Fundamental Algorithms

Lecture #5

Cluj-Napoca 30.10.2019

Rodica Potolea, CS, UTCN

Agenda

- Hash Tables
- Trees
  - Binary and multiway
  - Representation
  - Basic operations

Hash Tables

- Stores a dynamic set of data for fast access = data whose content varies (ex symbol table in a compiler)
- Frequent operation = search
- DS that maintains a set of items (identified by a key) subject to following operations:
  - insert (item): add item to set
  - delete (item): remove item from set
  - search (key): return item with key if it exists
- goal: O(1) time per operation.

Hash Tables - direct access table

- Items stored in an array (hash table) indexed by key (identifier of item)

1	/
2	/
…	…
key1	item1
…	…
key2	item2
…	…

Limitations:

keys must be nonnegative integers

large key range = large storage space

Solution

Reduce universe *U* of all keys down to reasonable range for table

⇔ project *U* onto sa table of size *m*

Hash Tables

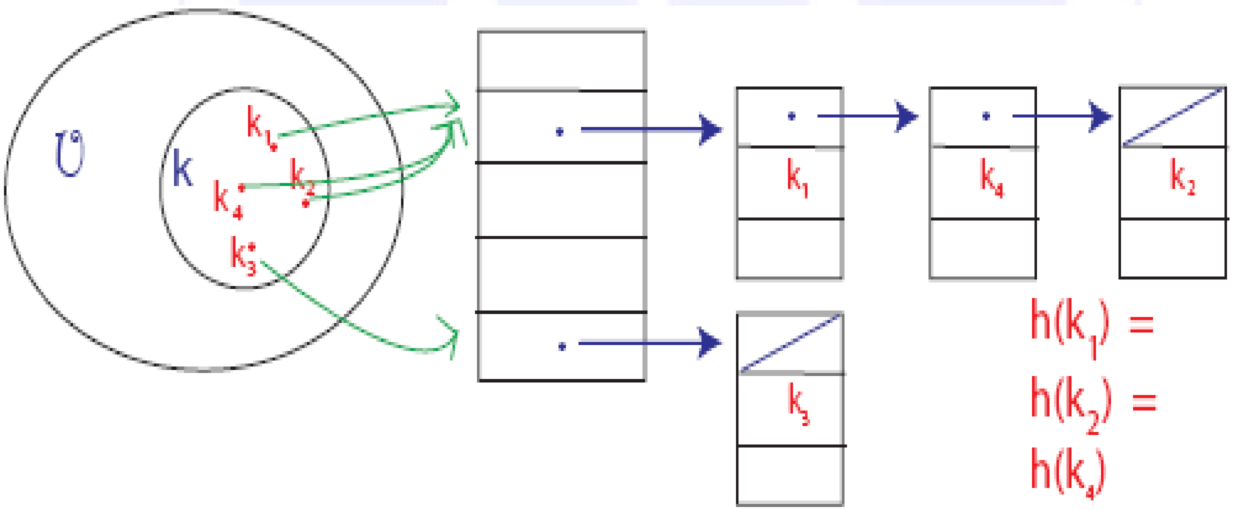
- m= table's size
- n=|U| (universe's size, number of possible keys)
- h:U->{0,1,...m-1} mapping. Properties?
- h calculates key's location in the table
- h(key1)=h(key2)
  - Is it possible?
    - Collision
  - What happens?
    - Deal with collision

Hash Tables – collision handling

- Chaining (linked lists) ~ find alternative solution
  - Hash functions
  - Universal hash
- Open addressing (in table) ~ find alternative place
  - Linear probing
  - Double hashing

Chaining (linked lists)

- Linked list of colliding elements in each slot of table



Picture taken from MIT OpenCourseWare, Introduction to Algorithms http://ocw.mit.edu6.006Introduction

Chaining (linked lists)

- $h(k)$=Dispersion value
- Search must go through the whole list
- Load factor: $\alpha=n/m$ =average number of keys per slot ($n=|K|$, K= key set, $m=|T|$, T=table)
- Expected performance of chaining assuming *simple uniform hashing* $O(1+\alpha) \Rightarrow O(1)$ iff $\alpha=1$, i.e. n=m!
- Worst case: all keys in K to the same slot $\Rightarrow O(n)$

iff =
if and only

Hash functions

- **Division Method**: $h(k) = k \bmod m$
- k_1 and k_2 collide when $k_1 = k_2 (\bmod m)$ i.e. when m divides $|k_1 - k_2|$
- OK if keys are randomly distributed
- Not OK if they are on a *pattern* distribution
- **Good Practice**: to *avoid* common regularities in keys make *m a prime number* that is not close to a power of 2 or 10.
- Drawbacks:
  - find prime numbers (finding small/reasonable prime values is not a problem – just take them from tables),
  - division is slow

Hash functions - cont

- **Multiplication Method**:

  h(k) =m{kA} fractional part from kA

  =m(kA-[kA]), KA – integer part of KA

  where 0<A<1, A=ct.

- **Good practice**:
  - considering w= the length of the word of the machine,
  - m = 2^p for some int p so that m fits a single word.
  - Thus, h(k) easy to calculate (check the textbook for justification).
- Knuth shown **A=(sqrt5-1)/2** it's good value
- malicious keys => all keys in the SAME location! => O(n) => any possible hash function is vulnerable

Universal hash

- **Random** select the hash function at the execution time, from a set of functions

Note: again, randomness helps efficiency

- **Theorem**: if $n \leq m$, the average number of collisions per key <1 if a class of Universal hash functions is considered
- Hw: Check the textbook for **a class of Universal hash functions**

Open addressing

- All keys kept in the table (no linked lists),

1 key/slot => $m \geq n$

- The hash function specifies the order of slots to try for a key, not just one slot
- Sequence to try for a key k:

<h(k,0), h(k,1),…, h(k,m-1)>

- The sequence should be a permutation

of <0,1,…,m-1>

/
$item_3$
$item_1$
…
$item_2$
…

Probing Strategies

- **Linear probing**

$h(k, i)=(h'(k)+i) \bmod m$, where $h'(k)$ is ordinary hash function, $i=0,1,\ldots,m-1$

- Drawback: *clustering =* consecutive group of filled slots grows=>average search time grows
- **Double hashing**

$h(k, i)=(h'(k)+ih''(k)) \bmod m$, where $h'(k)$ and $h''(k)$ are ordinary hash functions

$h''(k)$ should be relatively prime to m

Open addressing - eval

- $\alpha < n/m < 1$, α load factor
- **Theorem**

  average **un**successful search time is $1/(1- \alpha)$
- **Theorem**

  average successful search time is $1/\alpha \ln(1/(1- \alpha))$

Trees

- Dynamic structures
- Target
  - Faster (than on linked lists) retrieval of elements
  - Maintain good running time for other operations
- Basic operations (n=#nodes in T; h=height of T)
  - Traversal — O(n)
    - (pre, in, post) order
  - Search — O(n) regular; O(h) BST
  - Insert — O(h)
  - Remove — O(n) regular; O(h) BST

  $h \in [\lg n, n]$ Why? Best? Worst?
  - Find: min, max, pred, succ — in BST only O(h)

Trees (binary) - representation

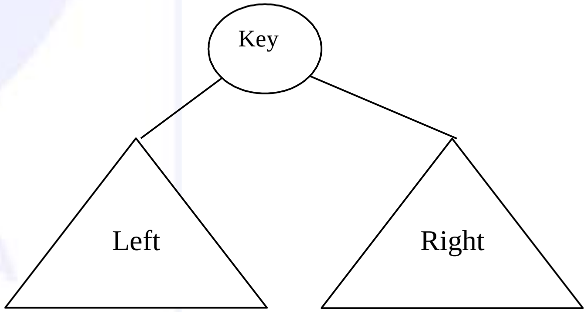
- Dynamic linked structures
- Minimal data representation:
  - key, left, right
  - parent, info
  - other (like balance, size, … depending on type)
- Empty tree = nil
- Types of nodes:
  - root (just one in a tree),
  - internal (non root, non leaf)
  - leaves (all nodes with both children nil)

Trees – representation - cont

- Multiway trees
  - A node has more than just 2 children (unspecified; unknown; variable)
  - Represented as:
    - a tree with just one child (linked list)
    - a binary tree:
      - left link= first child (proper tree link)
      - right link = brother (next child of the parent's node; right links form a singly linked list of brothers)
- Transformation?
- Ex? See blackboard

Binary Search Trees (BST)

- Binary Trees – if no order imposed on keys, NO improvement over lists! Why to have them?
- BST=BT with a **total order relation** defined on the key's set.
- $\forall x \in Left, \forall y \in Right, x \leq Key \leq y$
- Any subtree of a BST is a BST
- In general, the properties of a structure with recurrent definition are shared by the component structures (subtrees in our case)



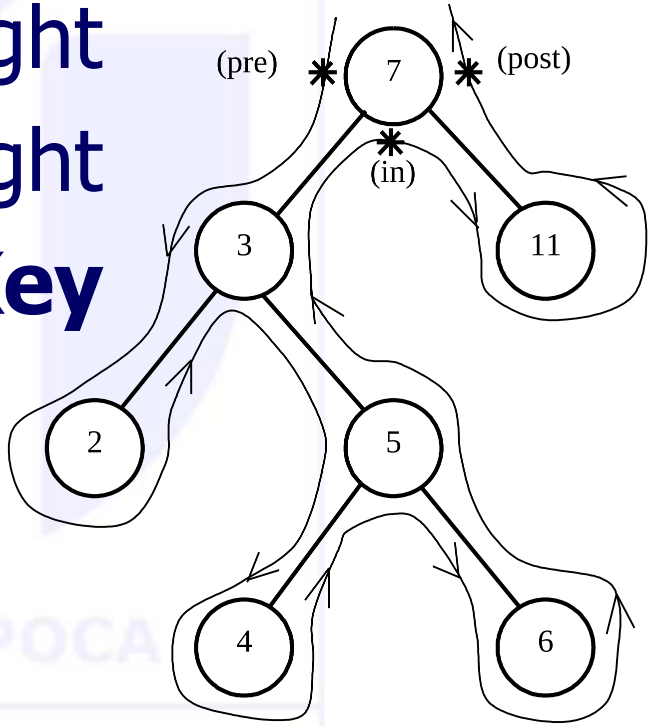
BST traversal

- Preorder: **Key**, Left, Right
- Inorder: Left, **Key**, Right
- Postorder: Left, Right, **Key**

(pre)=> **7** <u>3 2 5 4 6</u> 11

(in) => <u>2 3 4 5 6</u> **7** 11

(post)=> <u>2 4 6 5 3</u> 11 **7**



Root boldface

<u>Left</u> underlined

Inorder: keys are in nondecreasing order

BST traversal - code

```
tree_walk(x, order)  //x=root; order=in, pre,post
if x<>nil
  then      if order= pre
                 then write key[x]
            tree_walk(left[x],order)
            if order= in
                 then write key[x]
            tree_walk(right[x],order)
            if order= post
                 then write key[x]
```

Note: Just ONE `write` statement is executed (one color)

Look for the nonrecursive implementation!!!

BST traversal - eval

- order $\in$ {in, pre, post}
- Only one of the 3 statements **`write key[x]`** is executed
- O(n) (assuming constant time for the operation(s) performed at the level of each individual node – write in our case)

BST – search -recursive

```
r_tree_search(x,k)      //x=root; k=searched
if x=nil or k=key[x]
    then return(x)
    else if k<key[x]
        then r_tree_search(left[x],k)
        else r_tree_search(right[x],k)
```

- Running time: O(h)
- In a BST, $h \in [\lg n, n]$
- Discussion on recursive vs iterative implementation
- Recursive implementation: where to place the conditional statement (`if`) & why

BST – search -iterative

i_tree_search(x,k) //x=root; k=searched

```
while x<>nil and k<>key[x]
do
  if k<key[x]
     then x<-left[x]
     else x<-right[x]
return x
```

How does the time differ between iterative vs recursive implementation?

Same efficiency (big Oh), smaller machine time for iterative version (reason: overhead with stack)

BST – insert

Always as a leaf, regardless the particularity of the BT!!!!

NEVER EVER internal node. There is NO exception!

Running time: O(h).

Range of h LARGE for regular trees

Rooted tree = tree as a DS, root[T] its root

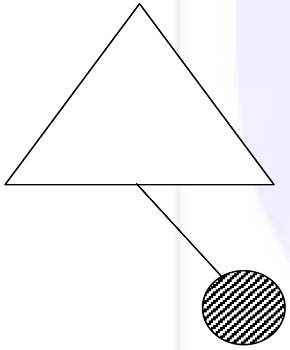
BST – insert - code

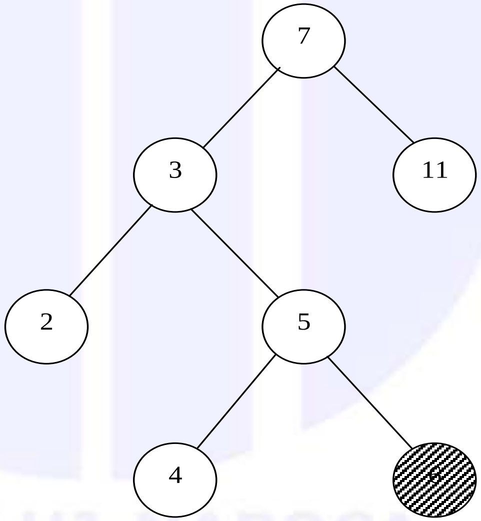
```
tree_insert(T,z)     //x=root; z=new node, already allocated
y<-nil               //y=x's parent; stays behind x;
x<-root[T]
while x<>nil         //search loop to find the position to insert
   do  y<-x          // y=x at the prev step
   if key[z]<key[x]
         then x<-left[x]
         else x<-right[x]
p[z]<-y         //position found; x=nil; y=new node (z)'s parent
if y=nil        //in case the tree was empty before this insertion
         then root[T]<-z
         else   if key[z]<key[y]
                      then left[y]<-z
                      else right[y]<-z
```

BST - delete

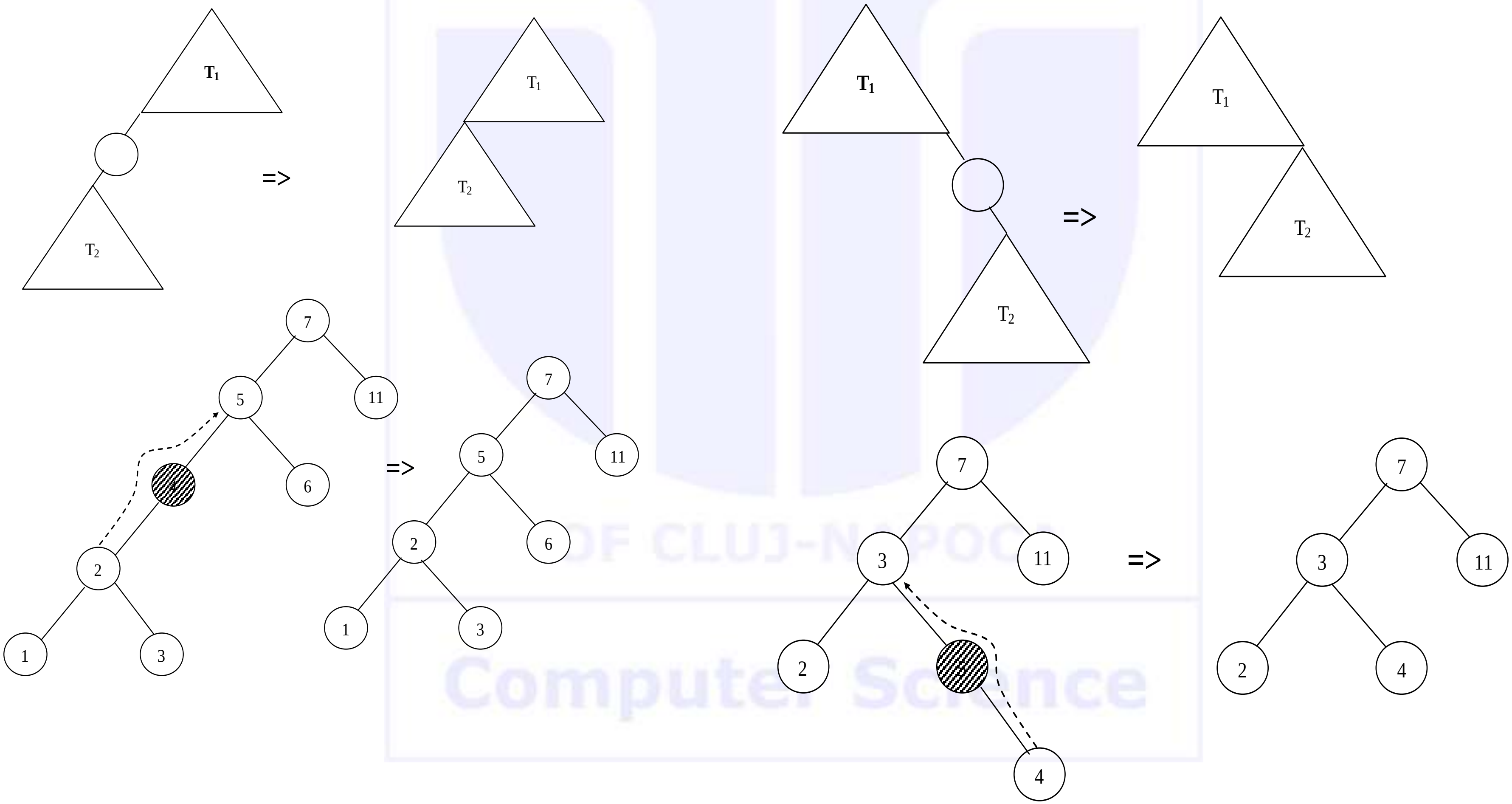
- Find the node
- Remove the node
- Cases:
  - Case 1: Leaf – remove it
  - Case 2: 1-successor node – skip it (its only child will become its parent’s child)
  - Case 3: 2-successors nodes!
    - Chain the tree (fast, unbalances the tree)
    - Replace the operation with an easier one:
      - Keep the structure= keep the node,
      - place a different (appropriate) content,
      - remove of the node with the given content

BST – delete – ex: leaf

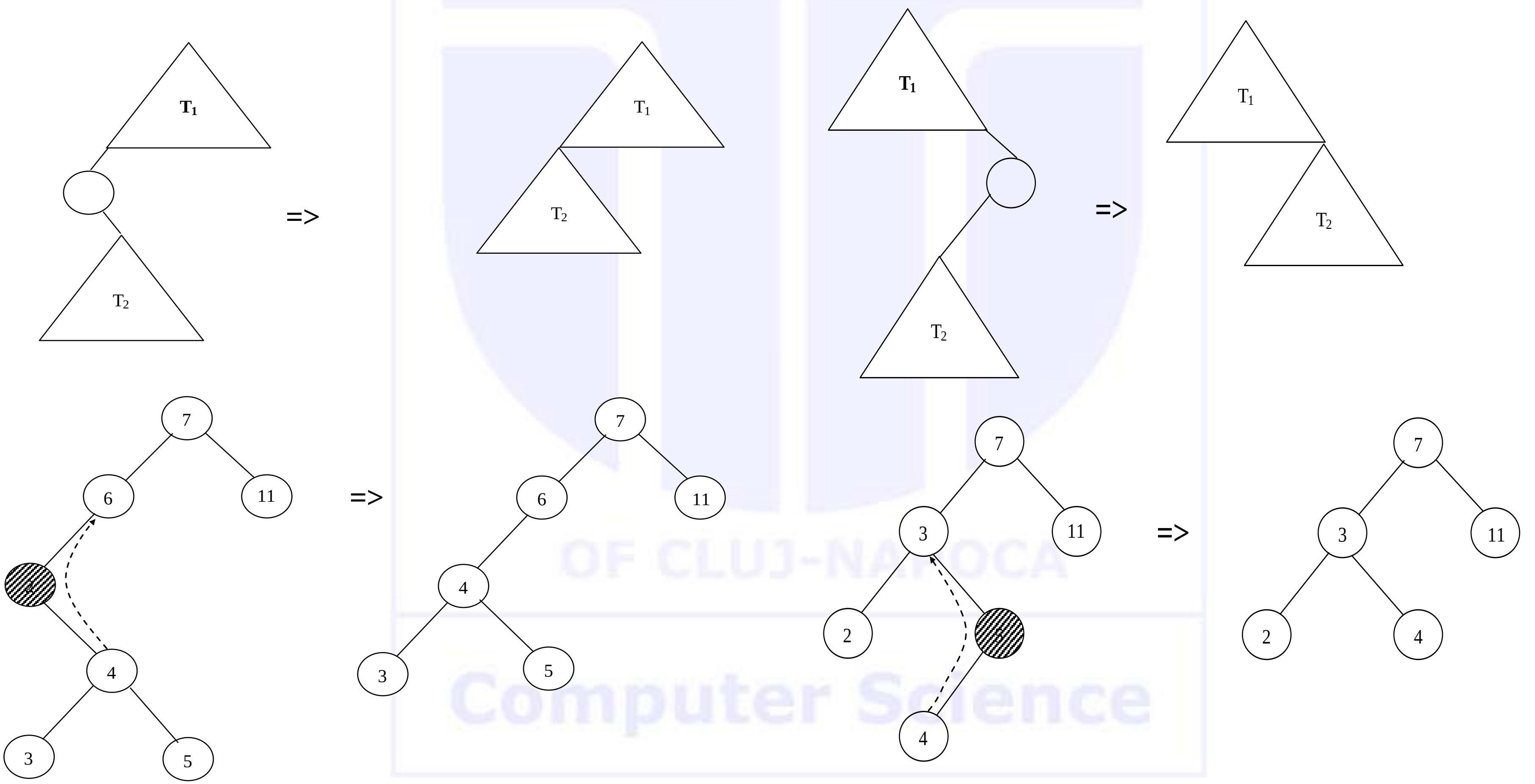




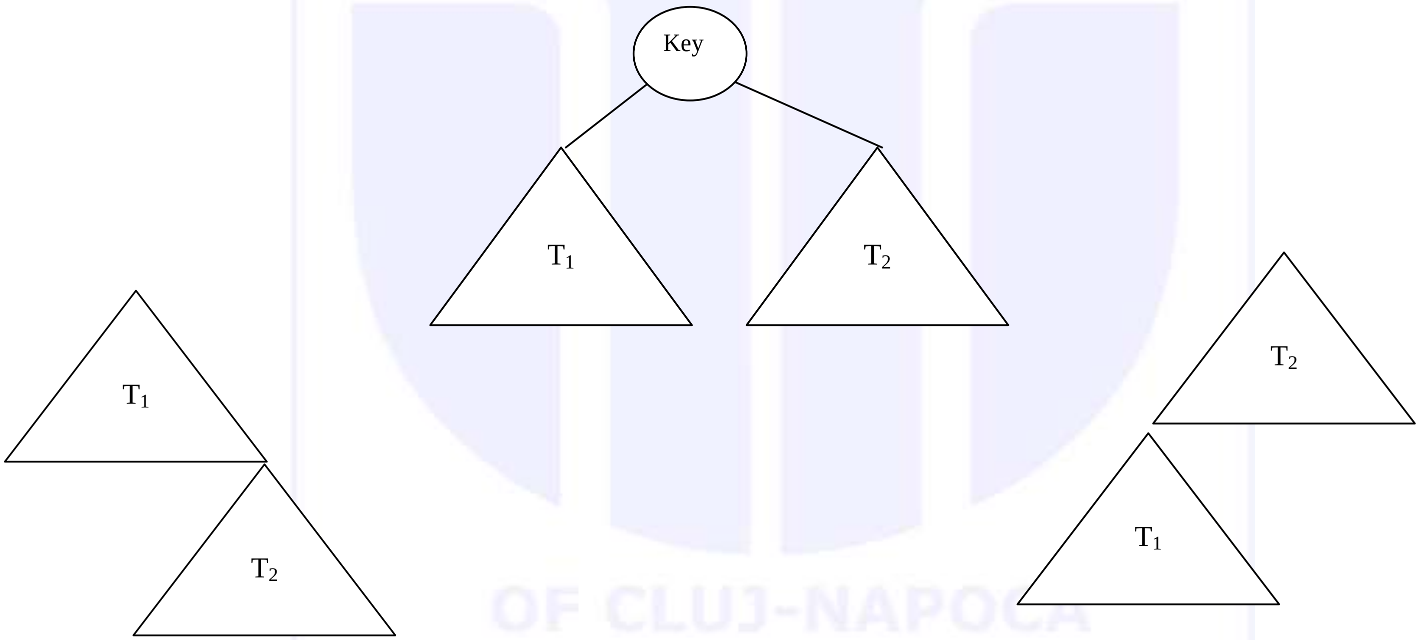
BST – delete – ex: a single successor node



BST – delete – ex: a single successor node



BST – delete – ex: root



Advantage: fast. How fast?

Drawback: increases the height

BST – delete - root

- Replace it with an easy-to-delete node
- Node to replace: *previous* or *next*. Why?
- Left<Key<Right
- Left'<*prev*(Key)<Key<*next*(Key)<Right
- Is it *prev*/*next* easier to del? Why?
- *prev*=max in Left=>no successor to the right
- *next*=min in Right=>no successor to the left
- =>both are nodes with at most 1 child (easy to del)

BST – delete - code

```
tree_delete(T,z)                    //z=node to delete; y physically deleted
if left[z]=nil or right[z]=nil
      then y<-z                     //Case 1 OR 2; z has at most 1 child => del z
      else y<-tree_successor(z)  //find replacement=min(right)
if left[y]<>nil                     //we are in Case 2; y is a single child node
      then x<-left[y]               //y has no child to the right; x=y's child
      else  x<-right[y]                   //case 2 or 3. Why?
if x<>nil                           //y is not a leaf;
      then p[x]<-p[y]               // y's child redirected to y's parent = x's parent
  //becomes the former single (why?) grandparent
if p[y]=nil                         //means y were the root
      then root[T]<-x               //y's child becomes the new root
      else  if y=left[p[y]]  //link y's parent to x which becomes its child
                  then left[p[y]]<-x
                  else right[p[y]]<-x
return[y]         //outside the procedure: copy y's info into z; dealloc y
```

BST – delete - eval

- Find node to delete O(h)
- Find successor O(h)
- BUT:
  - if finding node to delete takes O(h) =>case 1 => leaf (no succ needed)
  - if node to delete not a leaf=> case 2 or 3 => succ searched from that place down => find node + find succ=O(h)
- Delete takes O(h) overall