MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

26 Binary Search Trees and Priority Queues

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

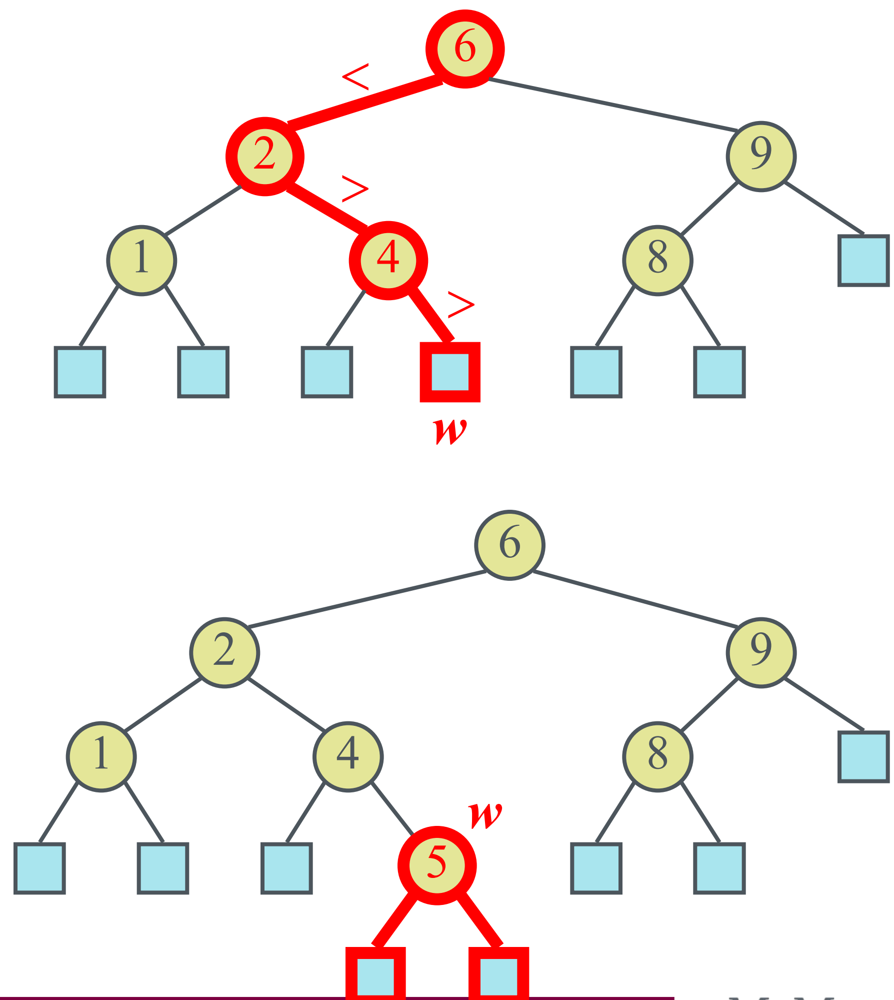
March 24, 2022

Admin

- Mid-Term 2:
  - Wednesday 30 March 2022
  - Duration: **1 hour**
  - **From 1:30 to 14:30 (lec. time)**
  - Location: ?
  - Covers: Topics from “Doubly Linked Lists” until the lecture of Wednesday 16 March 2022 (inclusive)

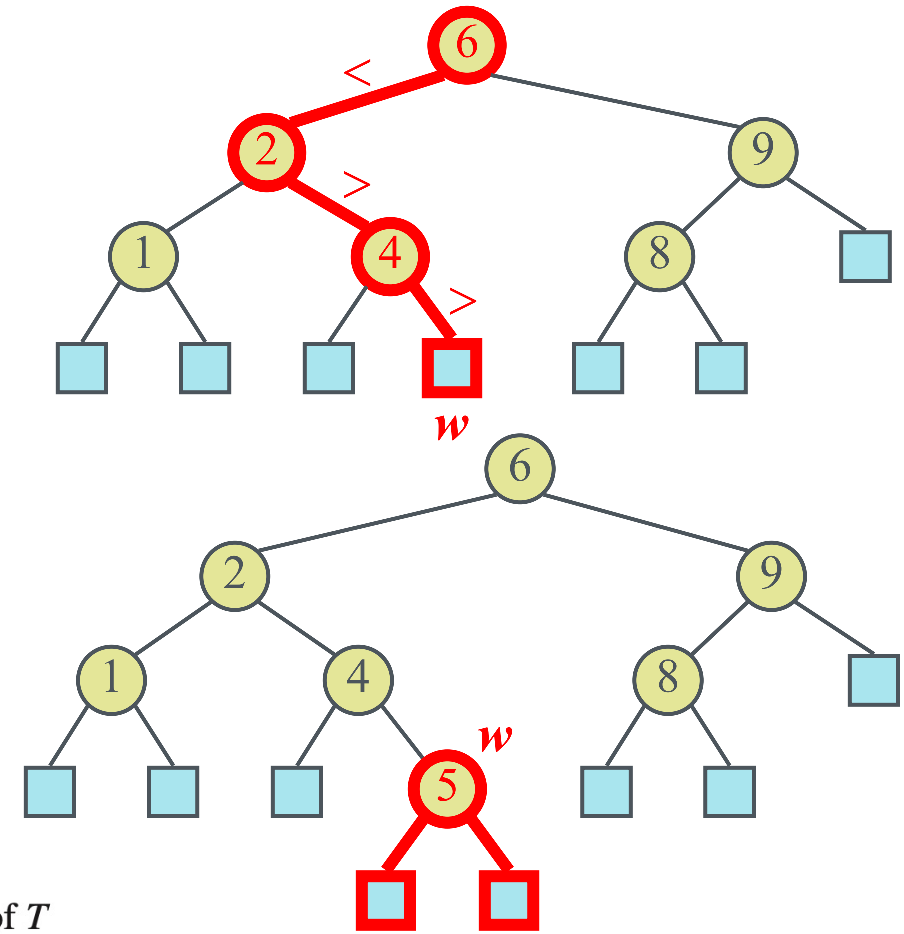
Binary Search Tree - Insert

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



Binary Search Tree - Insert

- **insertAtExternal(*v*,e)**: Insert the element **e** at the external node ***v***, and expand ***v*** to be internal, having new (empty) external node children; an error occurs if ***v*** is an internal node.
- The algorithm traces a path from **T**'s root to an external node

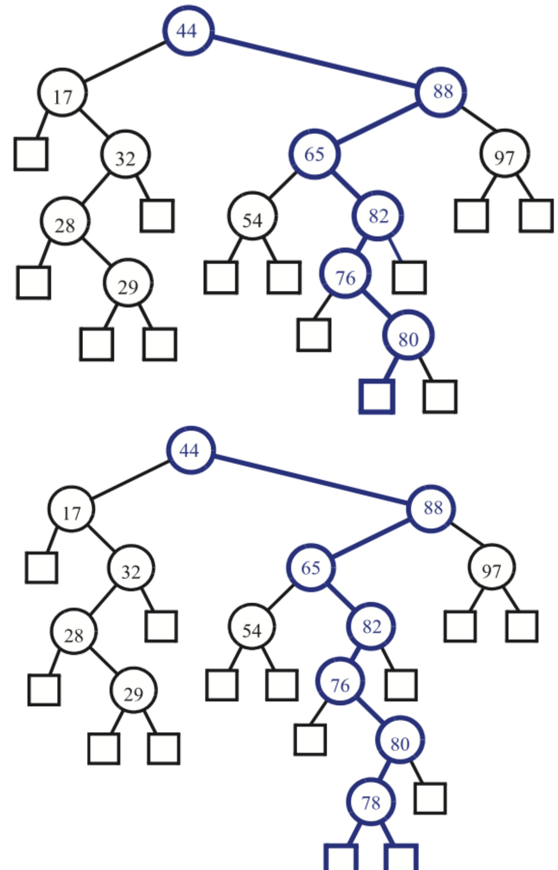


Algorithm TreeInsert(k, x, v):
    Input: A search key k, an associated value, x, and a node v of T
    Output: A new node w in the subtree $T(v)$ that stores the entry (k, x)
    $w \leftarrow$ TreeSearch(k, v)
    if T.isInternal(w) **then**
        return TreeInsert$(k, x, T.\text{left}(w))$ {going to the right would be correct too}
    T.insertAtExternal$(w, (k, x))$ {this is an appropriate place to put (k, x)}
    return w

Algorithm TreeSearch(k, v):
    if T.isExternal(v) **then**
        return v
    if $k <$ key(v) **then**
        return TreeSearch$(k, T.\text{left}(v))$
    else if $k >$ key(v) **then**
        return TreeSearch$(k, T.\text{right}(v))$
    return v {we know $k =$ key(v)}

Binary Search Tree - Insert (example)

- **insertAtExternal(*v*,*e*)**: Insert the element **e** at the external node ***v***, and expand ***v*** to be internal, having new (empty) external node children; an error occurs if ***v*** is an internal node.
- The algorithm traces a path from **T**'s root to an external node



(b)

Algorithm TreeInsert(k,x,v):

  Input: A search key k, an associated value, x, and a node v of T

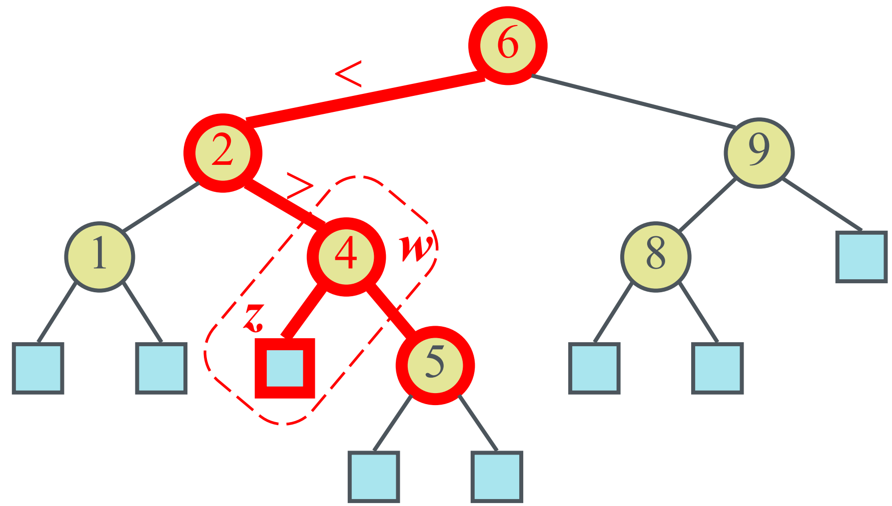
  Output: A new node w in the subtree $T(v)$ that stores the entry (k,x)

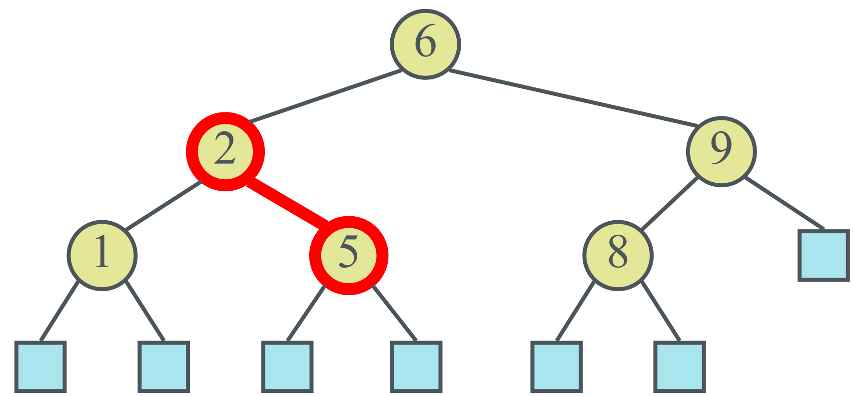
  $w \leftarrow$ TreeSearch(k,v)
  if T.isInternal(w) **then**
    return TreeInsert$(k,x,T.\text{left}(w))$ {going to the right would be correct too}
  T.insertAtExternal$(w,(k,x))$ {this is an appropriate place to put (k,x)}
  return w

Algorithm TreeSearch(k,v):
  if T.isExternal(v) **then**
    return v
  if $k <$ key(v) **then**
    return TreeSearch$(k,T.\text{left}(v))$
  else if $k >$ key(v) **then**
    return TreeSearch$(k,T.\text{right}(v))$
  return v {we know $k =$ key(v)}

Binary Search Tree - Deletion

- To perform operation **erase(k)**, we search for key **k**
- if **k** is not in tree => error!
- if key **k** is in the tree, and let **w** be the node storing **k**:
  - If node **w** has a leaf child **z**, we remove **z** and **w** from the tree with operation **removeAboveExternal**(**z**), which removes **z** and its parent
  - Example: remove 4
- **removeAboveExternal**(v): Remove an external node **v** and its parent, replacing **v**'s parent with **v**'s sibling; an error occurs if **v** is not external.

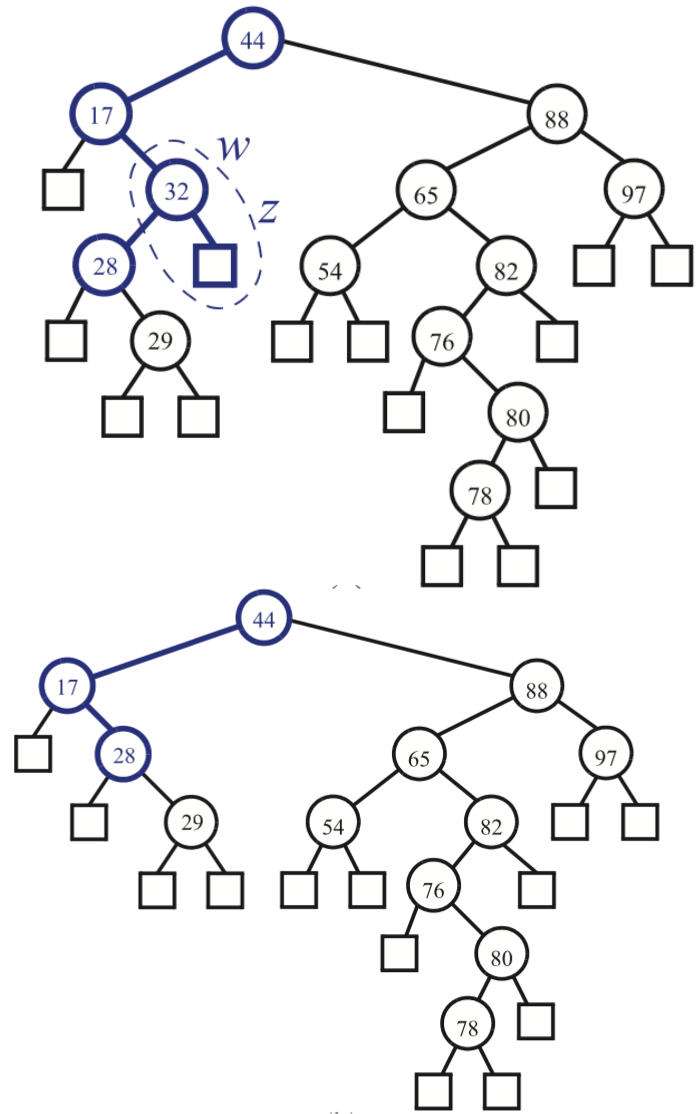




Case 1

Binary Search Tree - Deletion

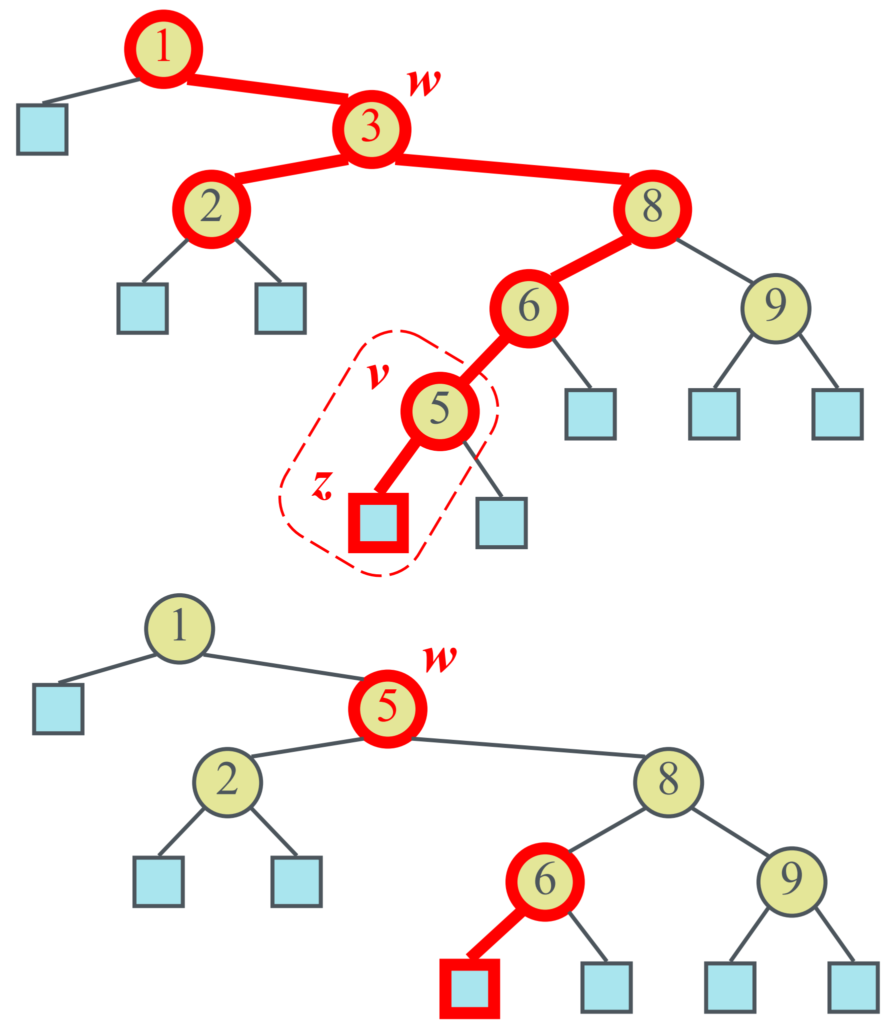
- To perform operation **erase(k)**, we search for key **k**
- if **k** is not in tree => error!
- if key **k** is in the tree, and let **w** be the node storing **k**:
  - If node **w** has a leaf child **z**, we remove **z** and **w** from the tree with operation **removeAboveExternal**(**z**), which removes **z** and its parent
  - Example: remove 4
- **removeAboveExternal**(v): Remove an external node **v** and its parent, <u>replacing</u> **v**'s parent with **v**'s sibling; an error occurs if **v** is not external.



Case 1

Binary Search Tree - Deletion

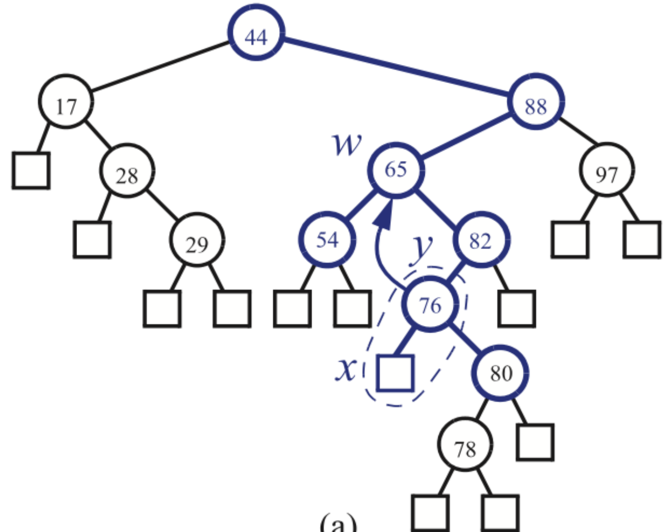
- We consider the case where the key **k** to be removed is stored at a node **w** whose children are both internal:
  - we find the internal node **v** that follows **w** in an inorder traversal. How?
  - we copy **key**(**v**) into node **w**
  - we remove node **v** and its left child **z** (which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



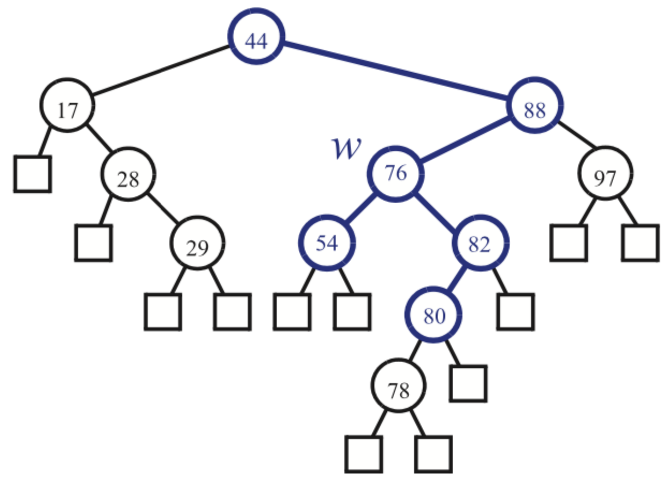
Case 2

Binary Search Tree - Deletion

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



(a)

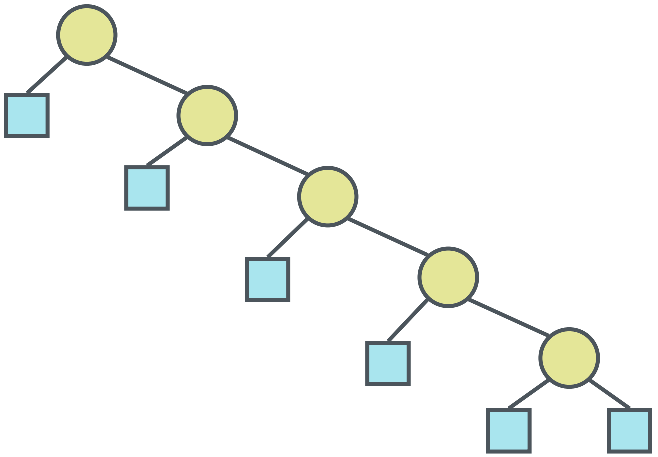


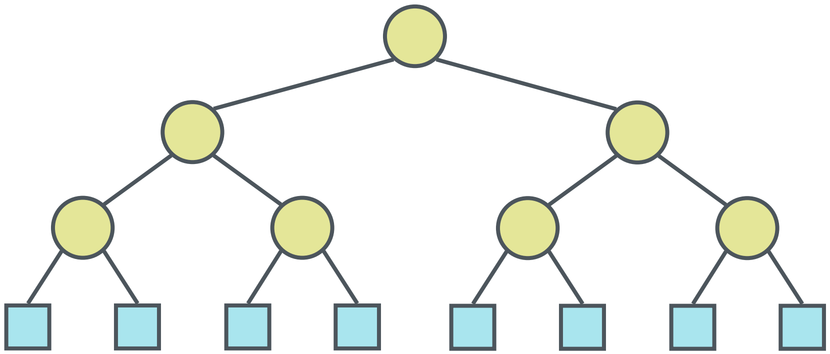
(b)

Case 2

Binary Search Tree - Performance

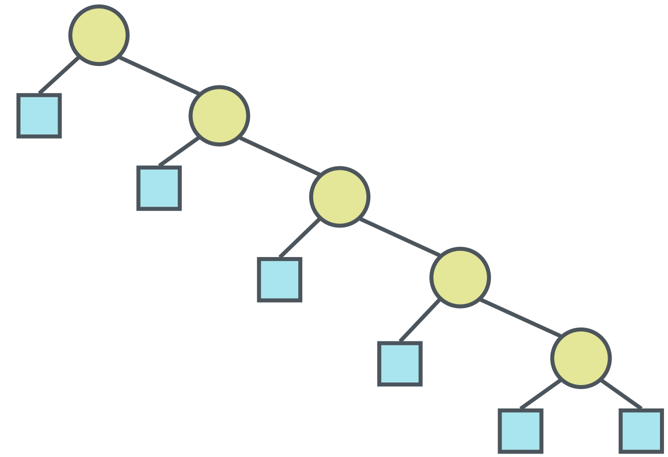
- For a Binary Search Tree of height **h**
  - the space used is O(n)
  - methods **search**, **insert** and **delete** take O(h) time
  - For **delete** of case 2 we need an **O(h)** time to locate the node, and an **O(h)** time to find the replacement => overall: **O(h)**
- The height **h** is
  - **O(n)** in the worst case
  - **O(log n)** in the best case: This usually happens
    - When insertions and deletions are made at random, the height is O(log n) on the average.

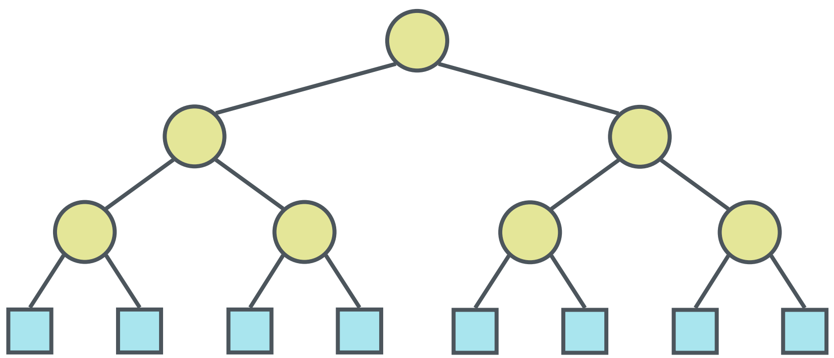




Binary Search Tree - Performance

- Search trees with a worst-case height of O(log n) are called balanced search tree.
- Balanced search trees permit each searching, insertion, or deletion can be performed in O(log n) time.
- AVL trees
- Red/black trees
- 2-3 trees
- 2-3-4 trees
- B trees
- B+ trees





Priority Queue

Priority Queue ADT

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

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key

- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Comparator ADT

- Implements the boolean function isLess(p,q), which tests whether p < q
- Can derive other relations from this:
  - (p == q) is equivalent to
  - (!isLess(p, q) && !isLess(q, p))
- Can implement in C++ by overloading “()”

Two ways to compare 2D points:

```
class LeftRight { // left-right comparator
public:
    bool operator()(const Point2D& p,
      const Point2D& q) const
    { return p.getX() < q.getX(); }
};
class BottomTop { // bottom-top
public:
    bool operator()(const Point2D& p,
    const Point2D& q) const
    { return p.getY() < q.getY(); }
};
```

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm

Input sequence S, comparator C for the elements of S

Output sequence S sorted in increasing order according to C

$P \leftarrow$ priority queue with comparator C

while $\neg S.empty()$

    $e \leftarrow S.front(); S.eraseFront()$

    $P.insert(e, \emptyset)$

while $\neg P.empty()$

    $e \leftarrow P.removeMin()$

    $S.insertBack(e)$

Sequence-based Priority Queue

- Implementation with an unsorted list

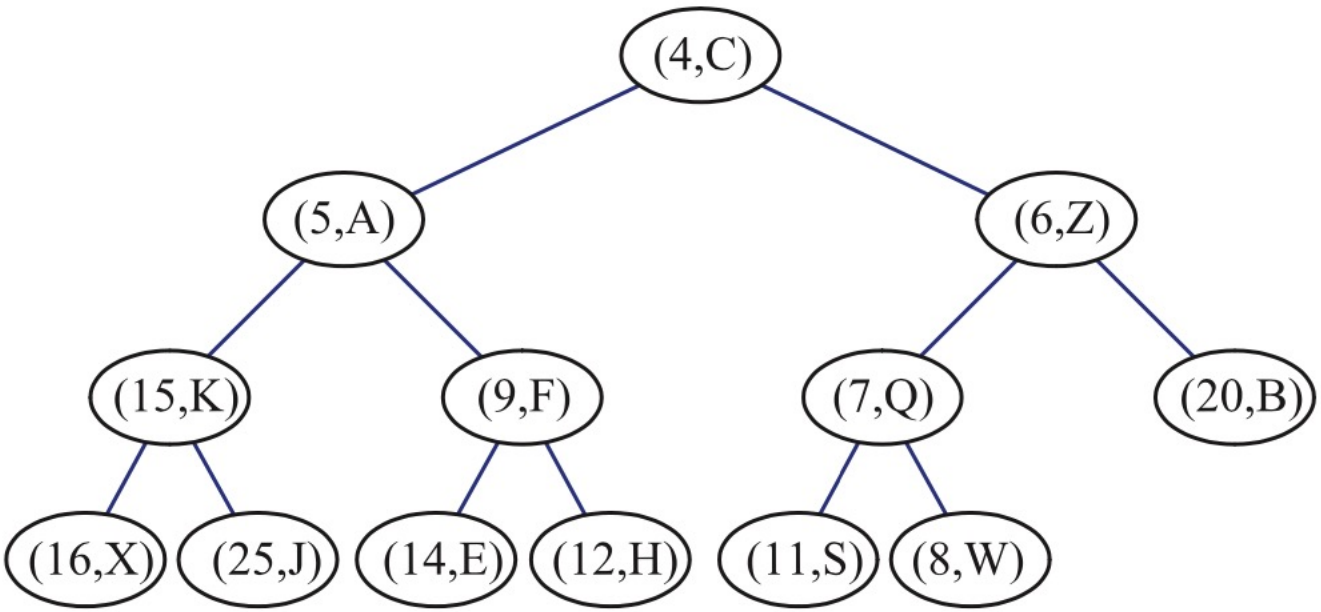
- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

Special case of a Priority Queue

- Heap
  - n a heap *T* , for every node *v* other than the root, the key associated with *v* is greater than or equal to the key associated with *v*'s parent.



Questions?