Scuola universitaria professionale
della Svizzera italiana
SUPSI

Università
della
Svizzera
italiana

Istituto Dalle Molle di studi sull'intelligenza artificiale

Algorithms and Data Structures

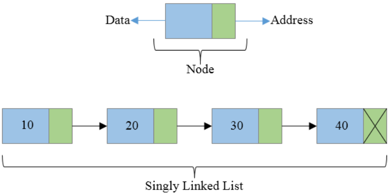
Lists and Trees

Matteo Salani
matteo.salani@idsia.ch

Dynamic Data Structures

Dynamic data structures are necessary when the amount of data to be managed is unknown a-priori and have the flexibility to grow or shrink in size, enabling a programmer to control how much memory is utilized.

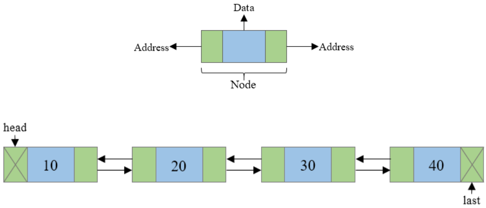
Singly linked lists




```
struct node {
    int data;              // Data
    struct node * next; // Address
};
```

image:https://codeforwin.org/

Double linked lists




```
struct node {
    int data;              // Data field
    struct node * prev; // Address of previous node
    struct node * next; // Address of next node
};
```

image:https://codeforwin.org/

Lists

Advantages:

- Lists are easy to implement
- Do not need to move elements after deletion
- Require a modest amount of additional memory (pointers)

Disadvantages:

- Inefficient, $O(n)$ for insertion and deletion
- No reverse visiting for singly linked list

Can be used to implement **Stacks and Queues**

Trees

More clever data structures exist to overcome the linear complexity for searching, inserting and deleting of lists.

A **Tree**:

- Is a collection of nodes
- It can be empty
- (recursive definition) It consists in a root node R and possibly empty set of subtrees whose roots are connected to node R by edges.

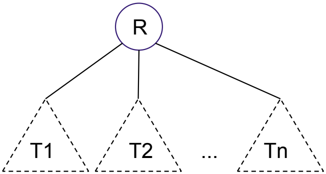


Figure: A generic tree

Trees

Terminology

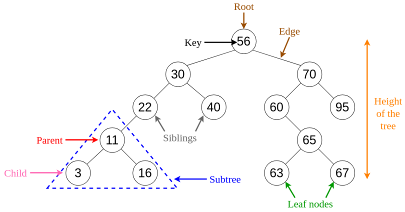


Figure: A generic tree

- Each node has a **parent** except the root
- Any node can have an arbitrary number of **children**
- **Leaves** are nodes with no children
- **Depth of a node** length of the unique path from the root to the node
- **Height of a tree** equal to the depth of the deepest leaf

Binary Trees

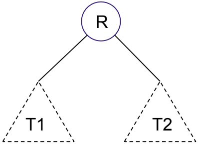


Figure: A binary tree

```
struct node {
    int data;             // Data field
    struct node * left; // Address of the left child
    struct node * right; // Address of the right child
};
```

- A tree in which no node can have more than two children
- The depth of an average binary tree is smaller than N (number of nodes)
- Worst case depth is $N-1$ (i.e. degenerates into a list)

Tree traversal

Tree traversals are used to explore all nodes of a tree.
Pre-order first process the data in a node and then visits the left subtree and the right subtree.

```
void pre-order(node * n){
  if (n!=NULL){
    process(n->data);
    pre-order(n->left);
    pre-order(n->right);
  }
}
```

Example Pre-Order

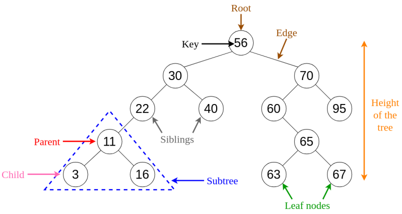


Figure: A binary tree

Output sequence:

56 – 30 – 22 – 11 – 3 – 16 – 40 – 70 – 60 – 65 – 63 – 67 – 95

In-order

In order first visits the left subtree, then process the data in the node and then visits the right subtree.

```
void in-order(node * n){
  if (n!=NULL){
    in-order(n->left);
    process(n->data);
    in-order(n->right);
  }
}
```

Example In-Order

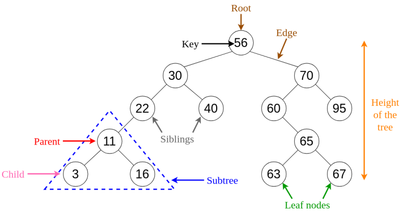


Figure: A binary tree

Output sequence:

3 – 11 – 16 – 22 – 30 – 40 – 56 – 60 – 63 – 65 – 67 – 70 – 95

Post-order

Post order first visits the left subtree, then the right subtree and finally process the data in the node.

```
void post-order(node * n){
  if (n!=NULL){
    process(n->data);
    post-order(n->left);
    post-order(n->right);
  }
}
```

Example Post-Order

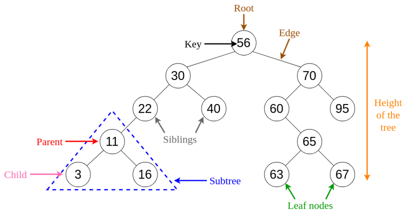


Figure: A binary tree

Output sequence:

3 – 16 – 11 – 22 – 40 – 30 – 63 – 67 – 65 – 60 – 95 – 70 – 56

Binary Search Tree

In case data possess an ordinal property (e.g. integers) we can store key so that searching, inserting and deleting is efficient.

We impose the Binary **Search** Tree property:

- For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X

Binary Search Tree Example

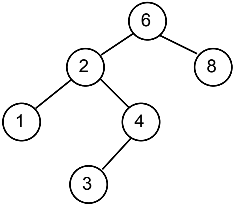


Figure: A binary search tree

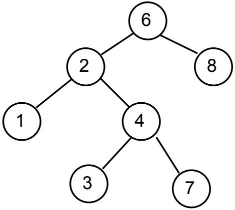


Figure: **Not** a binary search tree

Average depth of a node is $O(\log N)$, the maximum depth of a node is $O(N)$

Searching a key in a BST

Searching an element is a recursive procedure:

```
node * search(int key, node * n){
  if (n==NULL)
    return NULL;
  else if (key < n->data)
    return search(key, n->left);
  else if (key > n->data)
    return search(key, n->right);
  else
    return n; // Found
}
```

- Time complexity is $O(height - of - the - tree)$
- In-order traversal of a BST returns keys sorted in increasing order

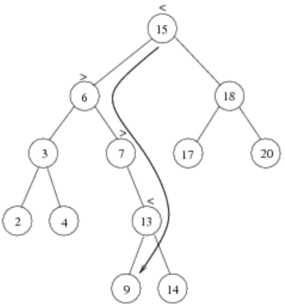


Figure: Searching for 9

Find the smalles/biggest element in a BST

Visit left/right subtrees as long as there are elements:

```
node * findMin(node * n){
  if (n==NULL)
    return NULL;
  else if (n->left == NULL)
    return n; //Found
  return findMin(n->left);
}
```

- Time complexity is $O(height - of - the - tree)$
- findMax is symmetric

Insert an element in a BST

Similar to search procedure:

```
node * insert(node * n, const node * el){
  if (n==NULL)
    return el;
  else if (el->data < n->data)
    n->left = insert(n->left, el);
  else if (el->data > n->data)
    n->right = insert(n->right, el);
  return n;
}
```

- Time complexity is $O(height - of - the - tree)$
- Example (blackboard) Insert: 50, 30, 20, 40, 70, 60, 80

Deleting an element in a BST

We need to take care of the **children** of the node to be deleted.

- The BST property must be mantained!
- Case 1. The element is a leaf: delete immediately
- Case 2. The element has just one child: adjust the pointer and delete
- Case 3. The element has two children: replace the element with the smallest element of the right subtree (biggest of the left subtree), then we are back in case 2
- Time complexity is $O(height - of - the - tree)$
- Example (blackboard) Insert: 50, 30, 20, 40, 70, 60, 80 Delete: 80, 70, 50

Delete an element in a BST - code

```
node* deleteNode(node* n, int key)
{
    if (n == NULL)
        return n;
    if (key < n->data)
        n->left = deleteNode(n->left, key);
    else if (key > n->key)
        n->right = deleteNode(n->right, key);
    // node to be deleted Found!
    else {
        if (n->left==NULL and n->right==NULL)
            return NULL;
        else if (n->left == NULL) {
            node* tmp = n->right;
            free(n);
            return tmp;
        }
        else if (n->right == NULL) {
            node* tmp = n->left;
            free(n);
            return tmp;
        }
        // node with two children
        node* tmp = findMin(n->right);
        n->key = tmp->key;
        n->right = deleteNode(n->right, tmp->key);
    }
    return n;
}
```