Data Structures

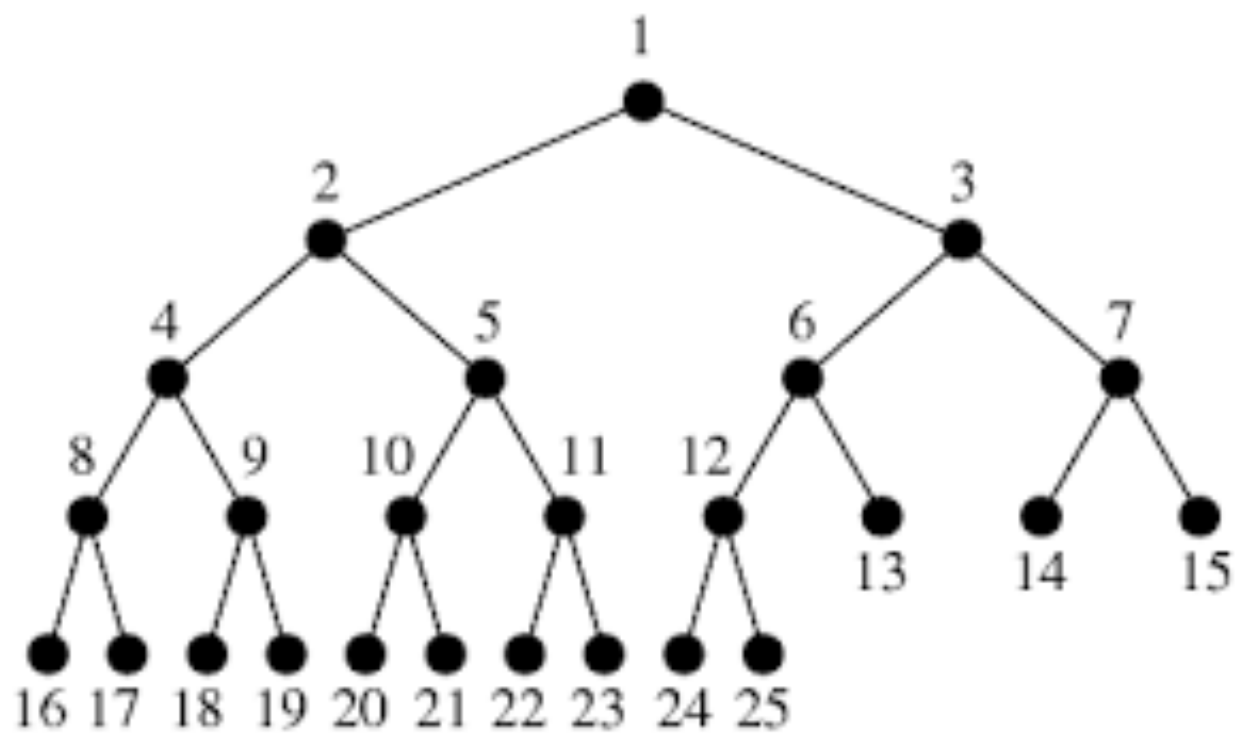
Lecture 19: Binary Search Trees

Nopadon Juneam
Department of Computer Science
Kasetsart university

Outlines

- Basics of binary search trees
    - Binary-search-tree property
    - Traversals of binary search trees
- Common operations on binary search trees

Binary Trees



- A ***binary tree*** is kind of an ordered tree in which every node has at most two children. However, if a node has just one child, the position of the child matters.

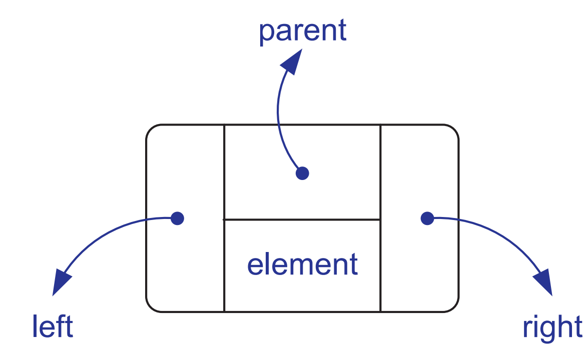
Linked Structure for Binary Trees

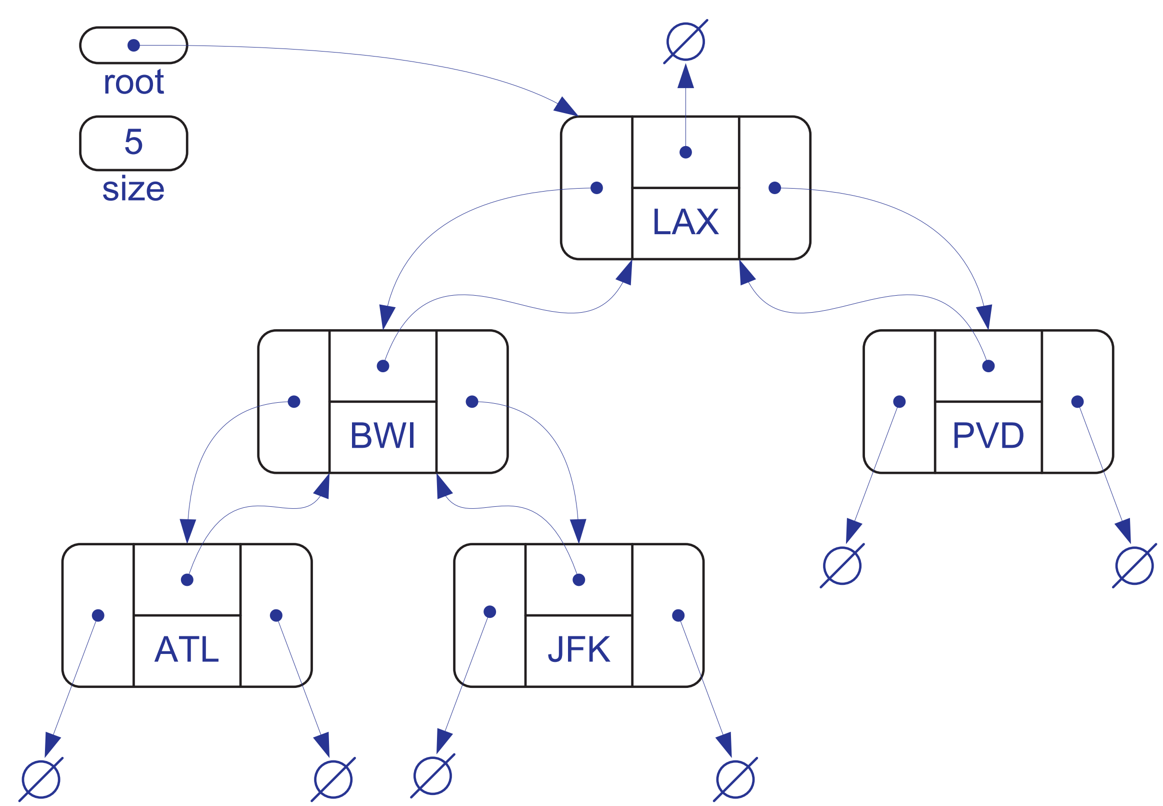
- In a linked structure for a binary tree *T*, we represent each node of *T* by a node object *p* with the following fields:

  - A reference to the node's element (key)

  - A link to the node's parent

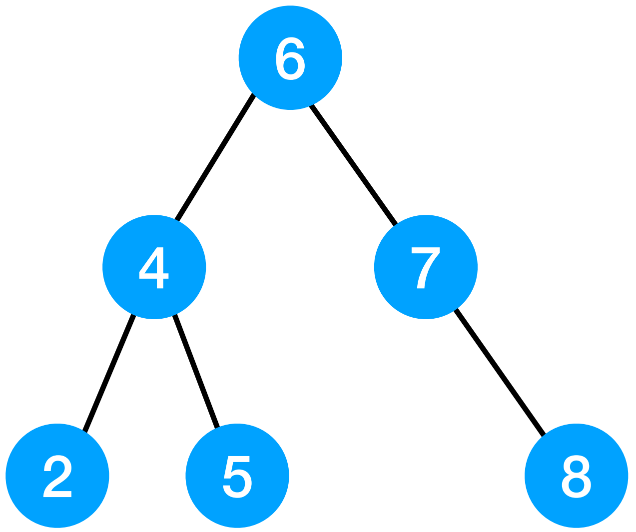
  - A link to the node's two children



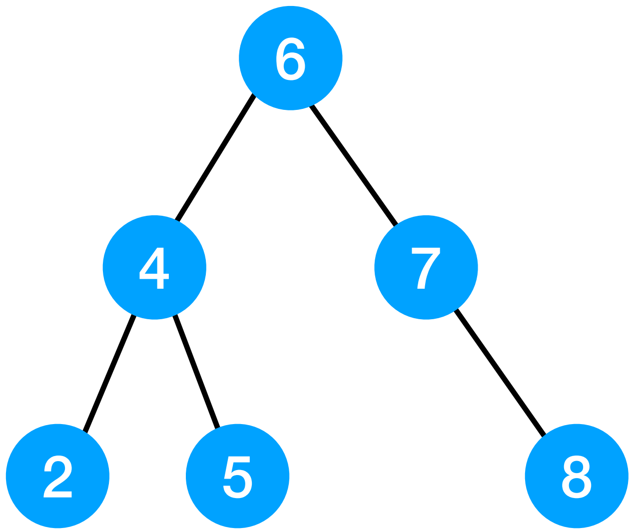


Binary Search Trees

- A *binary search tree* is a data structure organized, as the name suggests, in a *binary tree*
- Let *S* be a set whose elements have an order relation. For example, *S* could be a set of integers. A ***binary search tree*** for *S* is a *proper binary tree T* such that:
    - Each node *x* of *T* stores an element of *S*, denoted with x.*key*
    - For each internal node *x* of *T*, the elements stored in the left subtree of *x* are less than or equal to *x.key* and the elements stored in the right subtree of *x* are greater than to *x.key*



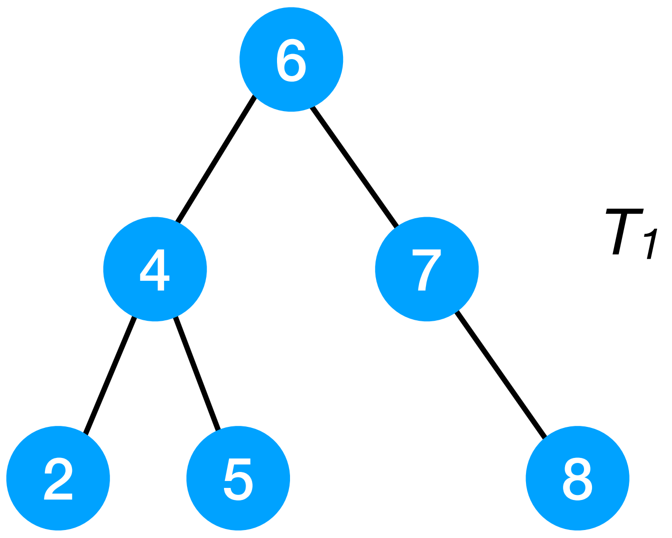
Binary-Search-Tree Property

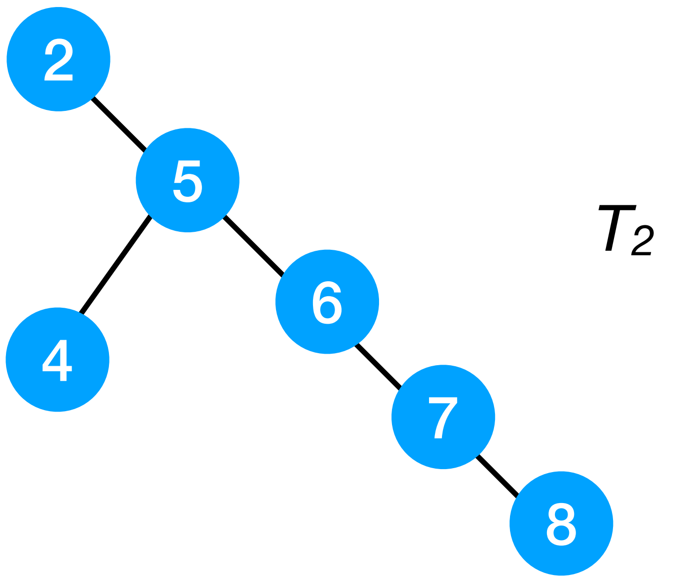


- The keys (elements) in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:
  - "*Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key* $\leq$ *x.key. If y is a node in the right subtree of x, then y.key > x.key.*"

Examples of Binary Search Trees

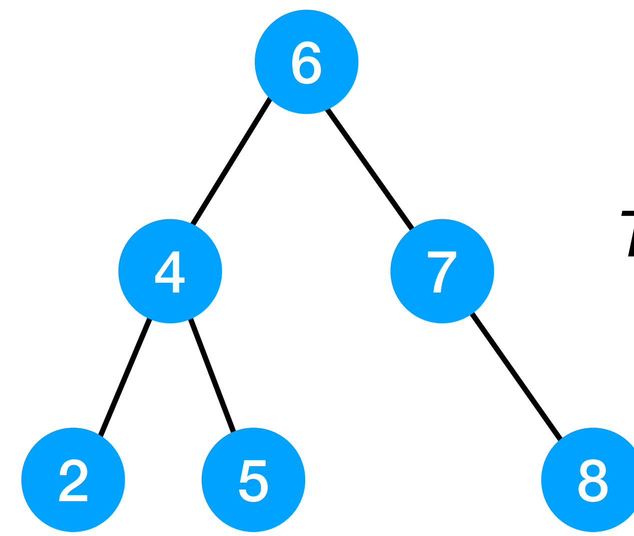
- For any node *x*, the keys in the left subtree of *x* are at most x.key, and the keys in the right subtree of *x* are greater than *x.key*
- Different binary search trees can represent the same set of values
- The worst-case running time for most search-tree operations is proportional to the height of the tree.
  - T_1 is a binary search tree on 6 nodes with height 2
  - T_2 is a less efficient binary search tree with height 4 that contains the same keys



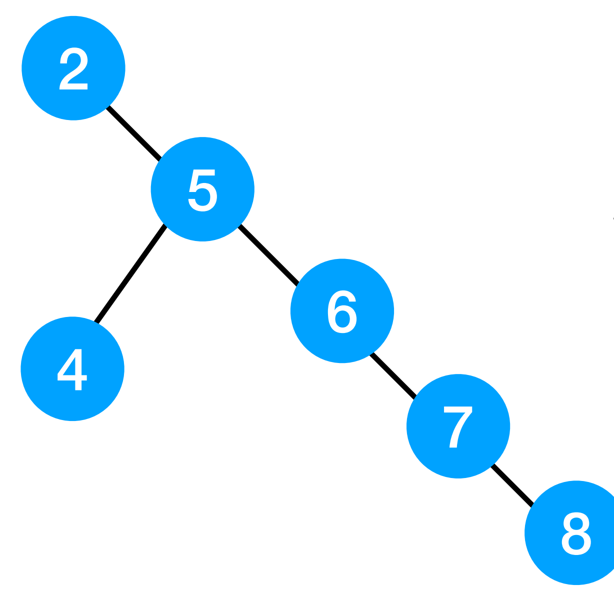


Traversals of Binary Search Trees

- Preorder traversal:
    - T_1: 6, 4, 2, 5, 7, 8
    - T_2: 2, 5, 4, 6, 7, 8
- Postorder traversal:
    - T_1: 2, 5, 4, 8, 7, 6
    - T_2: 4, 8, 7, 6, 5, 2
- **Inorder traversal:
    - T_1: 2, 4, 5, 6, 7, 8
    - T_2: 2, 4, 5, 6, 7, 8
- *Remark*: According to the binary-search-tree property, an *inorder traversal* of a binary search tree T visits the elements in *sorted order*



T_1



T_2

Common Operations on Binary Search Trees

- Common operations performed a binary search tree *T*:
  - `search(k, T):` return an object of a node whose key = k, if one exists, in the tree *T*
  - `minimum(T):` return an object of a node whose key is a minimum in the tree *T*
  - `maximum(T):` return an object of a node whose key is a maximum in the tree *T*
  - `successor(x, T):` return an object of the successor of node *x* (the node with the smallest key greater than `x.key`), if it exits
  - `predecessor(x, T):` return an object of the predecessor of node *x* (the node with the largest key smaller than `x.key`), if it exits

Operation: Search

- `search(k, T)`: return an object of a node whose key = k, if one exists, in the tree *T*

```
struct node
{
    int key;
    struct node* parent;
    struct node* left;
    struct node* right;
};
```

```
struct node* search(int key, struct node* node)
{
    if ((node == NULL) || (key == node->key))
        return node;
    if (key < node->key)
        return search(key, node->left);
    else
        return search(key, node->right);
}
```

```
int main()
{
    struct node* node1 = createNode();
    node1->key = 5;
    expandExternal(node1);
    node1->left->key = 2;
    node1->right->key = 6;
    expandExternal(node1->left);
    node1->left->left->key = -1;
    node1->left->right->key = 3;
    inorder(node1);
    printf("\n %d", search(6,node1)->key);
    return 0;
}
```

Operation: Minimum

- `minimum(T):` return a a node whose key is a minimum in the tree *T*

```
struct node
{
    int key;
    struct node* parent;
    struct node* left;
    struct node* right;
};
```

```
struct node* minimum(struct node* node)
{
    while(node->left != NULL)
        node = node->left;
    return node;
}
```

```
int main()
{
    struct node* node1 = createNode();
    node1->key = 5;
    expandExternal(node1);
    node1->left->key = 2;
    node1->right->key = 6;
    expandExternal(node1->left);
    node1->left->left->key = -1;
    node1->left->right->key = 3;
    printf("\n %d", minimum(node1)->key);
    return 0;
}
```

Operation: Maximum

- `maximum(T):` return an object of a node whose key is a maximum in the tree *T*

```
struct node
{
    int key;
    struct node* parent;
    struct node* left;
    struct node* right;
};
```

```
struct node* maximum(struct node* node)
{
    while(node->right != NULL)
        node = node->right;
    return node;
}
```

```
int main()
{
    struct node* node1 = createNode();
    node1->key = 5;
    expandExternal(node1);
    node1->left->key = 2;
    node1->right->key = 6;
    expandExternal(node1->left);
    node1->left->left->key = -1;
    node1->left->right->key = 3;
    printf("\n %d", maximum(node1)->key);
    return 0;
}
```

Operation: Successor

- `successor(x, T):` return an object of the successor of a node *x* (the node with the smallest key greater than `x.key`), if it exits

```
struct node* successor(struct node* node)
{
    if(node->right != NULL)
        return minimum(node->right);
    struct node* ancestor = node->parent;
    while((ancestor != NULL) && (node == ancestor->right))
    {
        node = ancestor;
        ancestor = node->parent;
    }
    return ancestor;
}
```

```
struct node
{
    int key;
    struct node* parent;
    struct node* left;
    struct node* right;
};
```

```
int main()
{
    struct node* node1 = createNode();
    node1->key = 5;
    expandExternal(node1);
    node1->left->key = 2;
    node1->right->key = 6;
    expandExternal(node1->left);
    node1->left->left->key = -1;
    node1->left->right->key = 3;
    inorder(node1);

    struct node* node3 = search(3,node1);
    printf("\n %d", successor(node3)->key);

    return 0;
}
```

Operation: Predecessor

- `predecessor(x, T):` return an object of the predecessor of node *x* (the node with the largest key smaller than `x.key`), if it exits

```c
struct node* predecessor(struct node* node)
{
    if(node->left != NULL)
        return maximum(node->left);
    struct node* ancestor = node->parent;
    while((ancestor != NULL) && (node == ancestor->left))
    {
        node = ancestor;
        ancestor = node->parent;
    }
    return ancestor;
}
```

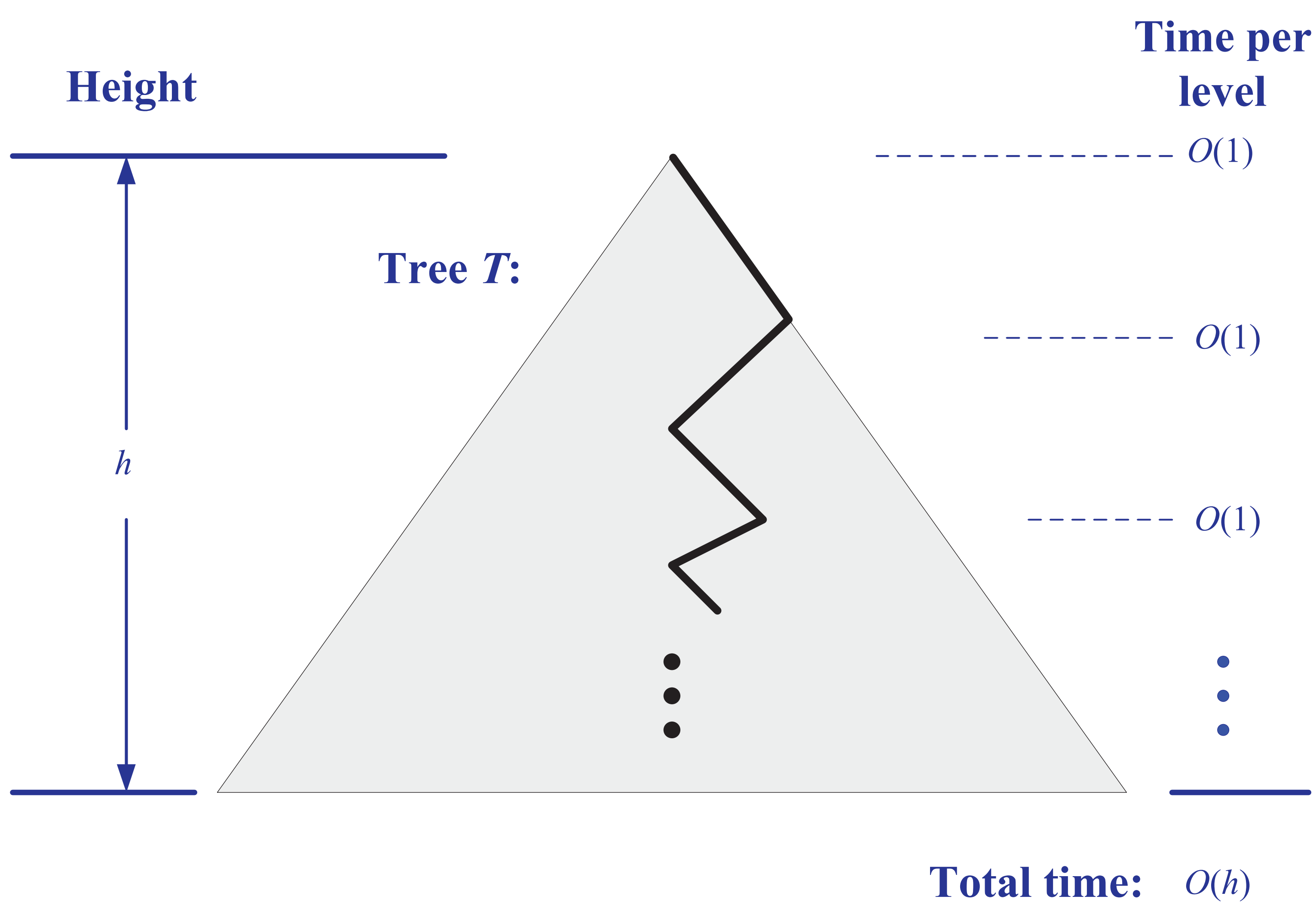
```c
struct node
{
    int key;
    struct node* parent;
    struct node* left;
    struct node* right;
};
```

```c
int main()
{
    struct node* node1 = createNode();
    node1->key = 5;
    expandExternal(node1);
    node1->left->key = 2;
    node1->right->key = 6;
    expandExternal(node1->left);
    node1->left->left->key = -1;
    node1->left->right->key = 3;
    inorder(node1);

    struct node* node6 = search(6,node1);
    printf("\n %d", predecessor(node6)->key);

    return 0;
}
```

Complexity of Operations on Binary Search Trees

Operations	Complexity
`search`	$O(h)$
`minimum`	$O(h)$
`maximum`	$O(h)$
`successor`	$O(h)$
`predecessor`	$O(h)$
	Remark: h is the hight of a binary tree - At worst, h can be n-1 - At best, h can be log(n+1)–1

Complexity of Operations on Binary Search Trees (2)



Binary Trees's Properties

- **Proposition 1**: Let T be a nonempty binary tree. Let n, n_E , n_I and h denote the number of nodes, number of external nodes, number of internal nodes, and height of T, respectively. Then T has the following properties:

1. $h+1 \leq n \leq 2^{h+1}-1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n+1)-1 \leq h \leq n-1$