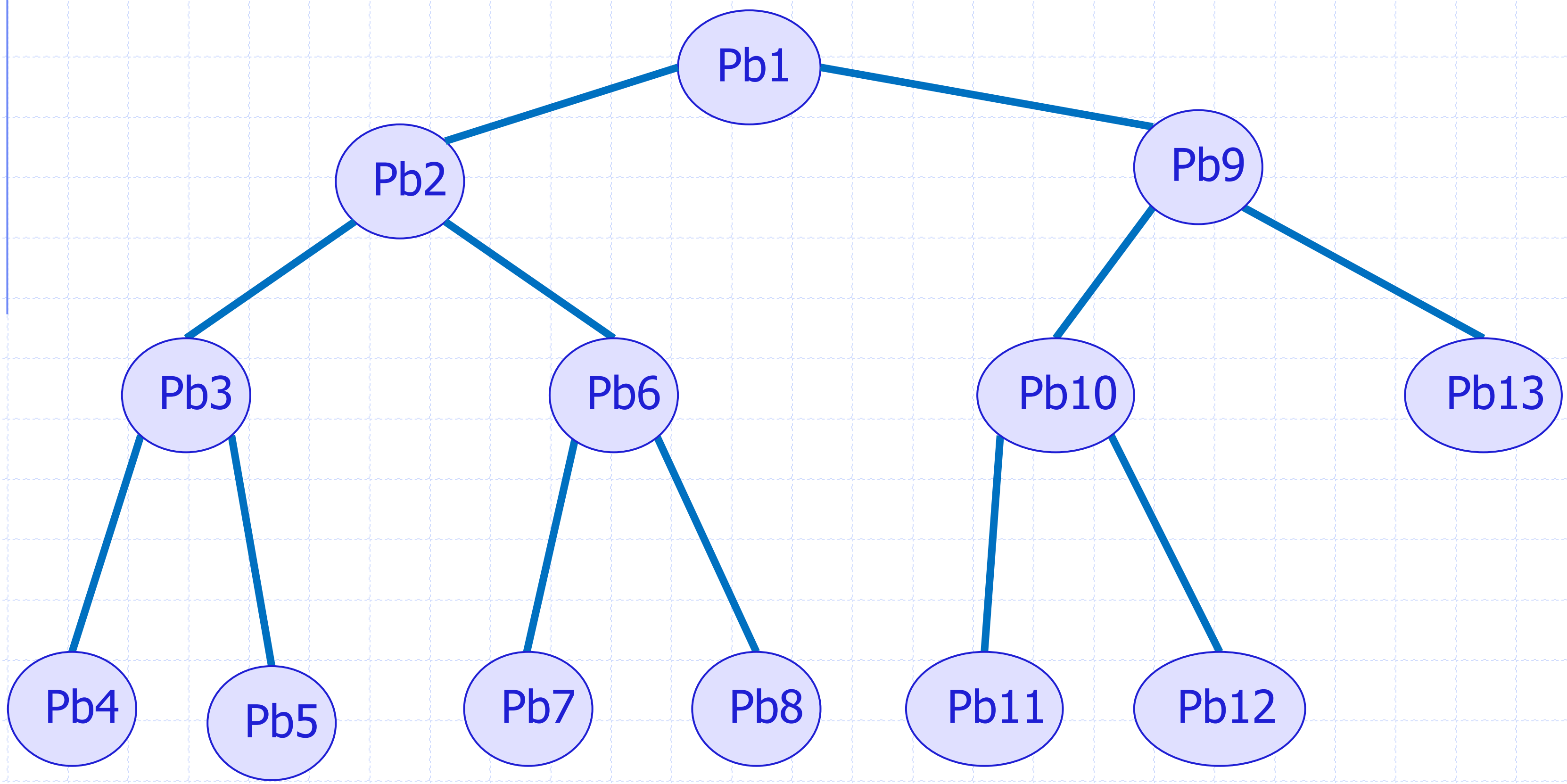
- Bounds
  - Lower bound
  - Upper bound
- Branching/partition scheme
- Exploration scheme

Exploration Schemes

- Depth first search (DFS)
- Breadth first search (BFS)
- Best first search

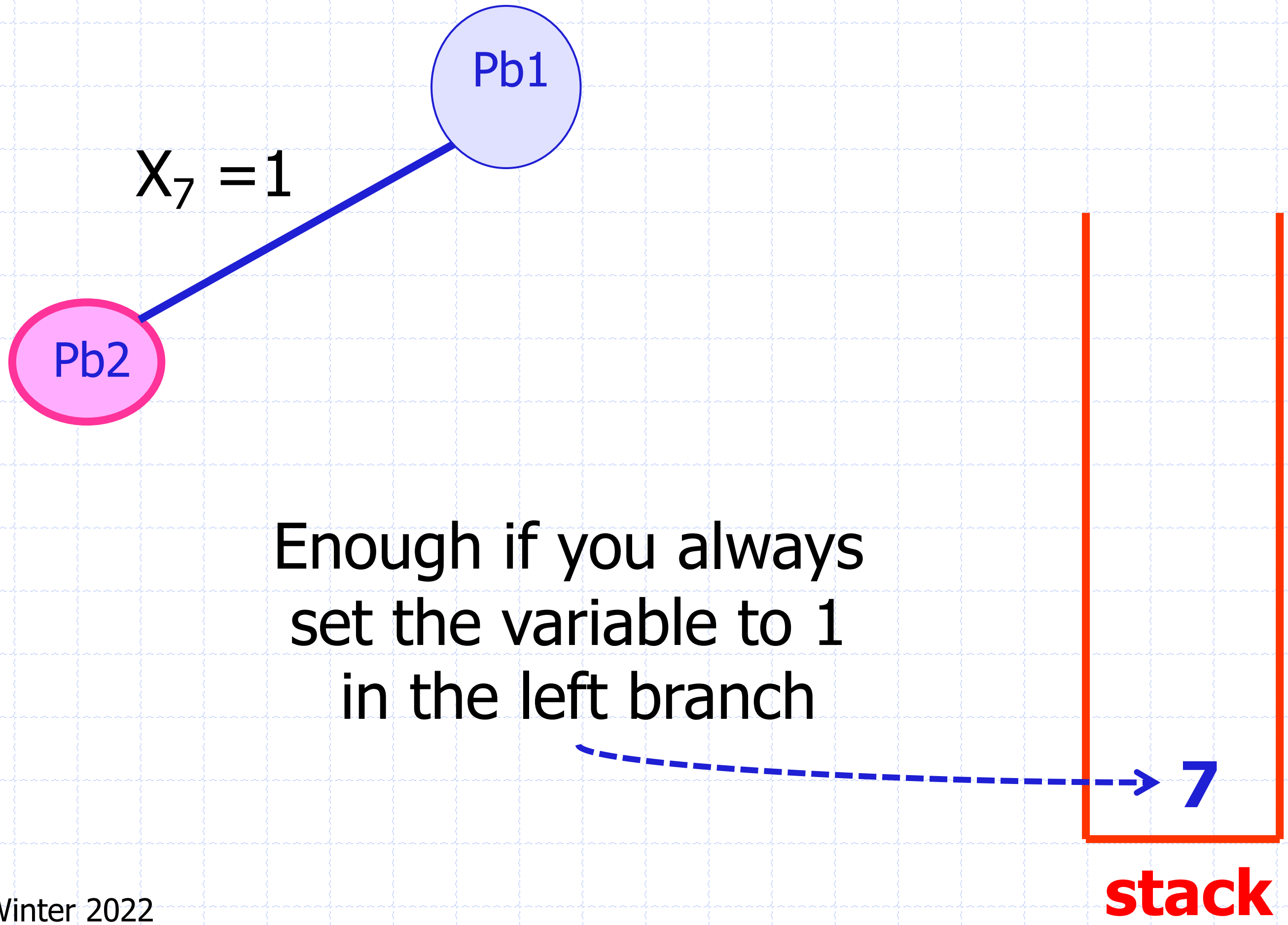
Depth first search

- Visit a node's children before its siblings



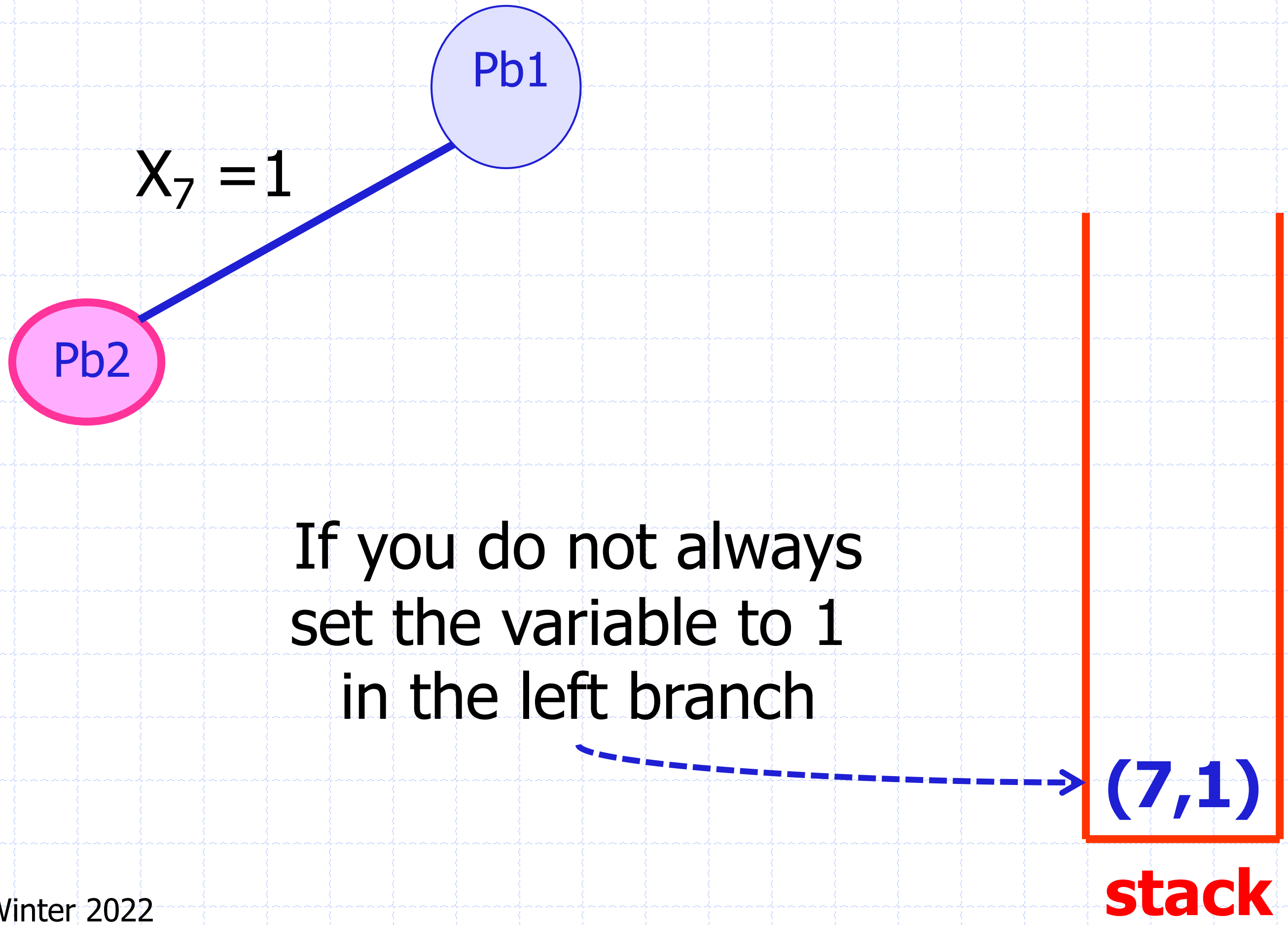
Depth first search

- Implementation with a stack



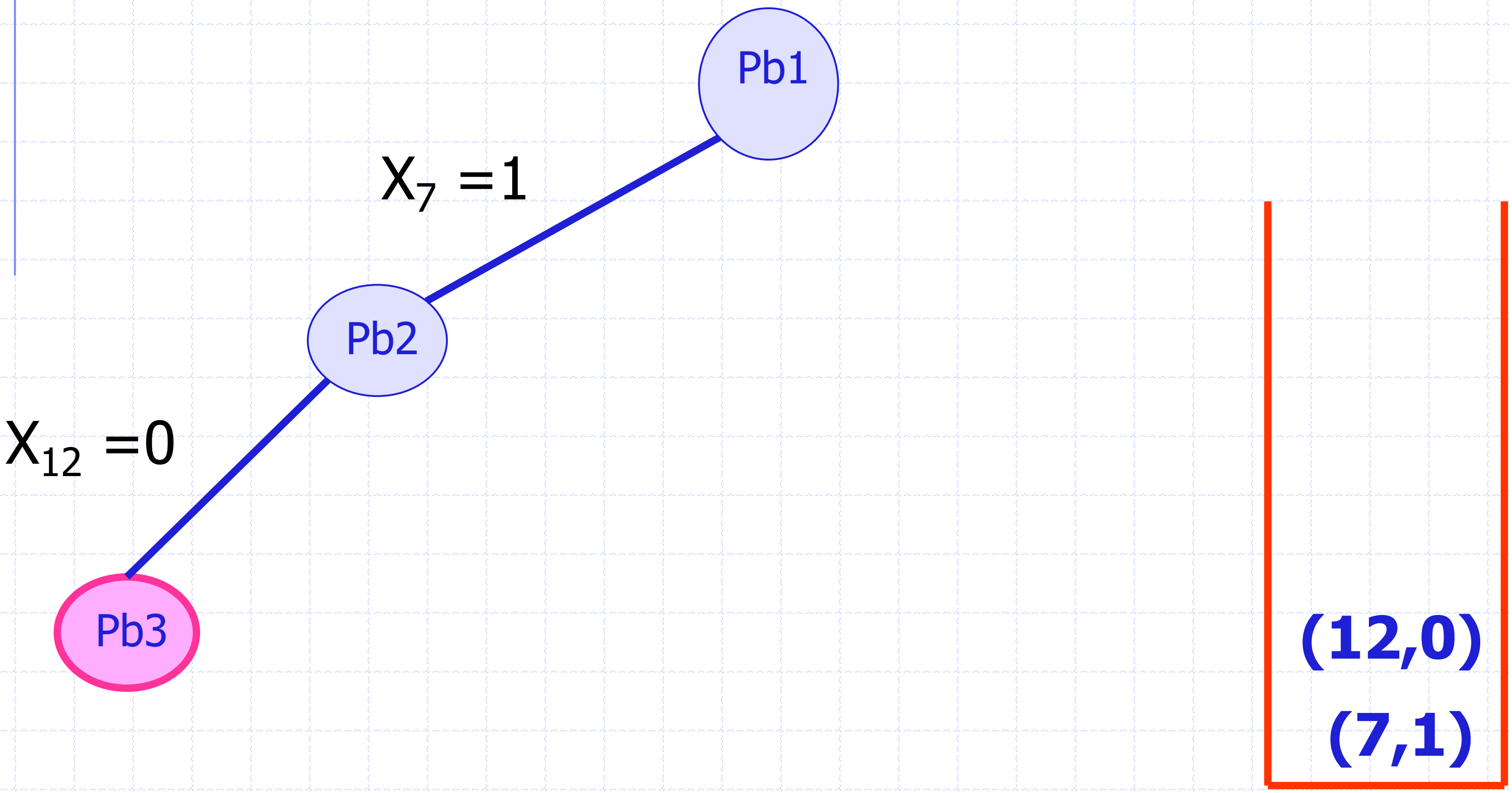
Depth first search

- Implementation with a stack



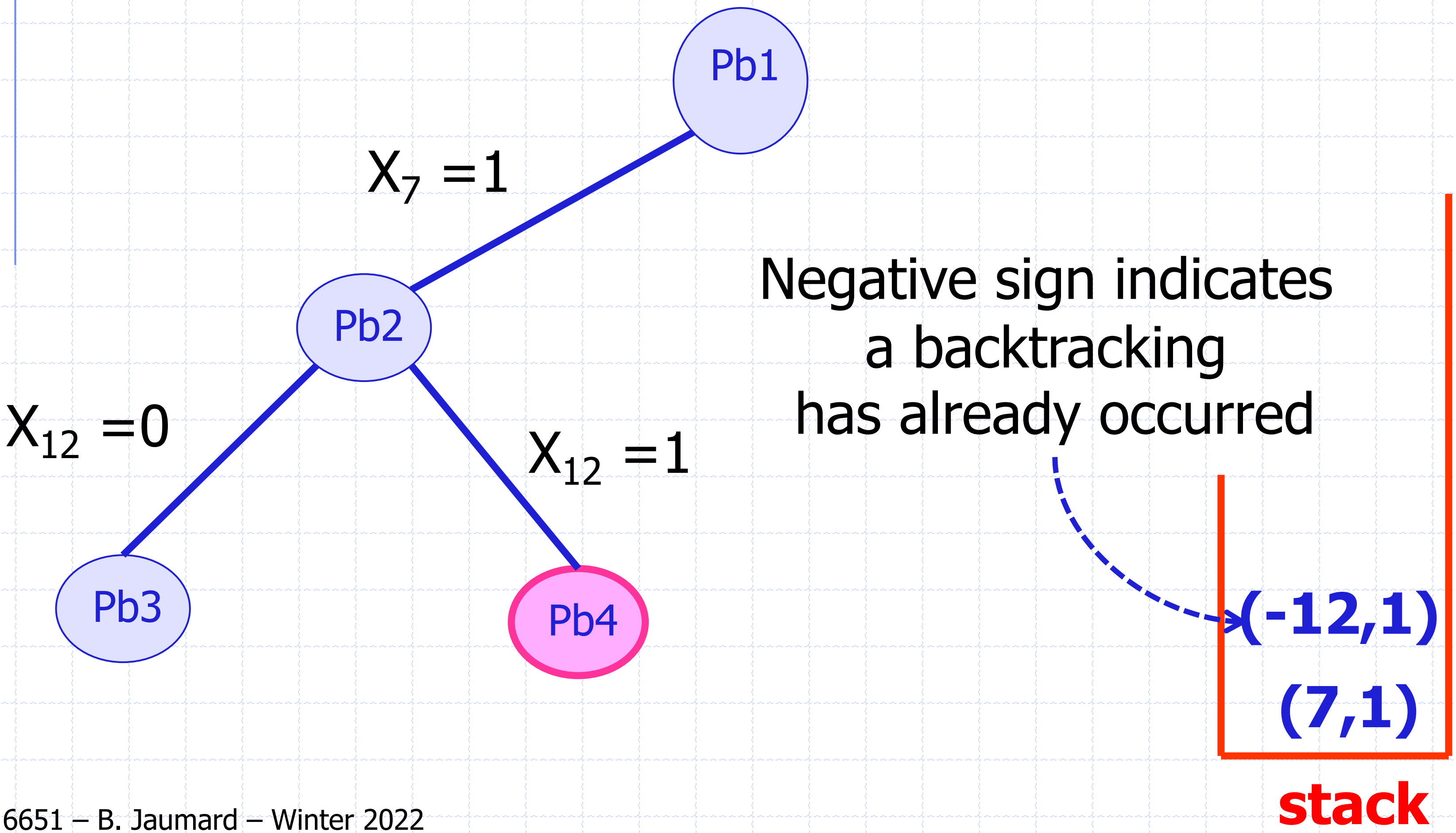
Depth first search

- Implementation with a stack



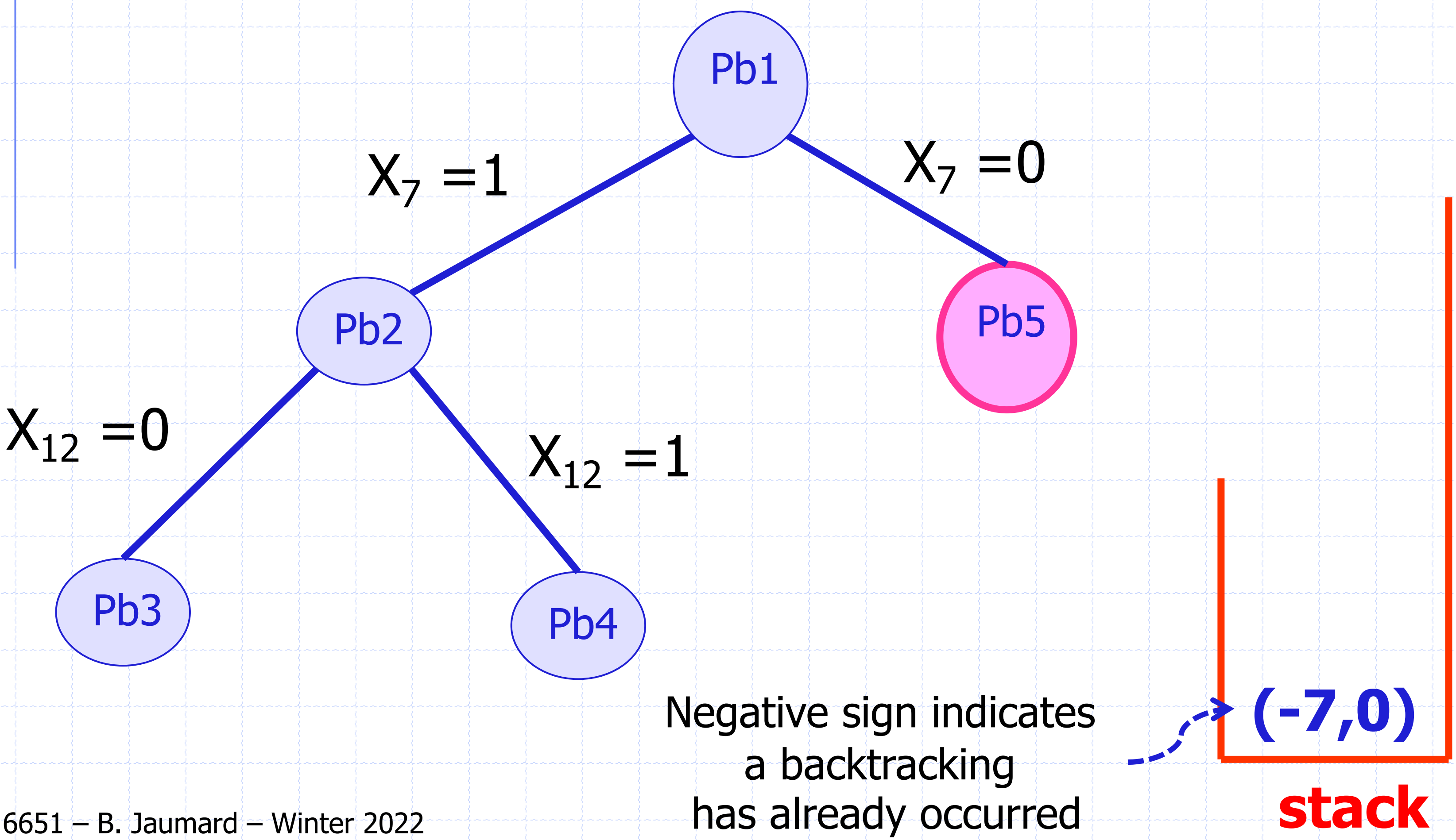
Depth first search

- Implementation with a stack



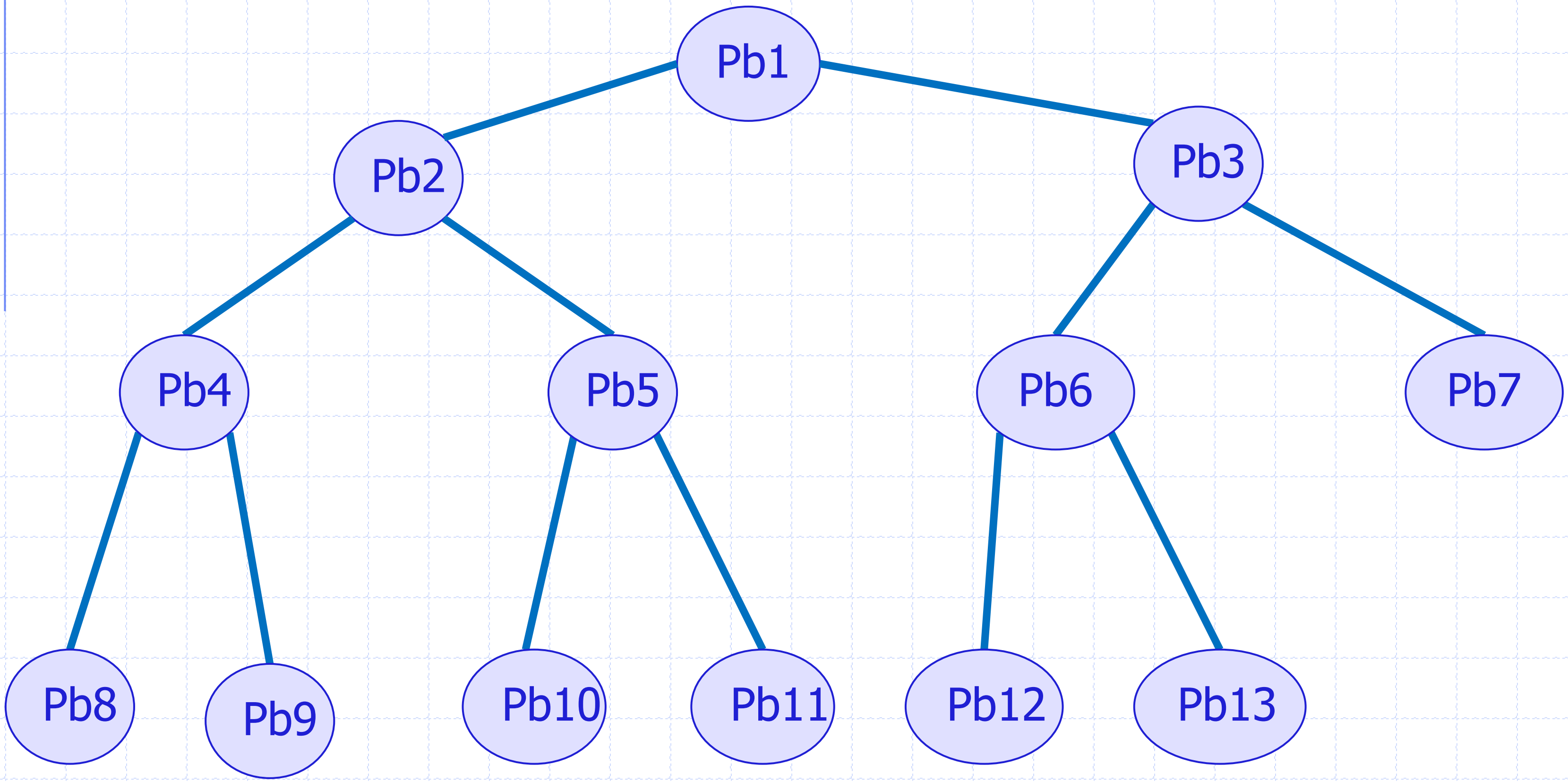
Depth first search

- Implementation with a stack



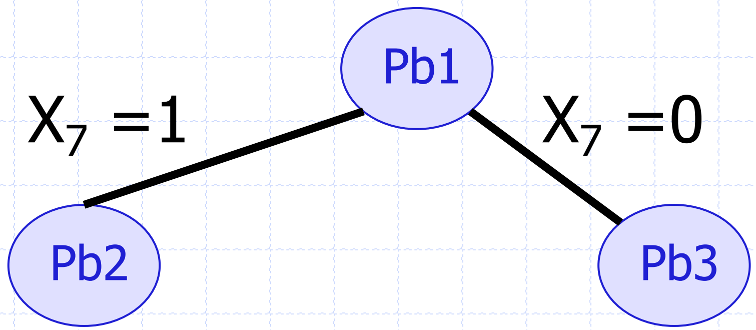
Breadth first search

- Visit a node's siblings before its children



Breadth first search

- Implementation: FIFO

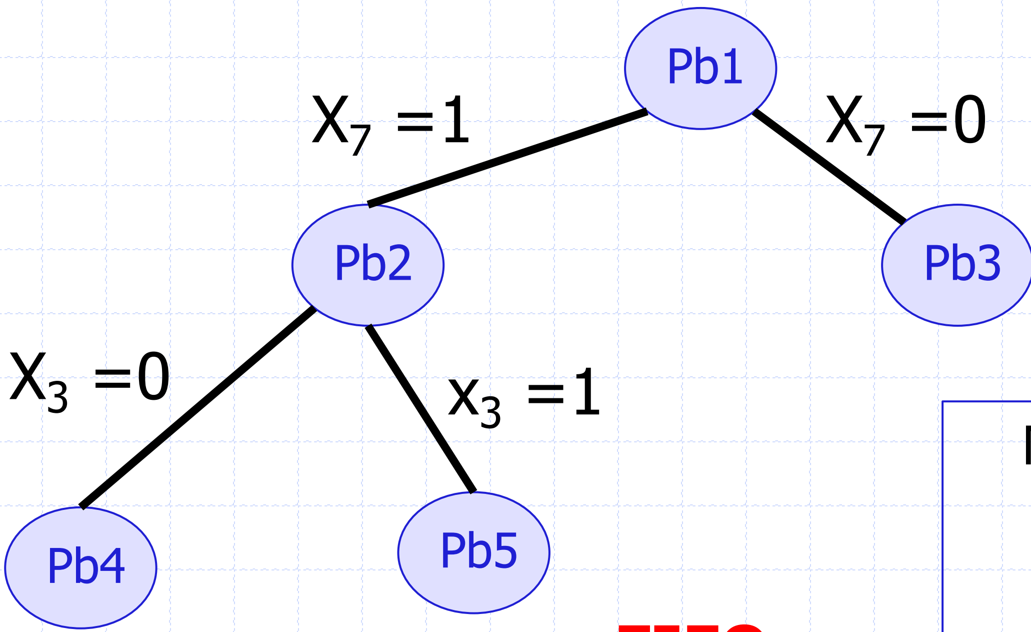


FIFO

tail 7 -7 head

Breadth first search

- Implementation: FIFO



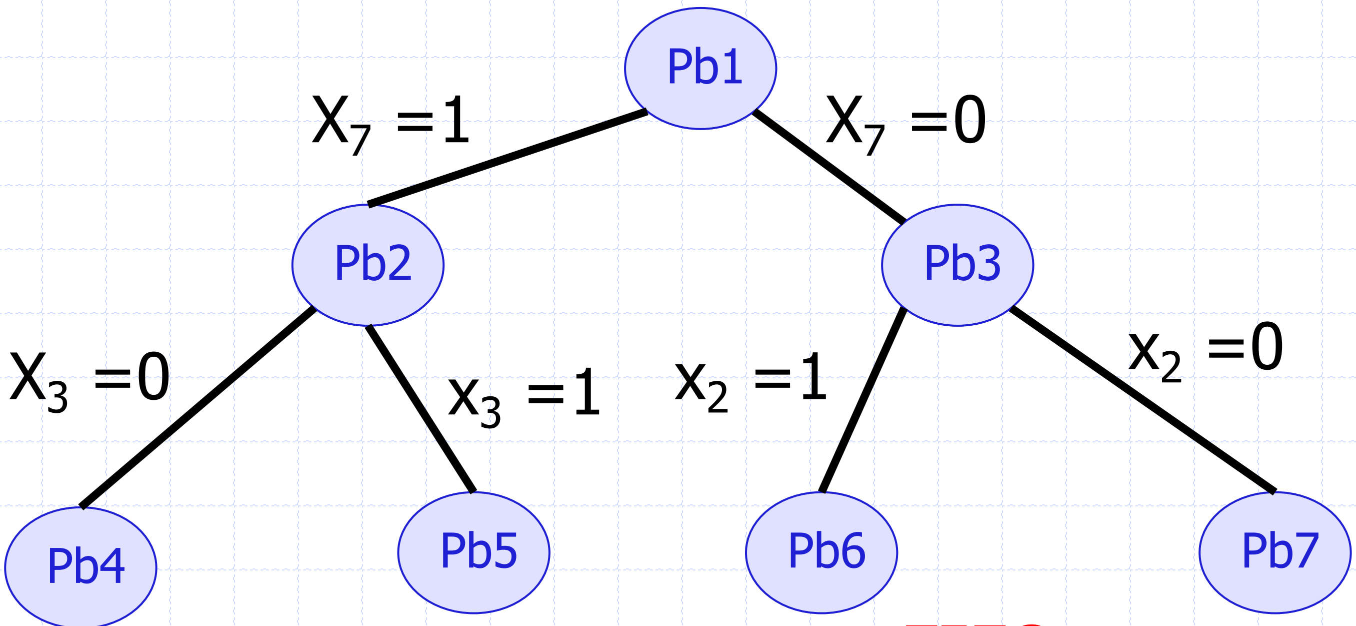
Negative (resp. positive) sign means branching variable is set to 0 (resp. 1)

FIFO

tail ~~7~~ -7 -3 3 head

Breadth first search

- Implementation: FIFO

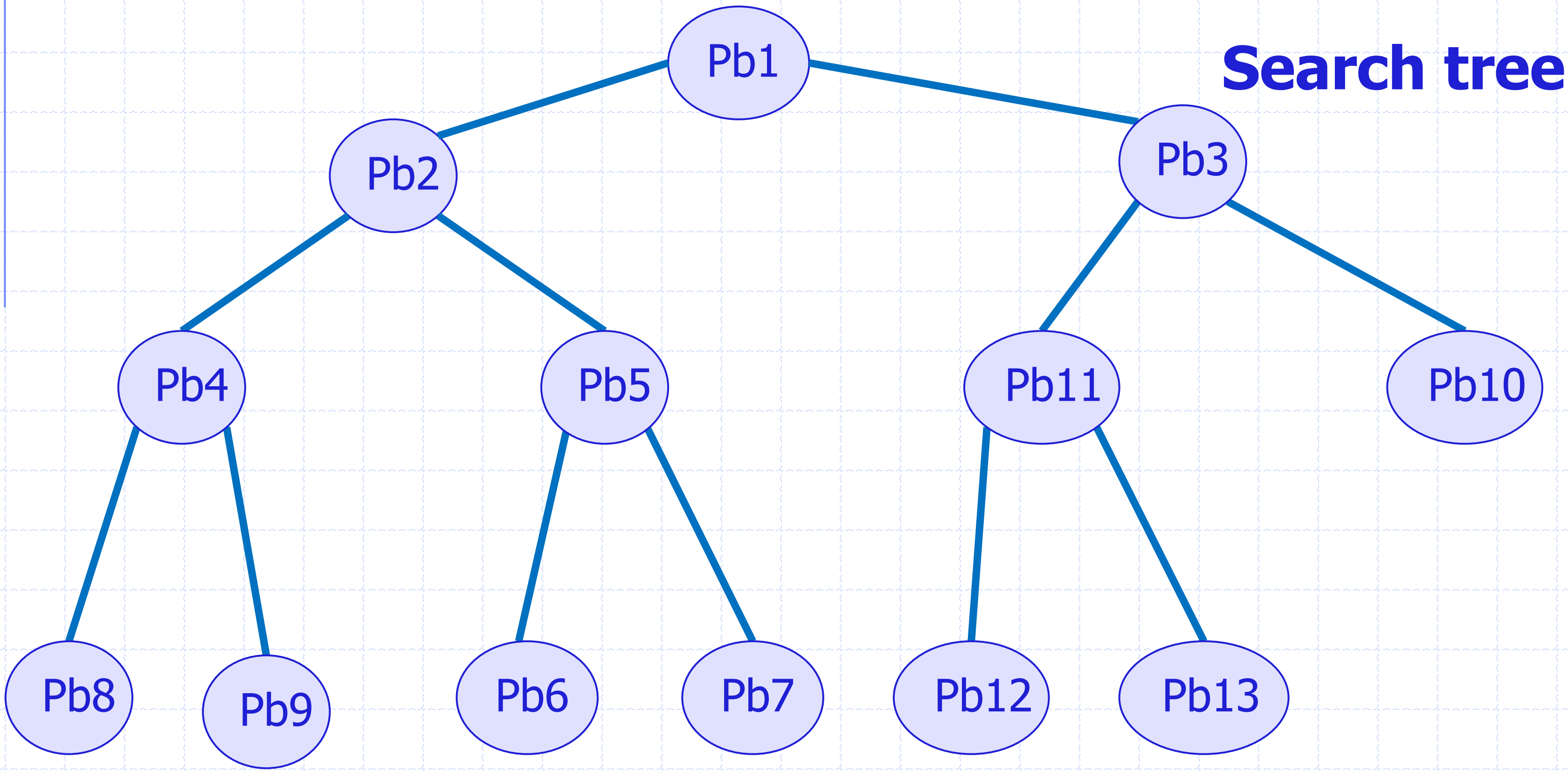


FIFO

tail ~~7~~ ~~-7~~ -3 3 2 -2 head

Best first search

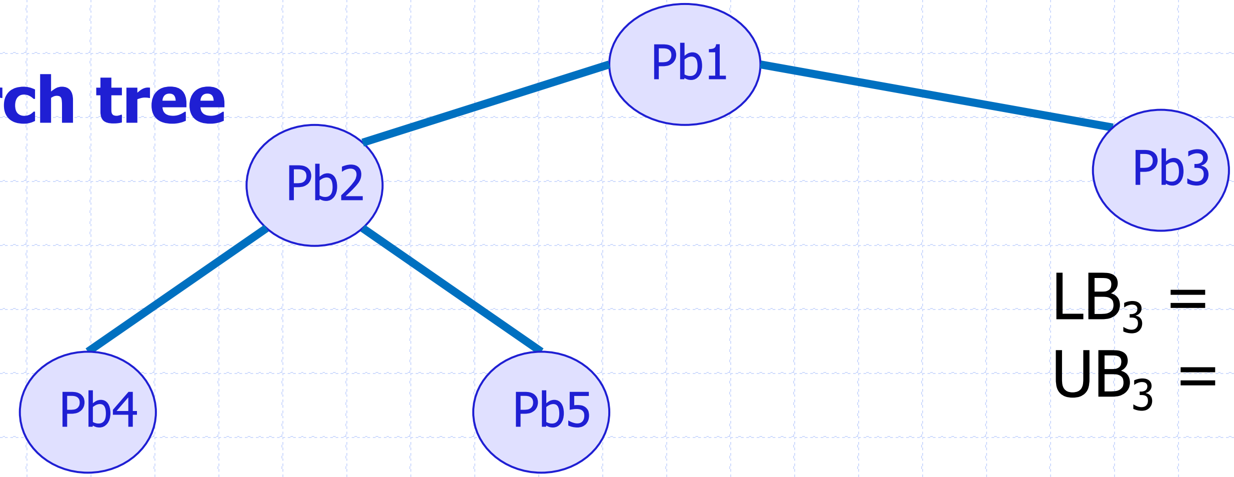
- Visit the best node



Best first search

- Visit the best node (maximization problem)

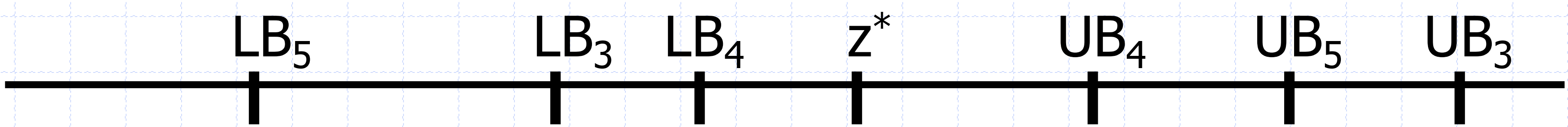
Search tree



$LB_3 = 13$
$UB_3 = 46$

$LB_4 = 23$
$UB_4 = 40$

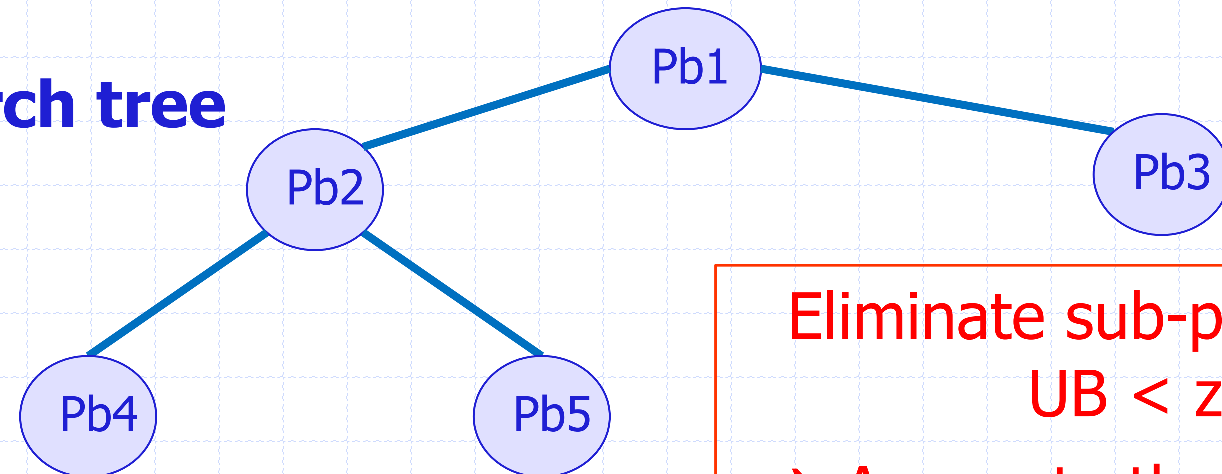
$LB_5 = 10$
$UB_5 = 44$



Best first search

- Visit the best node (maximization problem)

Search tree



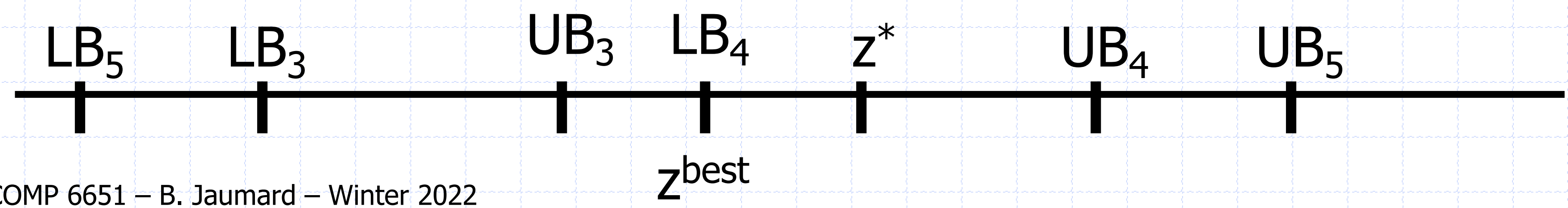
$LB_3 = 13$
$UB_3 = 20$

$LB_4 = 23$
$UB_4 = 40$

$LB_5 = 10$
$UB_5 = 46$

Eliminate sub-pbs such that
$UB < z^{best}$

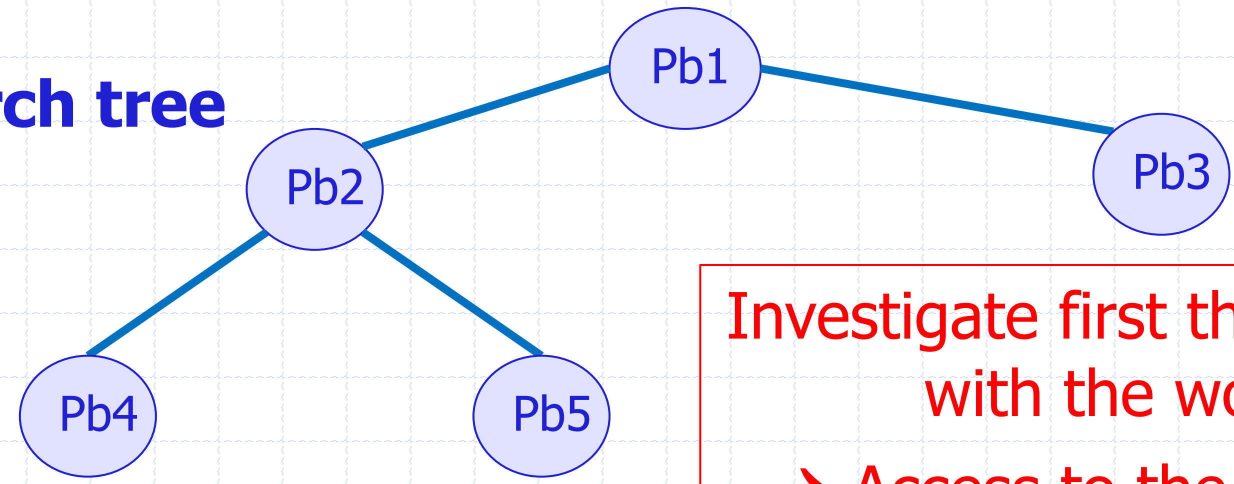
$\rightarrow$ Access to the subproblems in the increasing order of their UB



Best first search

- Visit the best node (maximization problem)

Search tree



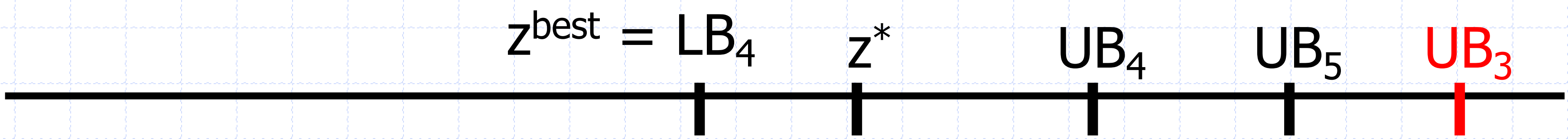
$LB_3 = 13$
$UB_3 = 46$

$LB_4 = 23$
$UB_4 = 40$

$LB_5 = 20$
$UB_5 = 44$

Investigate first the subproblem with the worst UB

→ Access to the subproblems in the decreasing order of their UB



Required data structure

- Need for a data structure with a fast access to
  - The min value
  - The max value

  → **Max-min heap** (or a double ended priority queue)

Min-Max Heap

Single-Ended Priority Queue (Max-Heap)

- n items
- Data structure that supports
  - Find the maximum value (FindMax) $O(1)$
  - Delete the maximum value (DeleteMax) $O(\log n)$
  - Add a new value x (Insert(x)) $O(\log n)$
- To build a Max-Heap $O(n)$

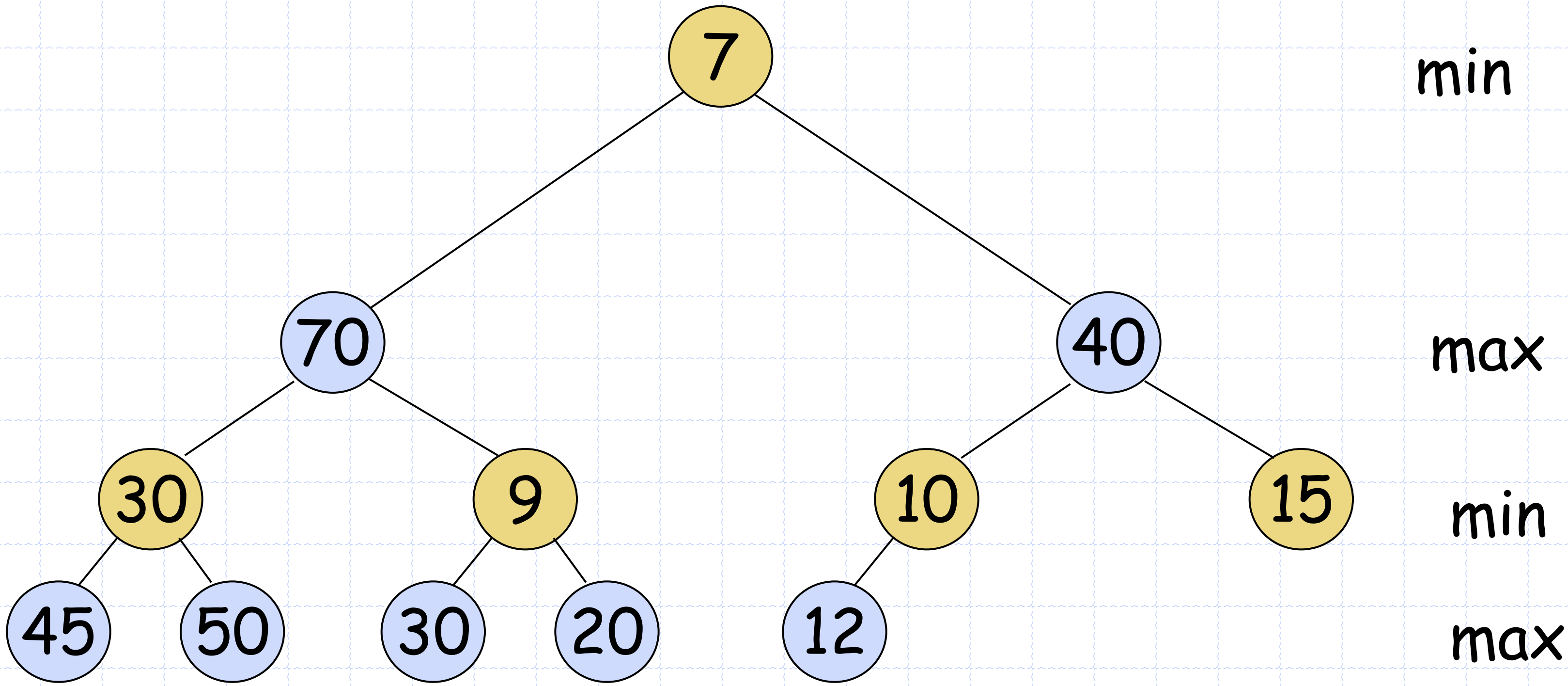
Reference

- M.D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte, Min-Max Heaps and Generalized Priority Queues, *Communications of the ACM*, October 1986, Volume 29, Number 10, 996-1000.

Min-max Heap: Definition

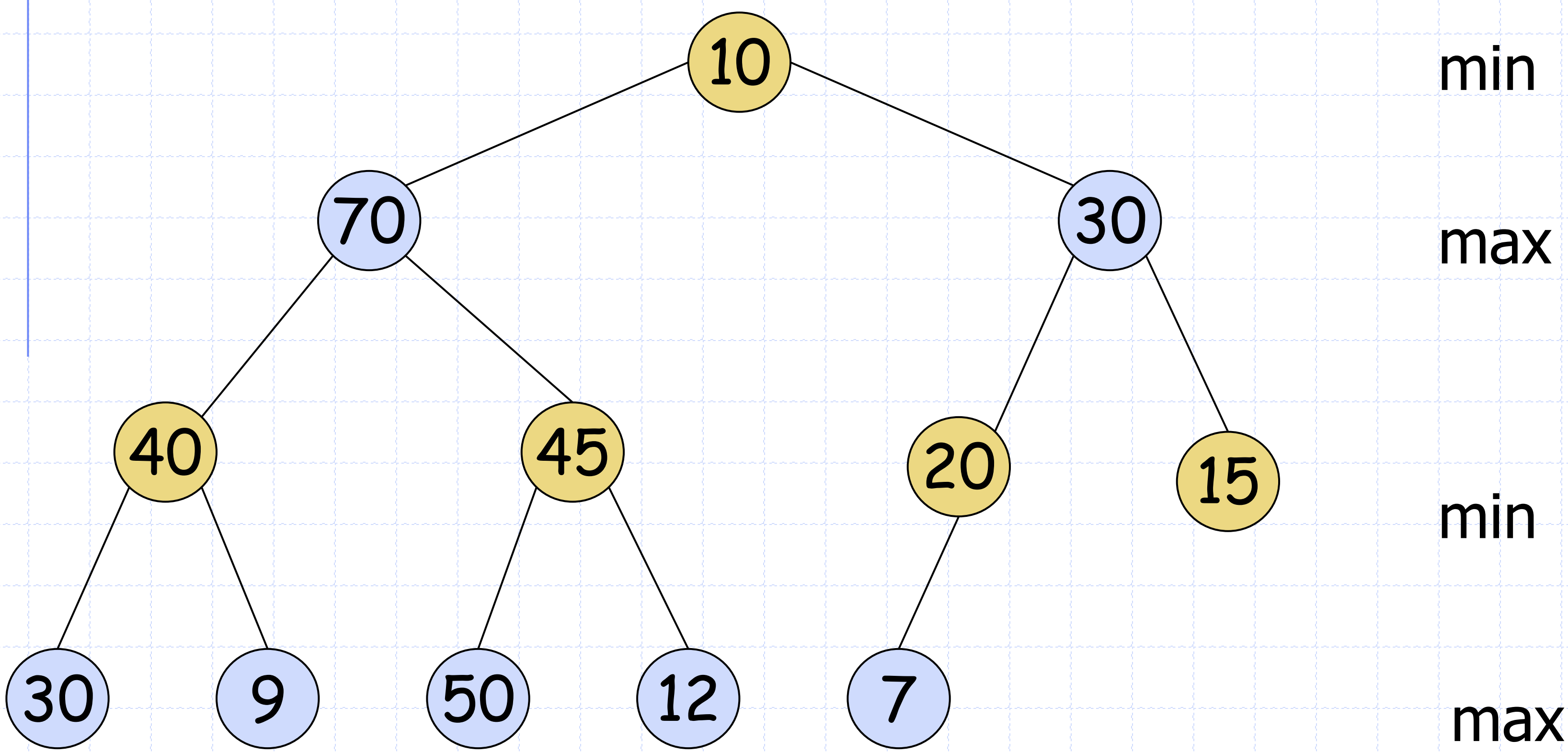
- Max-min heap
  - Binary tree *T*
  - T has the heap-shape: all leaves lie on at most two adjacent levels, and the leaves on the last level occupy the leftmost positions; all other levels are complete.
  - T is min-max ordered: values stored at nodes on even (odd) levels are smaller (greater) than or equal to the values stored at their descendants (if any) where the root is at level 0.
- Implementation
  - One-dimensional array *A*
  - Parent(A[i]) $\rightarrow$ A[$i/2$]
  - Left(A[i]) $\rightarrow$ A[$2i$]
  - Right(A[i]) $\rightarrow$ A[$2i+1$]
- Complexity: same as for a heap for all basic operations

Min-max Heap: Example

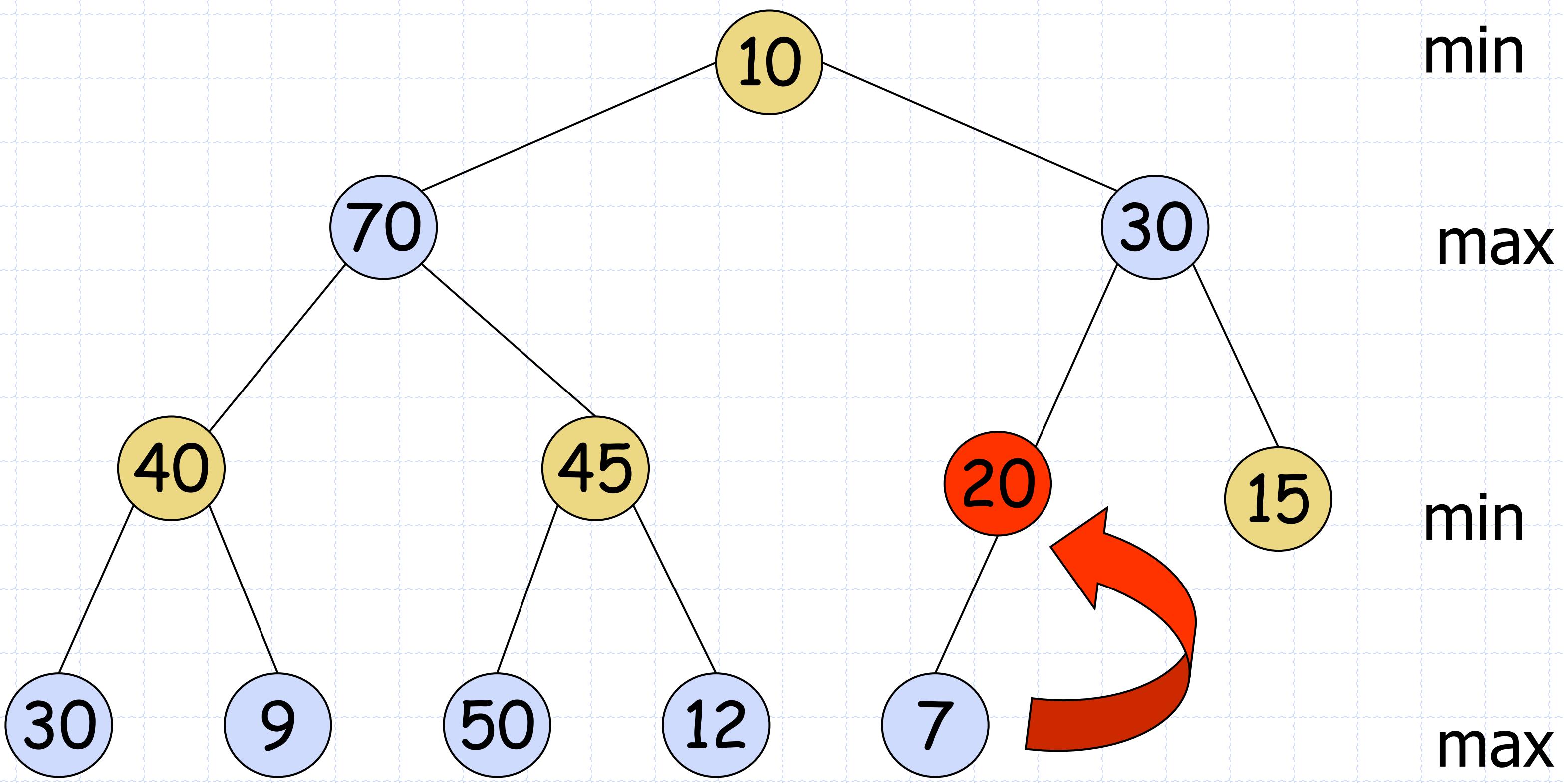


Smallest value is stored at the root of *T*
Largest value is stored at one of the root's children

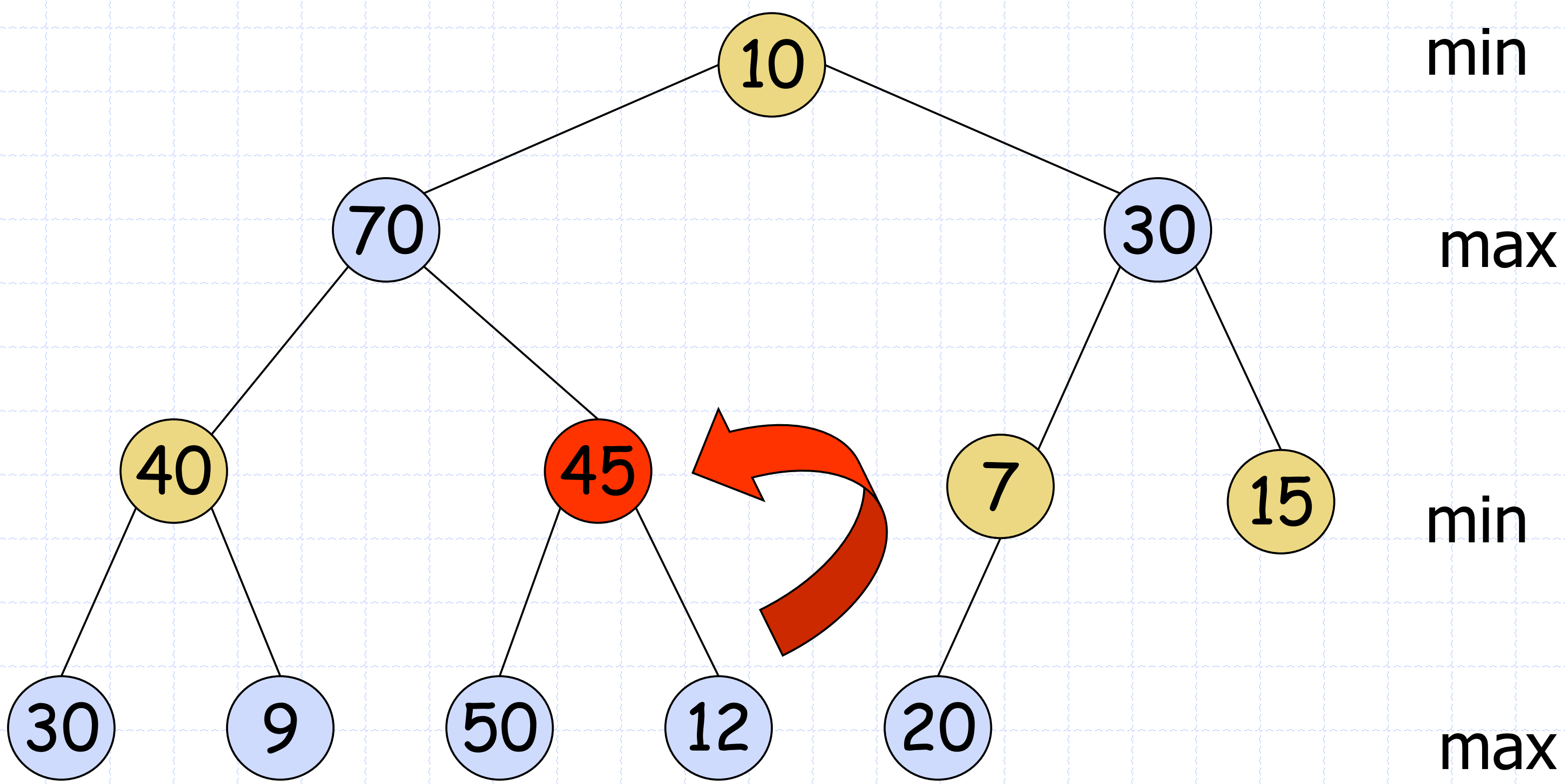
Building a Max-Min Heap: Bottom-up Fashion



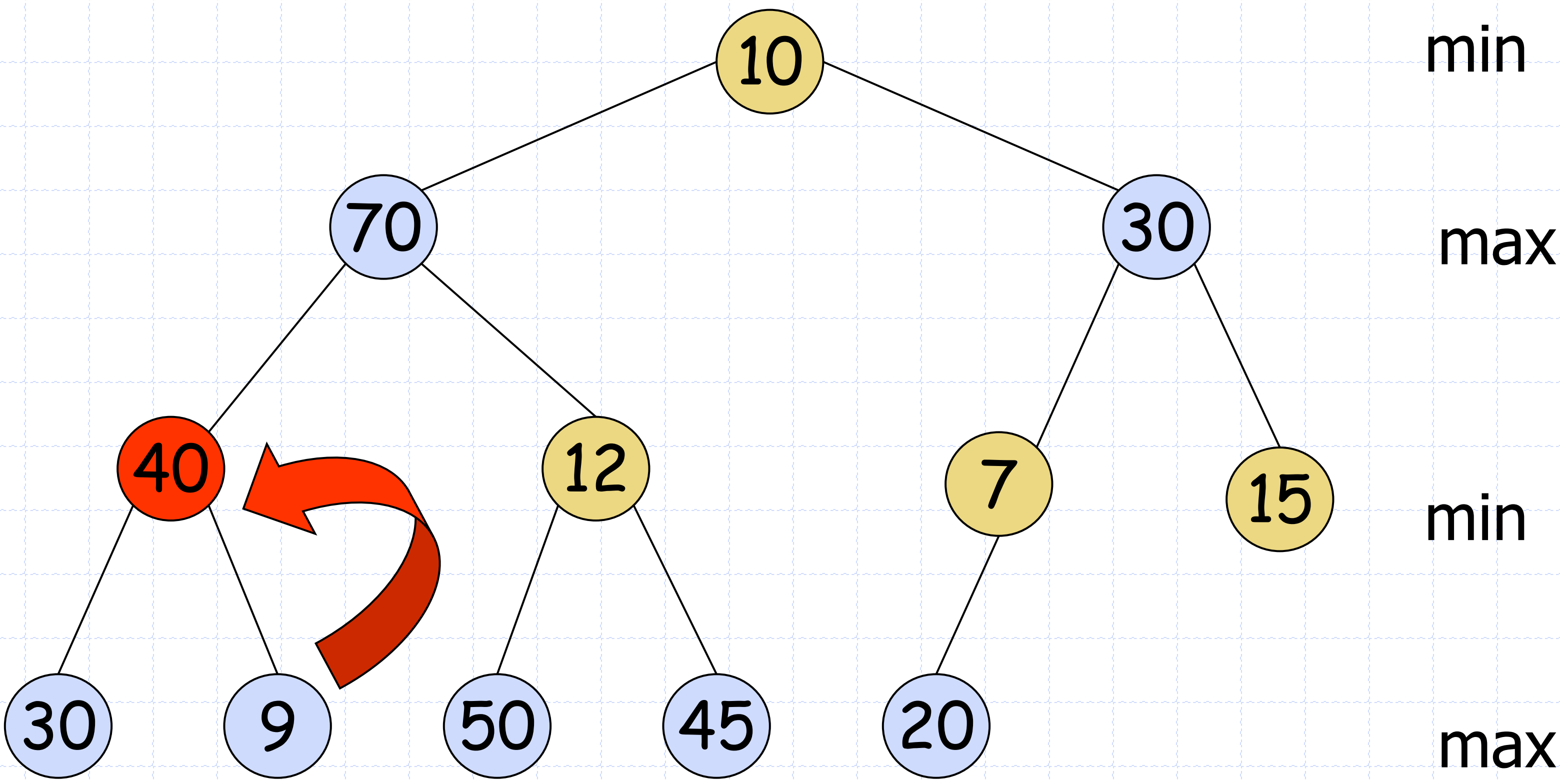
Building a Max-Min Heap



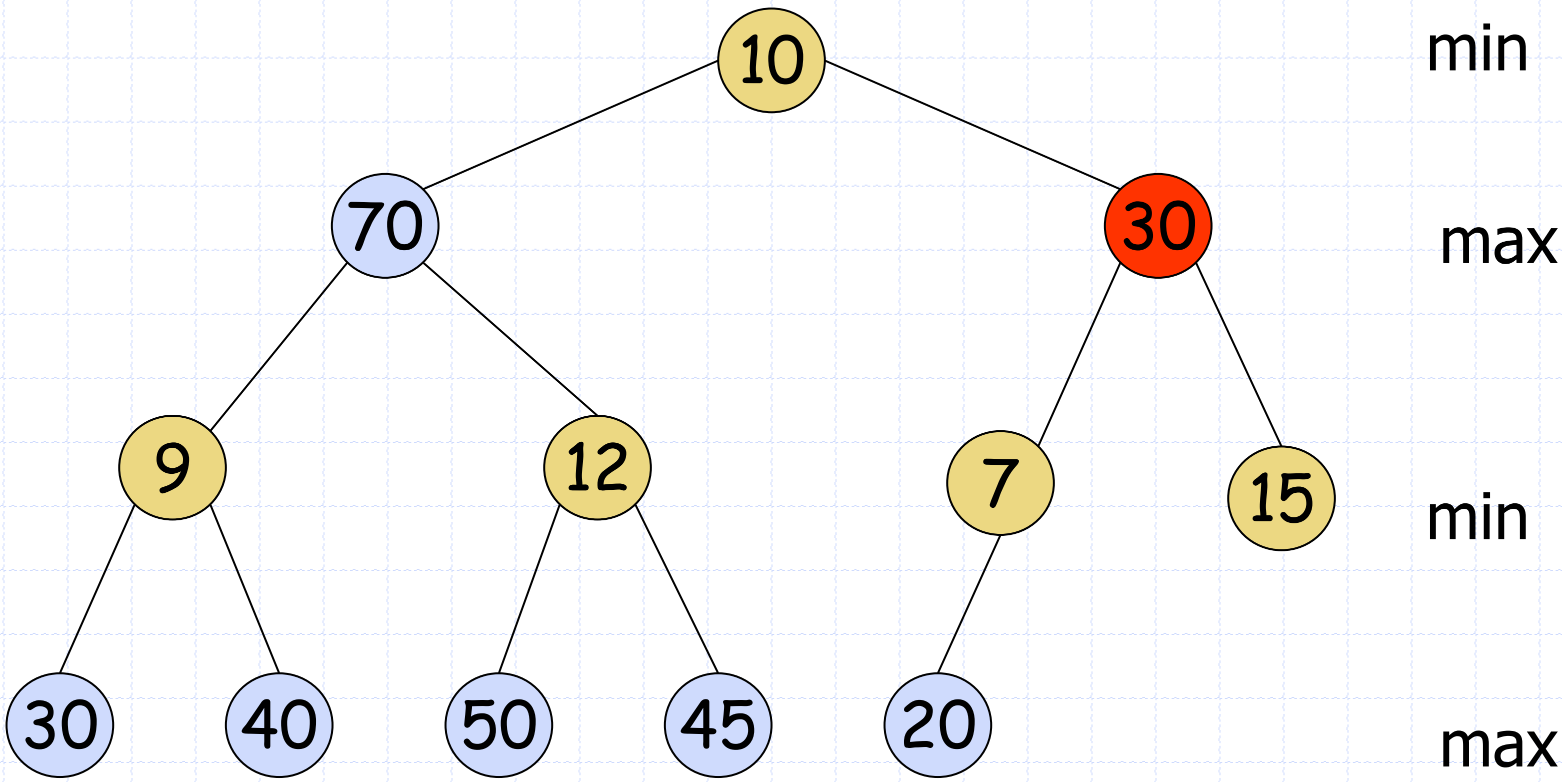
Building a Max-Min Heap



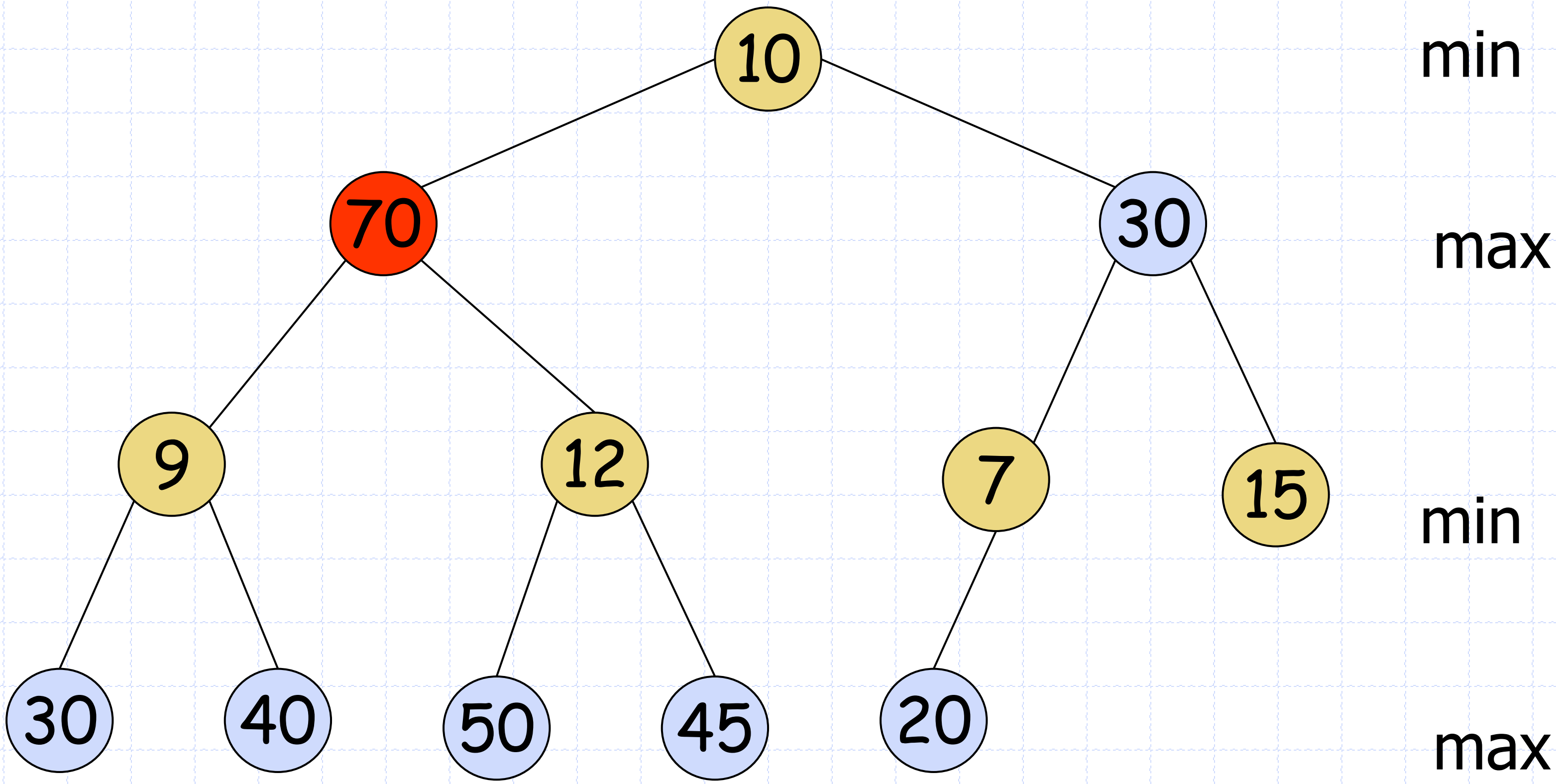
Building a Max-Min Heap



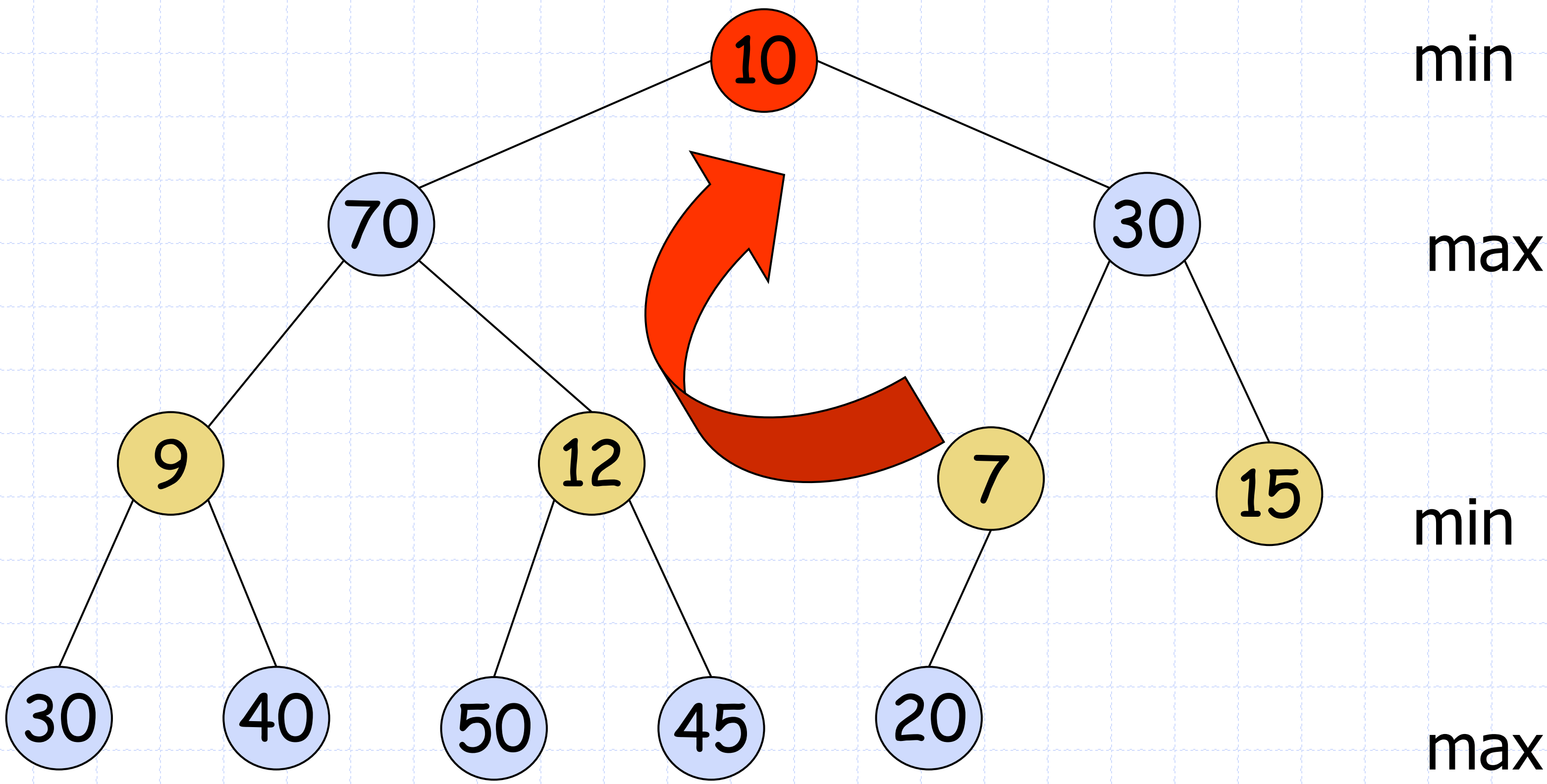
Building a Max-Min Heap



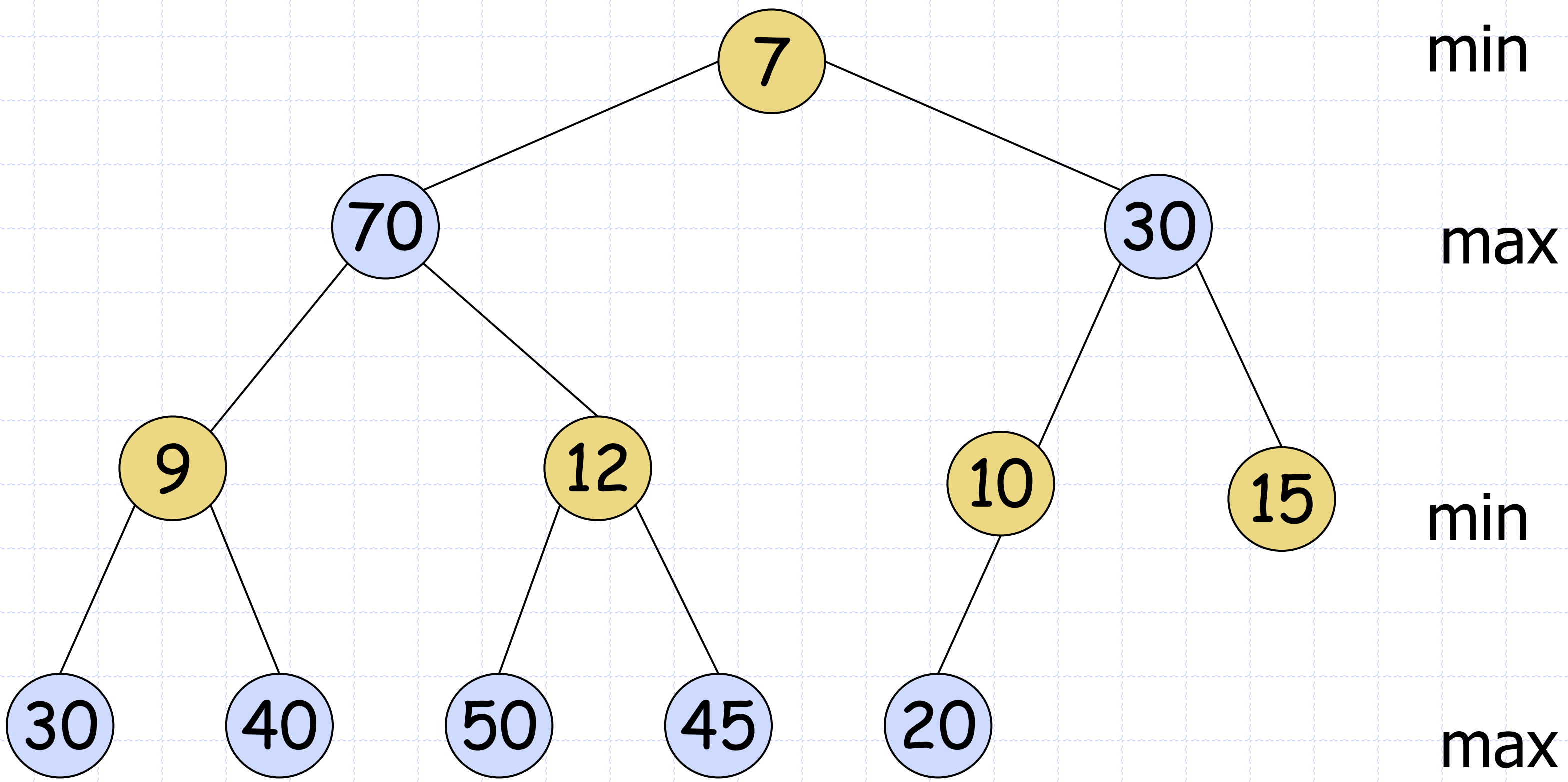
Building a Max-Min Heap



Building a Max-Min Heap



Building a Max-Min Heap



Algorithms

procedure TrickleDown(*i*)

<* *i* is the position in the array *>

if *i* is on a min level **then**

TrickleDownMin(*i*)

else

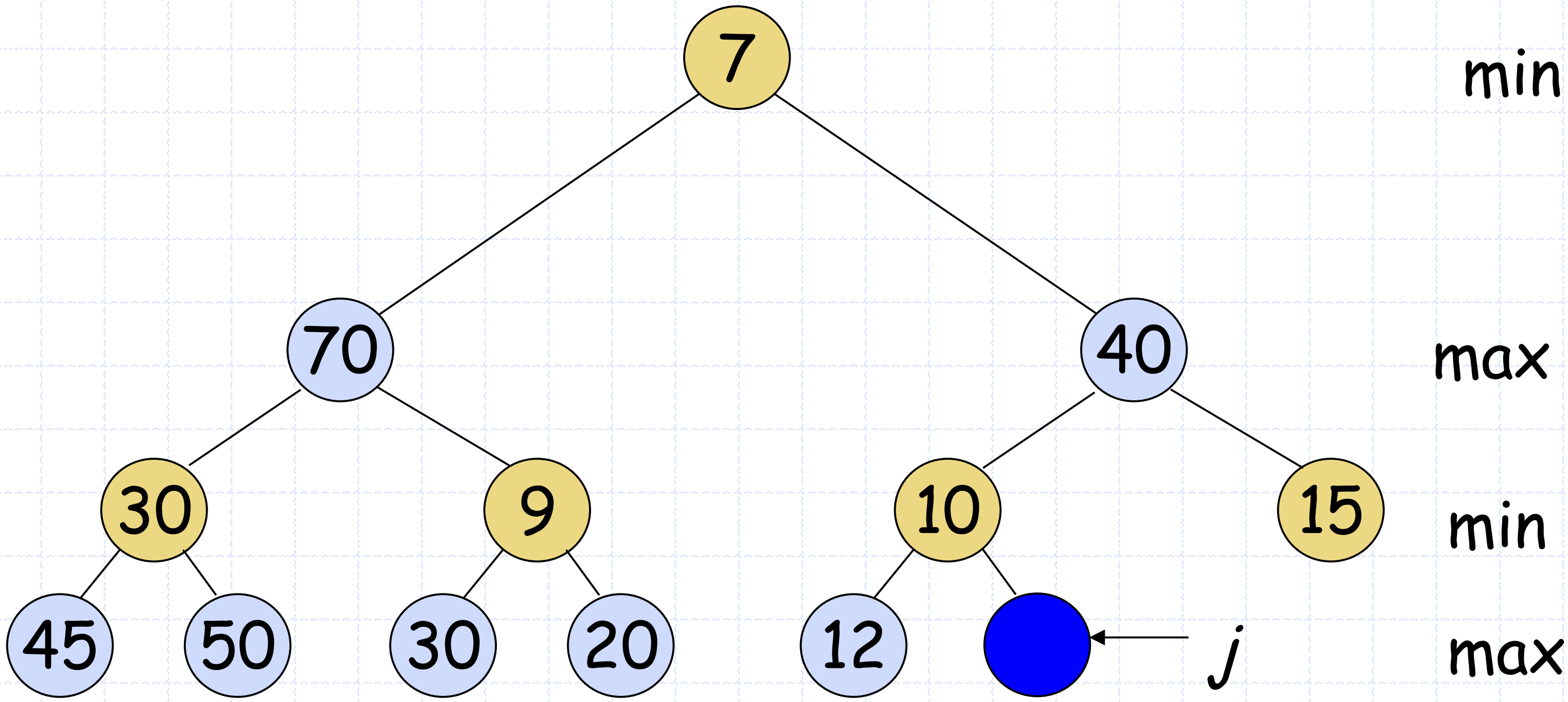
TrickleDownMax(*i*)

endif

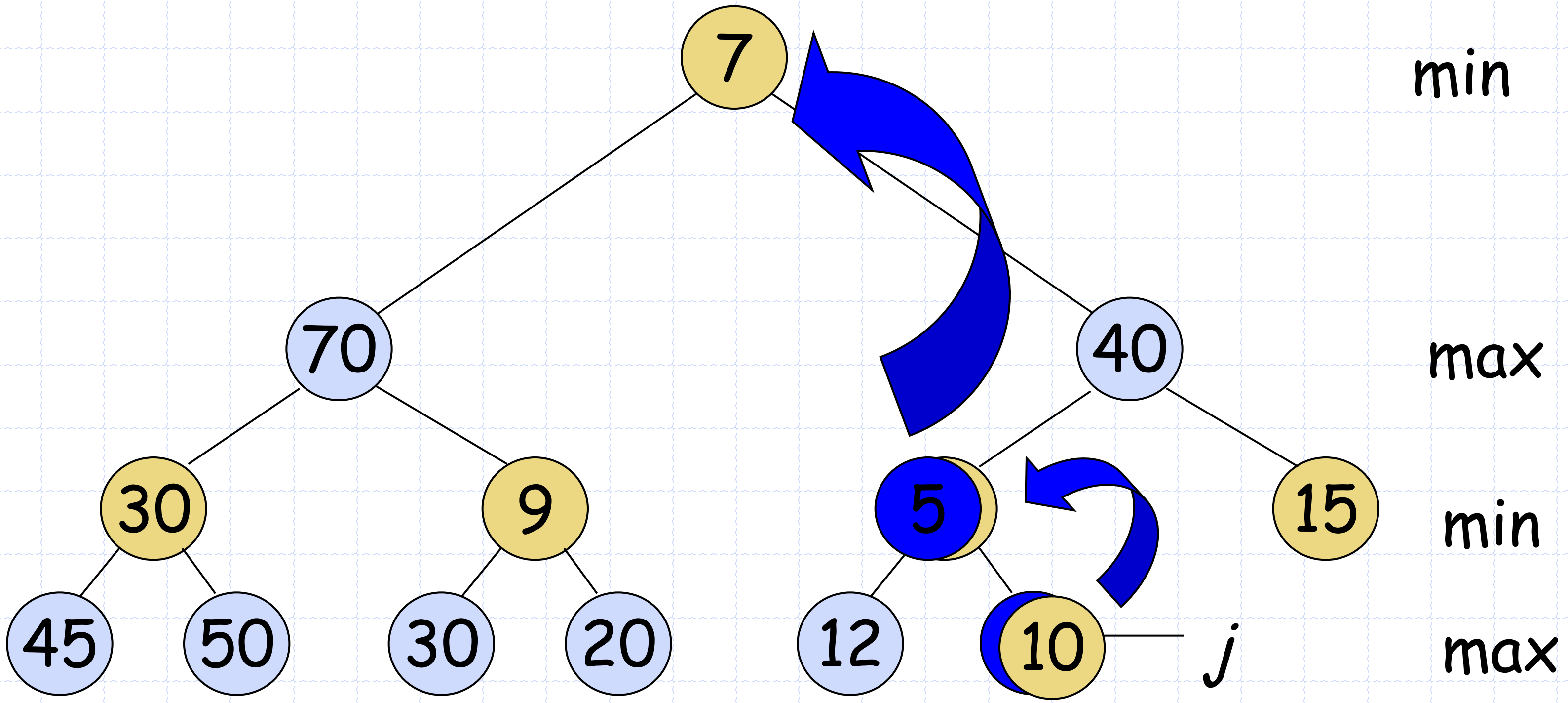
Algorithms

```
procedure TrickleDownMin(i)
 <* TrickleDownMax is the same, except
          that the relational operators are reversed    *>
    if A[i] has children then
          m := index of smallest of the children and grandchildren (if any) of
    A[i]
    if A[m] is a grandchild of A[i] then
          if A[m] < A[i] then
                    swap A[i] and A[m]
                    if A[m] > A[parent(m)] then
                              swap A[m] and A[parent(m)]
                    endif
                    TrickleDownMin(m)
          endif
    else <*  A[m] is a child of A[i]   *>
          if A[m] < A[i] then
                    swap A[i] and A[m]
          endif
    endif
```

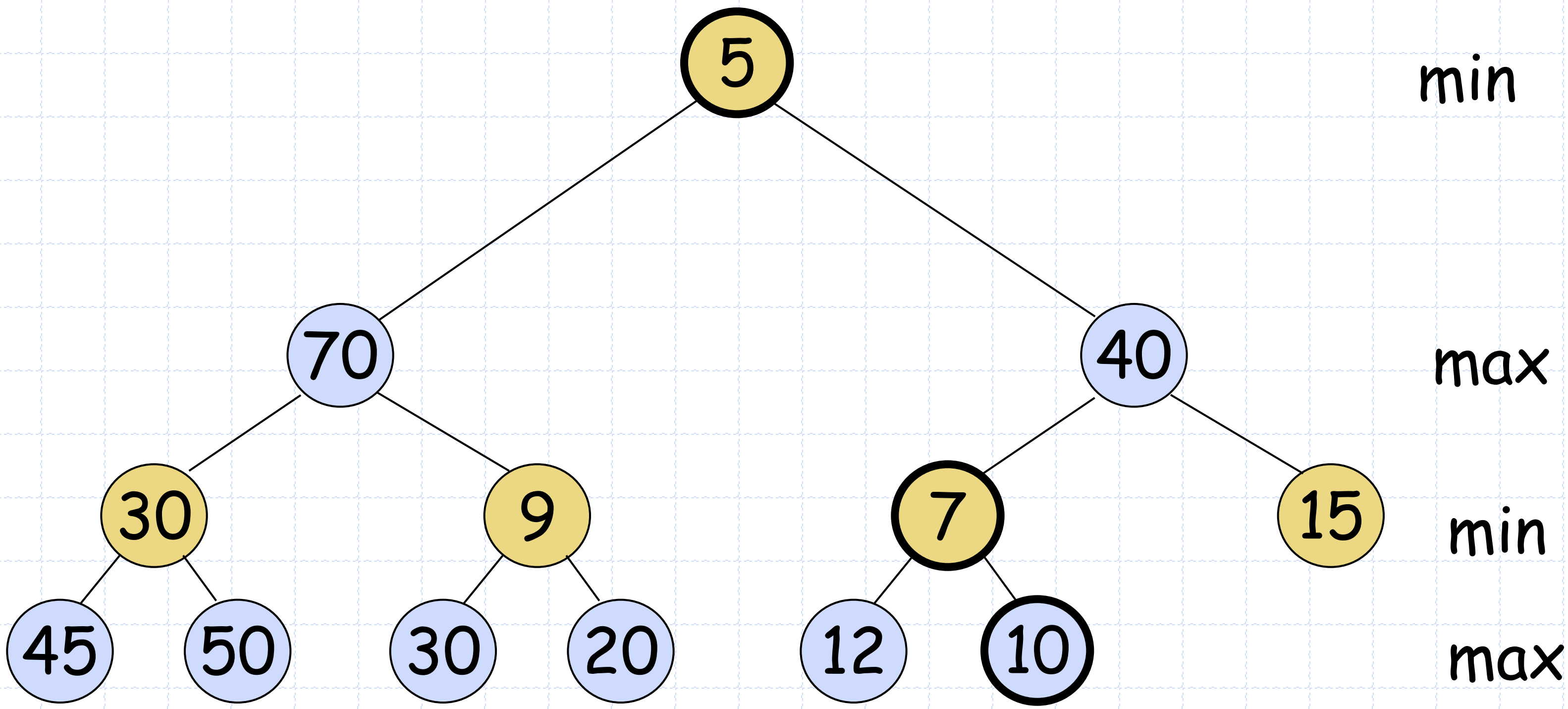
Inserting a New Key



Inserting a New Key

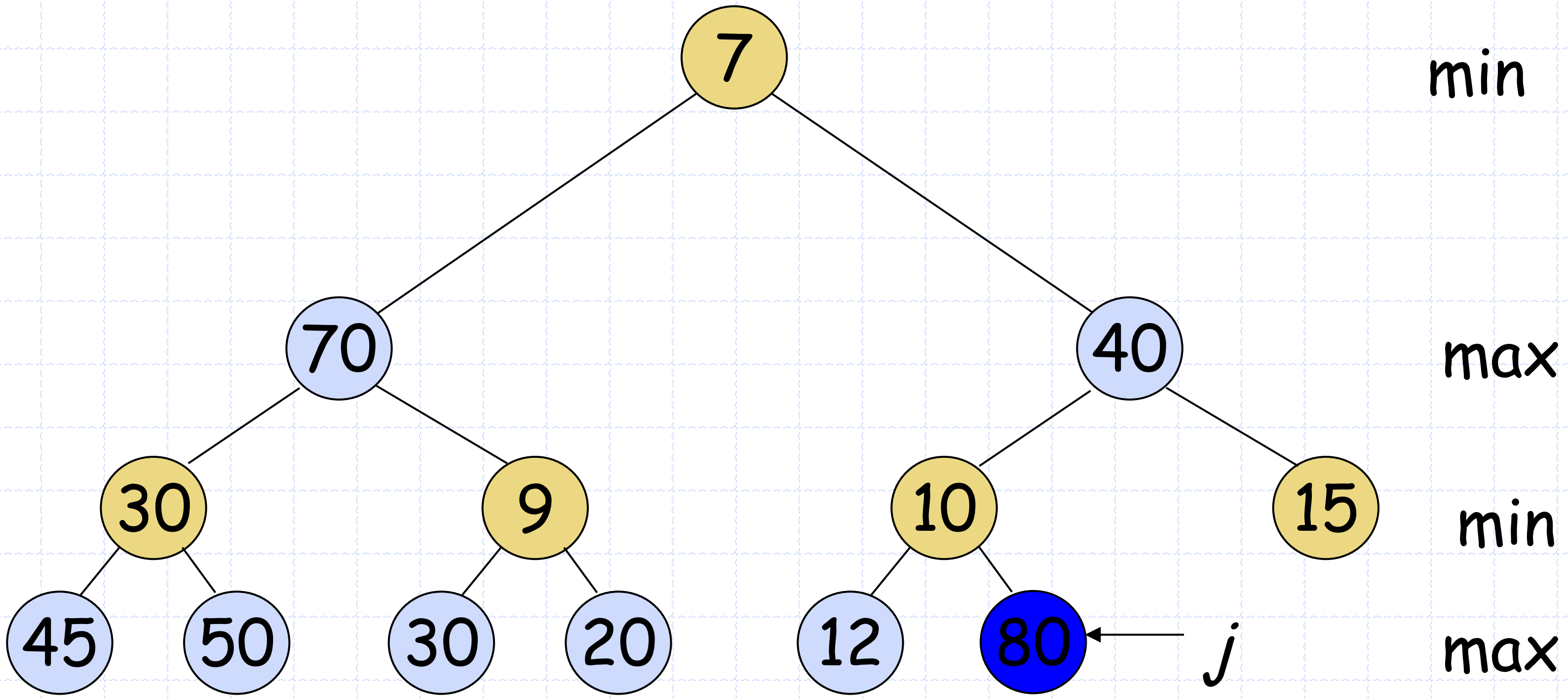


Insertion Key 5

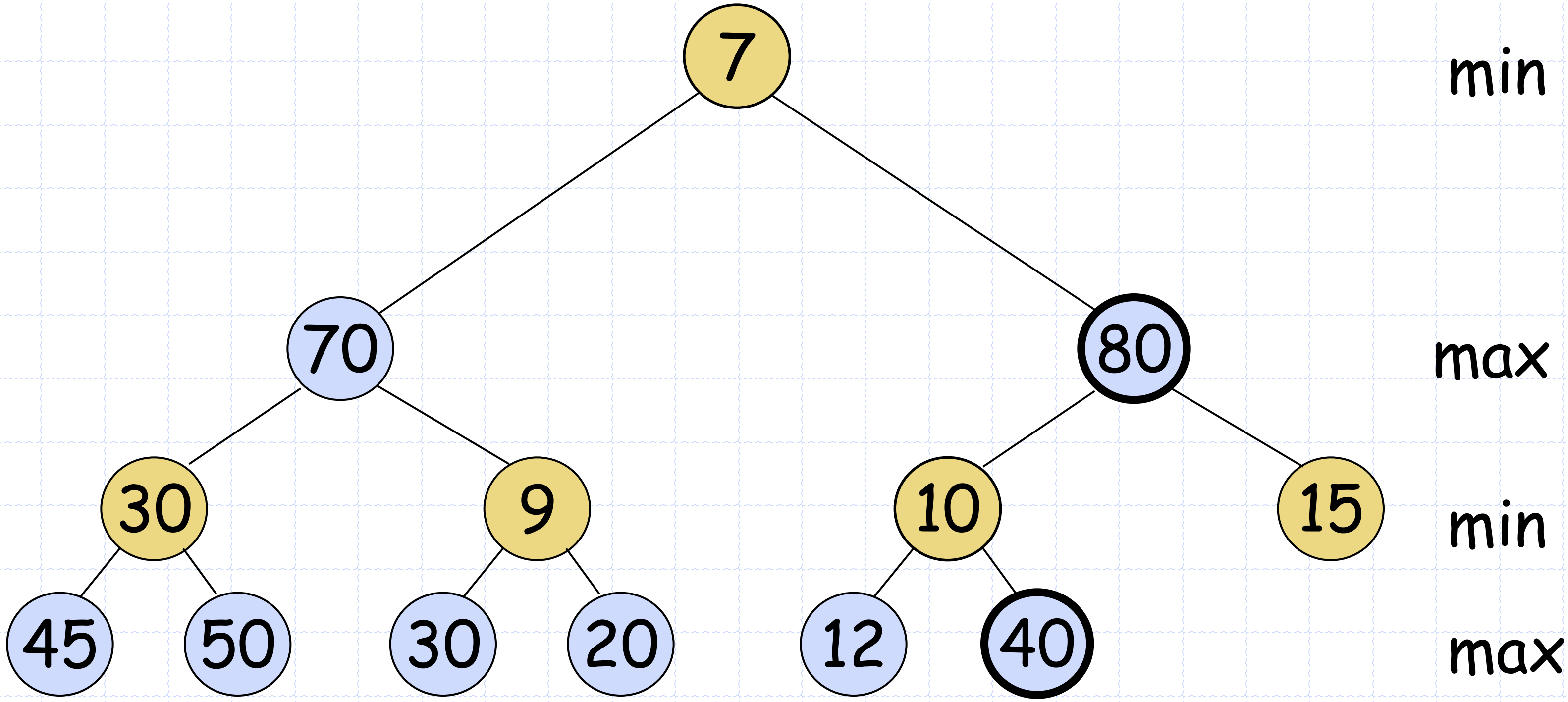


- Initially key 5 is inserted at j.
- Now since 5 < 10 (which is j's parent), 5 is guaranteed to be smaller than all keys in nodes that are on max levels on the path from j to root. Only need to check nodes on min levels.

Inserting a New Key



Insertion Key 80



- Since 80 > 10, and 10 is on the min level, we are assured that 80 is larger than all keys in the nodes that are both on min levels and on the path from j to the root. Only need to check nodes on max levels.

Algorithms (Cont'd)

```
procedure BubbleUp(i)
  <*  exchange the value at A[i] with the its parent   *>
  <*  i is the position in the array     *>
   if i is on a min-level then
        if i has a parent cand A[i] > A[parent(i)] then
     <*  cand: conditional AND     *>
                swap A[i] and A[parent(i)]
                BubbleUpMax(parent(i))
        else
                BubbleUpMin(i)
        endif
   else
        if i has a parent cand A[i] < A[parent(i)] then
                swap A[i] and A[parent(i)]
                BubbleUpMin(parent(i))
        else
                BubbleUpMax(i)
        endif
   endif
```

Algorithms (Cont'd)

```
procedure BubbleUpMin(i)
  if A[i] has grandparent then
      if A[i] < A[grandparent(i)] then
          swap A[i] and A[grandparent(i)]
          BubbleUpMin(grandparent(i))
      endif
  endif
```

Algorithms (Cont'd)

procedure BubbleUpMax(i)

if A[i] has grandparent **then**

if A[i] > A[grandparent(i)] **then**

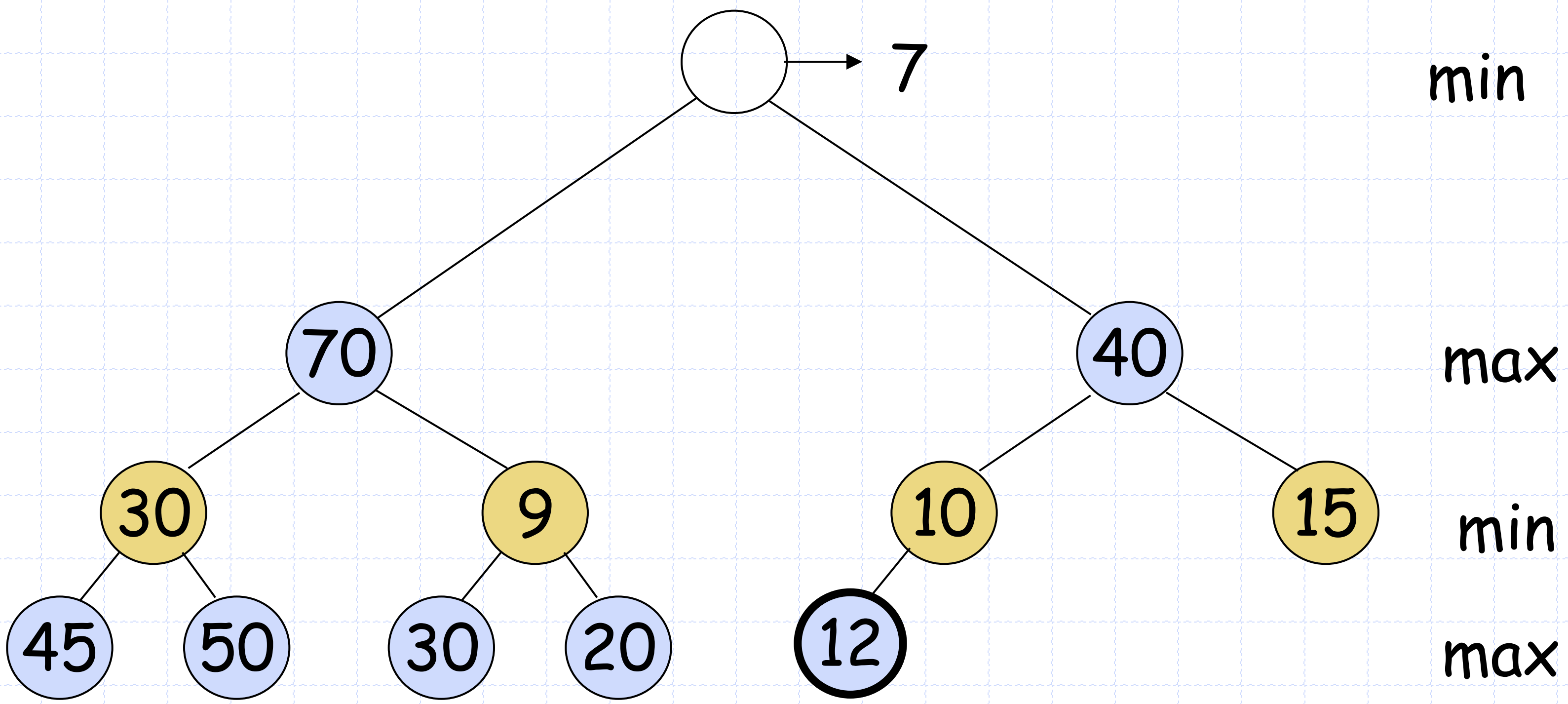
swap A[i] and A[grandparent(i)]

BubbleUpMax(grandparent(i))

endif

endif

Deletion of the Min Element

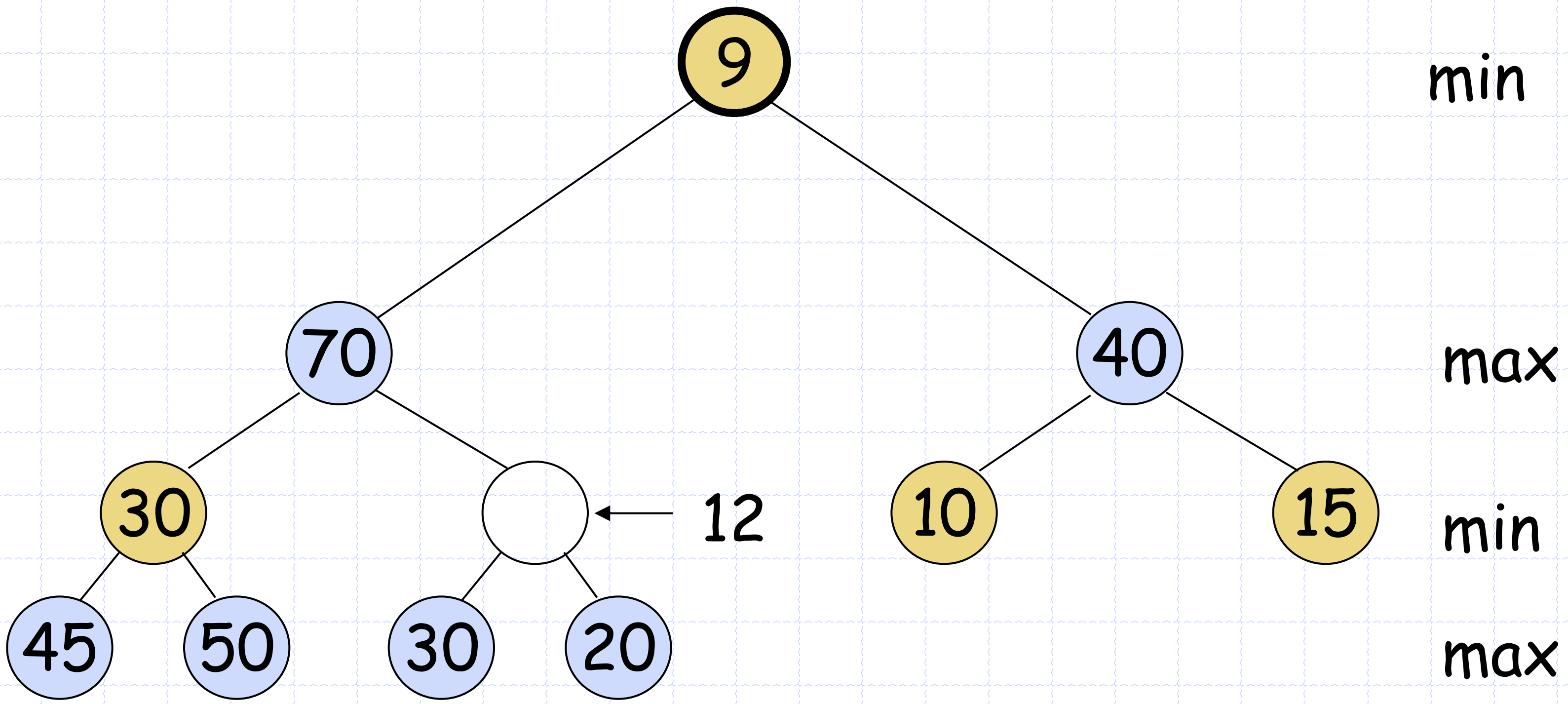


Min (max) deletion

- Analogous to deletion in conventional heaps
- Required element is extracted and vacant position is filled with the last element of the heap
- Min-max ordering is restored with *trickledown* procedure

Deletion of the Min Element

Min-Max Heap After Deleting Min Element



Complexity of the operations

- Identical to the same operations in the conventional heaps

References and Exercises

- Atkinson, M.D., J.R. Sack, N. Santoro, and T. Strothotte, Min-Max Heaps and Generalized Priority Queues, *Communications of the ACM*, October 1986, Volume 29, Number 10, pp. 996-1,000.

Questions ?